Lecture Notes in Computer Science 505

E. H. L. Aarts J. van Leeuwen
M. Rem (Eds.)

PARLE '91 Parallel Architectures and Languages Europe

Springer-Verlag Berlin Heidelberg GmbH

Lecture Notes in Computer Science 505

E. H. L. Aarts J. van Leeuwen
M. Rem (Eds.)

PARLE '91
Parallel Architectures and Languages Europe

Volume I: Parallel Architectures and Algorithms
Eindhoven, The Netherlands, June 10-13, 1991
Proceedings

Springer-Verlag
Berlin Heidelberg GmbH

Series Editors

Gerhard Goos
GMD Forschungsstelle
Universität Karlsruhe
Vincenz-Priessnitz-Straße 1
W-7500 Karlsruhe, FRG

Juris Hartmanis
Department of Computer Science
Cornell University
Upson Hall
Ithaca, NY 14853, USA

Volume Editors

Emile H. L. Aarts
Philips Research Laboratories
P.O. Box 80.000, 5600 JA Eindhoven, The Netherlands

Jan van Leeuwen
Department of Computer Science, University of Utrecht
Padualaan 14, 3584 CH Utrecht, The Netherlands

Martin Rem
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

CR Subject Classification (1991): C.1-4, D.1, D.3-4, F.1-3

ISBN 978-3-662-23206-4 ISBN 978-3-662-25209-3 (eBook)
DOI 10.1007/978-3-662-25209-3

Originally published by Springer-Verlag Berlin Heidelberg New York in 1991

2145/3140-543210 - Printed on acid-free paper

Preface

The innovative progress in the development of large- and small-scale parallel computing systems and their increasing availability have caused a sharp rise in interest in the scientific principles that underlie parallel computation and parallel programming. The biannual "Parallel Architectures and Languages Europe" (PARLE) conferences aim at presenting current research material on all aspects of the theory, design, and application of parallel computing systems and parallel processing. At the same time, the goal of the PARLE conferences is to provide a forum for researchers and practitioners to exchange ideas on recent developments and trends in the field of parallel computing and parallel programming. The first two conferences, PARLE '87 and PARLE '89, have succeeded in meeting this goal and made PARLE a conference that is recognized worldwide in the field of parallel computation.

PARLE '91 again offers a wealth of high-quality research material for the benefit of the scientific community. Compared to its predecessors, the scope of PARLE '91 has been broadened so as to cover the area of parallel algorithms and complexity, in addition to the central themes of parallel architectures and languages.

The proceedings of the PARLE '91 conference contain the text of all contributed papers that were selected for the programme and of the invited papers by leading experts in the field. The proceedings are presented in two volumes:

Volume I: Parallel Architectures and Algorithms

Volume II: Parallel Languages

In the Call for Papers of PARLE '91 papers were solicited describing original research in the areas of parallel architectures and systems, parallel programming, parallel languages, parallel algorithms and complexity, and applications of parallelism (including for example, neural computing). We greatly appreciate the efforts of the authors who submitted papers to this year's conference; a record number of 161 submissions were received. To maintain the high technical level of PARLE all papers went through an intensive review process. We extend our sincere thanks to the Programme Committee and to the referees who were consulted in the reviewing process, for the arduous task of selecting the programme out of these papers. Their dedication

and professionalism have greatly contributed to the technical quality of the PARLE '91 conference.

No conference of this size can be organized without the contributions of many dedicated people. As for the previous PARLE conferences, the organization of the PARLE '91 conference was again handled by the Philips Research Laboratories. A special tribute in this respect is due to Fred Robert and Frank Stoots, for their skilled and efficient handling of all organizational details of the conference over the past year.

We feel that the PARLE '91 conference has once again succeeded in bringing together a wealth of material on the theme of parallel computing. We therefore hope that these proceedings will contribute to the tradition of European research on Parallel Architectures and Languages.

Eindhoven/Utrecht
June 1991

Emile H.L. Aarts
Jan van Leeuwen
Martin Rem

Scientific Programme Committee

G. Agha, U.S.A.
P.M.G. Apers, the Netherlands
J.-P. Banâtre, France
H.P. Barendregt, the Netherlands
U. Baron, Israel
J.-C. Bermond, France
M. Broy, Germany
W.J. Dally, U.S.A.
W. Damm, Germany
J. Díaz, Spain
D. Gelernter, U.S.A.
A. Gibbons, United Kingdom
J. Gruska, Germany
J. Gurd, United Kingdom
D. Harel, Israel
S. Haridi, Sweden
L.O. Hertzberger, the Netherlands
W.J.-P. Jalby, France
Th. Johnsson, Sweden
Ph. Jorrand, France
V.E. Kotov, U.S.S.R.
F.E.J. Kruseman Aretz, the Netherlands
J.K. Lenstra, the Netherlands
H. Mühlenbein, Germany
E.A.M. Odijk, the Netherlands
E.-R. Olderog, Germany
I. Parberry, U.S.A.
W.P. de Roever, Germany
J.L.A. van der Snepscheut, U.S.A.
J.-C. Syre, France
P.C. Treleaven, United Kingdom
K. Ueda, Japan
M. Valero, Spain
D.H.D. Warren, United Kingdom
P. Wodon, Belgium

Contents Volume I

Invited Lectures

Submitted Presentations

Contents Volume II

Invited Lectures

Submitted Presentations

List of Referees

Aerts, A.T.M.
America, P.
Anishev, P.A.
Annot, J.K.
Asveld, P.R.J.
Augusteijn, A.
Augustsson, L.
Beaudoing, B.
Beaumont, T.
Beemster, M.
Belvide, R.
Bekkers, Y.
Benveniste, M.
Best, E.
Bjornson, R.
Blanken, H.M.
Boer, F. de
Bosc, P.
Bost, B.
Bronnenberg, W.
Breu, R.
Brix, H.
Carlsson, M.
Carriero, N.
Chassin de Kergommeaux, J. de
Cheese, A.
Cherkasova, L.A.
Ciancarini, P.
Cooper, M.D.
Cortes, U.
Creveuil, C.
Dederichs, F.
Delgado-Rannauro, S.A.
Dendorfer, C.
Dennison, L.R.
Desel, J.
Dutra, J.
Eekelen, M.C.J.D. van
Ellis, J.
Esparza, J.
Feijen, W.H.J.
Feijs, L.M.G.
Fraigniaud, P.
Forshaw, M.R.B.
Fuchs, M.
Gabarró, J.
Gamané, B.
Gerth, R.
Godefroid, P.
Goltz, U.
Gonzalez, A.
Gribomont, E.P.
Gritzner, T.F.
Gupta, G.
Haaften, P.J.M. van
Hausman, B.
Helmink, L.
Hesselink, W.H.
Hofman, R.F.H.
Hofstee, H.P.
Holden, M.
Horwat, W.
Houck, C.
Houtsma, M.A.W.
Hulin, G.
Hulshof, B.J.A.
Ibañez, M.B.
Inamura, Y.
Ingels, Ph.
Jansen, P.G.
Janson, S.
Jantzen, M.
Jard, C.
Jegov, Y.

Joseph, M.
Kalker, A.A.C.M.
Kamerbeek, J.
Kaplan, S.
Karpov, Ju.G.
Kausche, A.
Kennaway, J.R.
Keen, J.
Keckler, S.W.
Kempf, P.
Khebbal, S.
Koot, J.
Korst, J.H.M.
Labarta, J.
Laborelli, L.
Langendoen, K.G.
Le Metayer, O.
Lethin, R.
Liang Liang, L.
Lukkien, J.J.
Madelaine, E.
Martin, G.R.
Milikowski, R.
Miriyala, S.
Monien, B.
Morrison, J.P.
Muller, H.L.
Murakami, M.
Mussi, Ph.
Nakata, T.
Nepomniaschy, V.A.
Nijholt, A.
O'Boyle, M.F.
Omtzigt, F.T.L.
Overeinder, B.J.
Paech, B.
Panwar, R.
Tubella, J.
Parrott, D.J.
Pazat, J.-L.
Peikenkamp, T.
Peleg, D.
Peters, J.G.
Philippe, B.J.
Plasmeijer, M.J.
Priol, T.
Quinton, P.
Raina, S.
Ratcliffe, M.J.
Raucher, R.
Raynal, M.
Remscher, R.
Renardel de Lavalette, G.R.
Roberts, G.
Roelants, D.
Rokusawa, K.
Rouwce, P.A.
Sanders, B.
Santos-Costa, V.
Schalij, F.D.
Scholten, J.
Schoute, A.L.
Sedukhin, S.
Serna, M.J.
Seznec, A.
Sijstermans, F.
Sindaha, R.
Spertus, E.
Sun, C.Z.
Stomp, F.A.
Streicher, P.
Syska, M.
Taubner, D.
Tel, G.
Thibault, O.
Watson, I.

Twist, R.A.H. van
Valk, R.
Venkatasubramanian, N.
Veron, A.
Vissers, K.A.
Vlot, M.C.
Vree, W.G.
Vries, F-J. de
Wallach, D.
Weber, R.
Wester, R.H.H.
Wei, M.
Wiedermann, J.
Wilschut, A.N.
Wolper, P.
Yang, R.
Yamasaki, S.
Zakaria, L.A.H.J.

Lecture Notes in Computer Science 506

Edited by G. Goos and J. Hartmanis

Advisory Board: W. Brauer D. Gries J. Stoer

E. H. L. Aarts J. van Leeuwen
M. Rem (Eds.)

PARLE '91
Parallel Architectures and Languages Europe

Volume II: Parallel Languages
Eindhoven, The Netherlands, June 10-13, 1991
Proceedings

Springer-Verlag
Berlin Heidelberg GmbH

Contents Volume II

Invited Lectures

Submitted Presentations

Contents Volume I

Invited Lectures

Submitted Presentations

List of Referees

Aerts, A.T.M.
America, P.
Anishev, P.A.
Annot, J.K.
Asveld, P.R.J.
Augusteijn, A.
Augustsson, L.
Beaudoing, B.
Beaumont, T.
Beemster, M.
Belvide, R.
Bekkers, Y.
Benveniste, M.
Best, E.
Bjornson, R.
Blanken, H.M.
Boer, F. de
Bosc, P.
Bost, B.
Bronnenberg, W.
Breu, R.
Brix, H.
Carlsson, M.
Carriero, N.
Chassin de Kergommeaux, J. de
Cheese, A.
Cherkasova, L.A.
Ciancarini, P.
Cooper, M.D.
Cortes, U.
Creveuil, C.
Dederichs, F.
Delgado-Rannauro, S.A.
Dendorfer, C.
Dennison, L.R.
Desel, J.
Dutra, J.
Eekelen, M.C.J.D. van
Ellis, J.
Esparza, J.
Feijen, W.H.J.
Feijs, L.M.G.
Fraigniaud, P.
Forshaw, M.R.B.
Fuchs, M.
Gabarró, J.
Gamané, B.
Gerth, R.
Godefroid, P.
Goltz, U.
Gonzalez, A.
Gribomont, E.P.
Gritzner, T.F.
Gupta, G.
Haaften, P.J.M. van
Hausman, B.
Helmink, L.
Hesselink, W.H.
Hofman, R.F.H.
Hofstee, H.P.
Holden, M.
Horwat, W.
Houck, C.
Houtsma, M.A.W.
Hulin, G.
Hulshof, B.J.A.
Ibañez, M.B.
Inamura, Y.
Ingels, Ph.
Jansen, P.G.
Janson, S.
Jantzen, M.
Jard, C.
Jegov, Y.

Joseph, M.
Kalker, A.A.C.M.
Kamerbeek, J.
Kaplan, S.
Karpov, Ju.G.
Kausche, A.
Kennaway, J.R.
Keen, J.
Keckler, S.W.
Kempf, P.
Khebbal, S.
Koot, J.
Korst, J.H.M.
Labarta, J.
Laborelli, L.
Langendoen, K.G.
Le Metayer, O.
Lethin, R.
Liang Liang, L.
Lukkien, J.J.
Madelaine, E.
Martin, G.R.
Milikowski, R.
Miriyala, S.
Monien, B.
Morrison, J.P.
Muller, H.L.
Murakami, M.
Mussi, Ph.
Nakata, T.
Nepomniaschy, V.A.
Nijholt, A.
O'Boyle, M.F.
Omtzigt, F.T.L.
Overeinder, B.J.
Paech, B.
Panwar, R.
Tubella, J.
Parrott, D.J.
Pazat, J.-L.
Peikenkamp, T.
Peleg, D.
Peters, J.G.
Philippe, B.J.
Plasmeijer, M.J.
Priol, T.
Quinton, P.
Raina, S.
Ratcliffe, M.J.
Raucher, R.
Raynal, M.
Remscher, R.
Renardel de Lavalette, G.R.
Roberts, G.
Roelants, D.
Rokusawa, K.
Rouwce, P.A.
Sanders, B.
Santos-Costa, V.
Schalij, F.D.
Scholten, J.
Schoute, A.L.
Sedukhin, S.
Serna, M.J.
Seznec, A.
Sijstermans, F.
Sindaha, R.
Spertus, E.
Sun, C.Z.
Stomp, F.A.
Streicher, P.
Syska, M.
Taubner, D.
Tel, G.
Thibault, O.
Watson, I.

Twist, R.A.H. van
Valk, R.
Venkatasubramanian, N.
Veron, A.
Vissers, K.A.
Vlot, M.C.
Vree, W.G.
Vries, F-J. de
Wallach, D.
Weber, R.
Wester, R.H.H.
Wei, M.
Wiedermann, J.
Wilschut, A.N.
Wolper, P.
Yang, R.
Yamasaki, S.
Zakaria, L.A.H.J.

Parallel Evaluation of Functional Programs: The $\langle \nu, G \rangle$-machine approach (Summary)

Thomas Johnsson
Department of Computer Science
Chalmers University of Technology
S-412 96 Göteborg, Sweden

For a number of years this author, together with Lennart Augustsson, have been developing fast implementations of lazy functional languages, based on graph reduction, for ordinary (sequential) computers. Our approach can be summarised very briefly as follows.

Our ideas stem from Turner's S, K, I standard combinator reduction approach [Tur79]. But instead of using a standard, fixed set of combinators, a compiler transforms the program into a new set of specialised combinators, or 'super-combinators' [Hug82]. This transformation process is called *lambda lifting* [Joh85]. Each of these super-combinators are then compiled into machine code for the machine at hand, this code implements the corresponding graph rewrite rule. In other words, the compiler constructs a specialised, machine-language coded combinator interpreter from each program. However, rather than compiling each combinator into machine code directly, we first compile them into code for an abstract machine, the *G-machine* [Joh84]. Also, rather than letting the code rewrite the graph for a combinator application into the graph of the right hand side of the combinator definition, quite a lot of improvements to this scheme is possible. The G-machine is a convenient abstraction for expressing these improved compilation schemes.

An overview of the techniques used in our compiler for Lazy ML can be found in [Aug84] and [AJ89b]. The compilation of pattern matching into efficient code is described in [Aug85]. Our method of lambda lifting is described in [Joh85], and the G-machine is described in [Joh84]. The approach to machine code generation used in the Lazy ML compiler is described in [Joh86].

Parallel computers, consisting of dozens, hundreds, or thousands of processors connected to either a shared memory or a message passing network, are now becoming available on the marketplace. Recently, we have done work on extending the G-machine techniques to perform parallel graph reduction on such computers [AJ89a], to obtain real speedup compared to the sequential implementation.

It is possible to modify the sequential G-machine into a parallel one straighforwardly, by having multiple threads of control each with one or two stacks, all of which perform graph reduction in a common graph — this is the shared memory model. Such systems have been designed and implemented by Maranget [Mar91] and [Geo89]. This is also the approach taken in the GRIP project [Jon87, JCS89].

However, there are some properties of the standard G-machine that made us want to try a different approach for a parallel implementation. Firstly, in the G-machine, when reduction of a function application starts, the arguments of the application (either in the

form of a chain of binary application nodes, or a vector application node) are moved to the stack, and when reduction is finished the result is moved back into the heap by updating the root application node with the value of the function application. This seems like a lot of unnecessary data movement when the datum could have been accessed from the node in the first place (this reasoning has nothing to do with parallelism, of course). Secondly, the prospect of having to manage a cactus stack was not very appealing, we wanted something simpler.

In the machines we would like to consider, and in particular the machine we have implemented our parallel graph reducer on, the Sequent Symmetry™, a memory reference into the heap has the same cost as a reference into a stack, since they reside in the same (shared) memory. Thus the cost of moving a word to the heap while building a node is the same as pushing a word into the stack.

Thus, in the abstract machine we have designed, called the $\langle \nu, G \rangle$-machine, function applications are represented by *frame nodes*. A frame node holds the arguments of the function application and a pointer to the code for the function being applied, but in addition also contains enough space for temporaries needed for reduction of the function application. Figure 1 shows what happens when EVAL is called: the 'current point of reduction' is moved to the frame node to be evaluated, and a 'dynamic link' field is set to point back to the frame which called did the EVAL. Thus, instead of an ordinary stack we have a linked list of stack frames. In the parallel case, we have many points of reduction. For further details of the abstract machine, see [AJ89a].

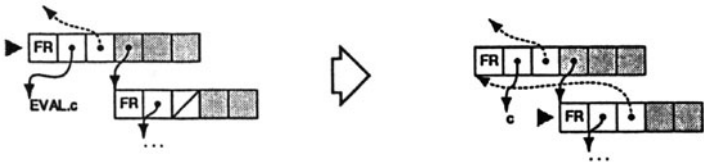


Figure 1: Calling EVAL to reduce a frame node.

To be fair, the stack model has some advantages too. The spineless G-machine [BRJ88], which offers a more general and efficient tail call mechanism than the 'standard G-machine' in particular when dealing with higher order functions, requires essentially an arbitrarily big stack. However, Lester [Les89] has devised an analysis technique based on abstract interpretation, to determine the maximum size a stack might have under the 'spineless' evaluation regime. Thus it would be possible to merge the $\langle \nu, G \rangle$-machine model with the 'spineless' model of execution, by allocating a frame node of the required maximum size.

Both the stack model and the frame node model have their advantages, and it is too early to nominate an overall winner.

So far, to introduce parallelism in the LML programs the programmer has to write *spark* annotations [CJ86] in the programs explicitly. The spark annotation is advisory: if there is a processor available then it may evaluate the sparked expression, otherwise the process really needing the value will evaluate it itself.

Code generation now works rather differently from the way it was described in [AJ89a]. Code is generated by first translating the combinators into three-address form, with liberal use of temporary names. We illustrate this with the code for the combinator `f x y z = x y`, which is:

```
funstart f 3        start of f, which takes 3 args
load  t0,0(nu)      load z from frame into t1
load  t1,1(nu)      load y from frame into t2
load  t2,2(nu)      load x from frame into t3
move  t1,t4
move  t4,t5
store t5,0(nu)      store the arg y of the application into the frame
eval  t2            evaluate x, the function
move  t2,t3
do    t3,1          tail call, function is x, one arg in frame
```

The code for a combinator starts by loading all arguments into temporaries from the frame node. Then in the example above the argument y of the tail call, is moved into the current frame at the location of the last argument; the function x of the tailcall is evaluated into function form, and finally the tail call is performed with the general tail call instruction do.

This 'raw' code is then subjected to various improvement transformations; for instance, the loads and stores are moved around to minimise the number om live variables across eval. Finally, temporaries are bound to machine registers. The resulting code is:

```
funstart f 3        start of f, which takes 3 args
load  r0,2(nu)      load x from frame into register r0
eval  r0            evaluate x, the function, in r0
load  r4,1(nu)      load y from frame into register r4 ...
store r4,0(nu)      ... and store y, the arg of the application into the frame
do    r0,1          tail call, function is x, one arg in frame
```

From this code the actual machine code is generated. A notable feature of the generated code is that we have abandoned the method of coding the tag as a pointer, either to a table (as described in [Joh86]) or to code directly, as in the spineless tagless G-machine [JS89]. Instead, the tag word contains various tag bits. The reason comes from two observations: firstly, most of the time when doing eval the node is allready canonical — according to measurements 80% of the time is typical. Secondly, in most modern architectures with instruction prefetch, it is rather costly to break the sequential flow of control. We therefore implement eval with code that tests a canonical-bit in the tag field of the node to be evaluated; if canonical the next instruction is executed, and only if it is not canonical does a jump occur to code that performs the actual call to the eval routine. The call to eval is surrounded by code that stores and reloads the content of live registers. Our

implementation of the parallel $\langle \nu, G \rangle$-machine is for the Sequent Symmetry, a bus-based shared memory machine. The architecture supports up to 30 processors connected to the bus; our machine has 16 processors. This machine has some features that helps very much in the implementation of the parallel $\langle \nu, G \rangle$-machine, for instance, any cell in the memory can be used as an atomic lock.

At the moment of writing this, a new garbage collector is being tested [Rõj91]. It is an improved version of the Appel-Ellis-Li garbage collector, which is an efficient real-time copying garbage collector which runs concurrently with the mutator processes. Rõjemo has extended it also to collect processes which have become garbage.

Since the publication of [AJ89a] we have improved the performance somewhat due to the improved code generation method, as described briefly above. The improvement is

about 25% for code purely sequential code, but for parallel programs the improvement is less than that – depending on how big a proportion of the time is spent in activities like syncronisation, task switching etc. Figure 2 shows the current speedup charts for three benchmark programs. Garbage collection time is not included in these figures.

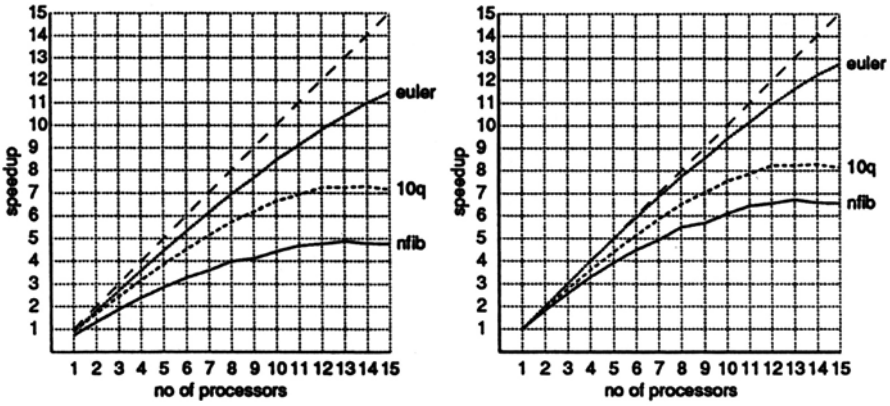


Figure 2: Speedup graphs for three benchmark programs: The left graph shows the speedup relative to one processor, the right graph shows the speedup relative to the 'standard G-machine' in the LML compiler.

References

[AJ89a] L. Augustsson and T. Johnsson. Parallel Graph Reduction with the $\langle \nu, G \rangle$-machine. In *Proceedings of the 1989 Conference on Functional Languages and Computer Architecture*, pages 202–213, London, England, 1989.

[AJ89b] L. Augustsson and T. Johnsson. The Chalmers Lazy-ML Compiler. *The Computer Journal*, 32(2):127 – 141, 1989.

[Aug84] L. Augustsson. A Compiler for Lazy ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 218–227, Austin, 1984.

[Aug85] L. Augustsson. Compiling Pattern Matching. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, Sweden, November 1987.

[BRJ88] G. Burn, J. Robson, and S. Peyton Jones. The Spineless G-machine. In *Proceedings of the 1988 ACM Symposium on Lisp and Functional Programming*, Snowbird, Utah, 1988.

[CJ86] C. Clack and S.L. Peyton Jones. The Four-Stroke Reduction Engine. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 220–232, 1986.

[Geo89] Lal George. An abstract machine for Parallel graph reduction. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, London, Great Britain, 1989.

[Hug82] R. J. M. Hughes. Super Combinators—A New Implementation Method for Applicative Languages. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 1–10, Pittsburgh, 1982.

[JCS89] S. L. Peyton Jones, C. Clack, and J. Salkild. High-Performance Parallel Graph Reduction. In *Proceedings of PARLE'89 Parallel Architectures and Languages Europe (Vol I)*, volume LNCS 365, pages 193–206. Springer-Verlag, June 1989.

[Joh84] T. Johnsson. Efficient Compilation of Lazy Evaluation. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 58–69, Montreal, 1984.

[Joh85] T. Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science 201, Nancy, France, 1985. Springer Verlag.

[Joh86] T. Johnsson. Code Generation from G-machine code. In *Proceedings of the workshop on Graph Reduction*, Lecture Notes in Computer Science 279, Santa Fe, September 1986. Springer Verlag.

[Joh87] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, Sweden, February 1987.

[Jon87] S. L. Peyton Jones. GRIP: A Parallel Graph Reduction Machine. In *Proceedings of the 1987 Conference on Functional Programming Languages and Computer Architecture*, Portland, Oregon, U.S.A, September 1987.

[JS89] S.L. Peyton Jones and Jon Salkild. The Spineless Tagless G-machine. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, London, Great Britain, 1989.

[Les89] David R. Lester. Stacklessness: compiling recursion for a distributed architecture. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, London, Great Britain, 1989.

[Mar91] Luc Maranget. GAML: a Parallel Implementation of Lazy ML. Technical report, Department of Computer Sciences, INRIA Rocquencourt, BP 105, 78153 Le Chesnay CEDEX, France, 1991.

[Röj91] Niklas Röjemo. A Concurrent Garbage Collector for the $\langle \nu, G \rangle$-machine. Technical report, Department of Computer Sciences, Chalmers University of Technology,S–412 96 Göteborg, February 1991.

[Tur79] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software—Practice and Experience*, 9:31–49, 1979.

Towards a Single Model of Efficient Computation in Real Parallel Machines

Pilar de la Torre *
Department of Computer Science
University of New Hampshire
Durham, New Hampshire 03824

and

Clyde P. Kruskal
Department of Computer Science
University of Maryland
College Park, Maryland 20742.

*The research of this author was supported in part by the National Science Foundation under Grant CCR-9010445.

Abstract

We propose a model of parallel computation, the *Y-PRAM*, that allows general parallel algorithms to be designed for a wide class of the parallel models. There are two basic quantities captured by the model, which the algorithm designer must leave open as parameters: *latency* and *bandwidth*.

We design *Y-PRAM* algorithms for solving several fundamental problems: parallel prefix, sorting, sorting numbers from a bounded range, and list ranking. We show that our model predicts, reasonably accurately, the actual known performances of several basic parallel models – PRAM, hypercube, mesh, and tree – when solving these problems.

1 Introduction

There is a large variety of models of parallel computation. Typically algorithms are developed for every model individually. Some understanding of parallel algorithm design has emerged: for example, we know that parallel prefix and sorting are important basic routines. And there are now many results showing separations between and equivalences of different models. But, there is no general framework for developing algorithms that apply to a variety of models. This paper attempts to introduce such a framework.

Our basic idea is to try to find a model of parallel computation with a limited number of parameters that capture the performance of an algorithm on a large class of parallel machines. With this tool, parallel algorithms can be developed once and for all, leaving open the few parameters for customization to a given machine. Substituting the parameter values will provide some idea of how well the particular machine can execute the algorithm. Furthermore, a compiler could use the algorithm to produce code for the machine.

If the model truly reflects parallel machine performances, it can provide a simple model for proving general upper and lower bounds. Similarly, by leaving in the parameters, one can often explicitly see the bottleneck of an algorithm.

We propose the Y-PRAM model of parallel computation, which captures two basic quantities of parallel computation: *latency* and *bandwidth*. We design *Y-PRAM* algorithms for solving several fundamental problems: parallel prefix, sorting, sorting numbers from a bounded range, and list ranking. We show that our model predicts, reasonably accurately, the actual known performances of several basic parallel models – PRAM, hypercube, mesh, and tree – when solving these problems.

A model that has many of the aspects of the Y-PRAM is the X-PRAM model of

Valiant [24]. Other related work is the investigation of latency by Aggarwal, Chandra, and Snir in their LPRAM and BPRAM models [1, 2].

1.1 Performance of parallel algorithms

We are restricting consideration to tightly coupled machines. We assume that moderately sized parallel computers will be used to solve very large problems [9] i.e., problems whose sequential times are much larger than the number of processors. We follow the work of Kruskal, Rudolph, and Snir [14], who define six classes of parallel algorithms.

From our point of view, the most interesting class is **EP** (Efficient Polynomially fast, or Efficient Parallel), which are those algorithms that achieve "polynomial" reduction in running time with constant "inefficiency". In order to define these classes formally, we need some definitions.

Let $t(N)$ be the sequential time for an algorithm to solve some problem of size N. Let $T_P(N)$ be the time for P processors to solve the same problem. A problem is in **EP** if

$$T_P(N) = O(t(N)/P) + P^{O(1)}.$$

The first term on the right side guarantees that the inefficiency is at most constant for large enough problems, and the second term guarantees that the constant inefficiency obtains whenever the sequential time is polynomially larger than the machine size (for some polynomial).

1.2 Models of parallel computation

The most powerful parallel models are the synchronous PRAM's (Parallel Random Access Machines), which allow the processors to access a common memory. The weakest variant is the EREW (Exclusive Read, Exclusive Write) PRAM; it does not allow concurrent accesses to a common location. The CREW (Concurrent Read, Exclusive Write) PRAM allows concurrent reads to a common location, but not concurrent writes. The CRCW (Concurrent Read, Concurrent Write) PRAM allows concurrent reads and concurrent writes to a common location. CRCW PRAM itself has many variants including the common, arbitrary, priority, and random models.

The next most powerful model is the DCM (Direct Connection Machine). It consists of autonomous unit-cost RAM's, each with its own local memory, that communicate by message passing – there is no shared memory. (This model is also known as a Fully

Connected Direct Connection Machine [12], Module Parallel Computer [19], and Seclusive PRAM [24]).

Finally, we have the sparse networks. Here each processor are connected only to a subset of other processors. In this paper, we will concentrate on the hypercube, butterfly, shuffle-exchange, complete binary tree, 2-dimensional mesh, and 1-dimensional mesh models. For conciseness, we will refer to a complete binary tree simply as a tree, a 2-dimensional mesh simply as a mesh, and a 1-dimensional mesh as a linear array. For a fuller discussion of models than given below see [15, 11, 24].

1.3 The Y-PRAM

Before presenting the *Y-PRAM* model, we provide a few definitions. A parallel machine is *recursively decomposable into halves* if the processors can be partitioned into two groups of equal size, so that each group is itself a submachine (and therefore recursively decomposable into halves). The *latency*, $\delta(P)$, of a parallel model is the maximum time to communicate between any pair of processors (or between any processor and shared memory). Typically, it will be the diameter of a sparse machine. The *bandwidth inefficiency*, $\beta(P)$, of a parallel model is the ratio of P to the bandwidth. We will discuss these machine properties below.

Y-PRAM MODEL: The *Y-PRAM*$[\beta, \delta]$ is a recursively decomposable model consisting of $P = 2^p$ RAM processors, which operate synchronously. Furthermore,

1. It has shared memory, which is partitioned among the processors so that each processor *owns* an equal amount.

2. Any submachine can block itself off from the rest of the machine. While blocked off, the processors within the submachine interleave *periods of computation* with *periods of memory access*; the accesses may be only to the portion of memory owned by the processors within the submachine.

3. Memory accesses may not conflict, i.e., there are no concurrent accesses to a common location. Thus, a blocked off submachine acts like a little EREW PRAM, except that the cost of accessing memory depends on δ and β.

4. The time for the processors within a submachine of size S to make a total of M memory accesses is
$$\Theta(\delta(S) + m + M\beta(S)/S),$$
where m is the most accesses of any one processor.

In many ways the *Y-PRAM* is an especially good model of the CM* machine at Carnegie-Mellon [23] and of the Cedar machine at the University of Illinois [16]. Both machines have clusters of processors, which produces a hierarchy of memories – CM* within the context of a tree and Cedar within the context of a PRAM.

We now discuss the three properties defined above. Many parallel models are recursively decomposable into halves, including the PRAM, linear array, and hypercube models. While many parallel models are not, most come close enough. For example a 2-dimensional mesh decomposes into four meshes, each once fourth the size of the original mesh. Since this decomposition produces a constant number of copies, it will suffice for our purposes.

A ring machine does not decompose at all: proper connected subgraphs of a ring are linear arrays. However, a linear array of size P can emulate a ring of size $2P$ with only a constant loss in efficiency, by having processor i of the linear array emulate processors i and $2P-1-i$ of the ring. Thus a ring of size P can emulate a recursively decomposable ring of size $2P$.

While, a tree machine decomposes naturally into two subtrees, the original root is not in either subtree. However, every decomposition of a (sub)tree uses more than half of its processors, so there is only a constant loss of efficiency.

More problematic, are the butterfly and shuffle-exchange models. They are not recursively decomposable; in fact, no machine that has only a constant number of connections per processor and that can permute in logarithmic time is recursively decomposable [18]. However, they have many aspects of recursively decomposable machines. In particular, ascend-descend algorithms [20], which are recursive, can be implemented efficiently on them. Furthermore, since many operations take polylogarithmic time, executing them globally for the whole machine is not much slower than working locally within submachines.

Now consider the latency. For sparse networks, it will typically be the diameter of the machine, i.e., the maximum distance between any pair of processors. The DCM model has constant diameter; the hypercube, shuffle-exchange, butterfly, and tree models have diameter $\Theta(\log P)$; and a d-dimensional mesh has diameter $\Theta(P^{1/d})$. The latency bound can be used to capture pipeline delays; many machines have a high start up time to send a message.

From our point of view, the latency bound is, in some sense, irrelevant: The class **EP** is invariant under latency, as long as the latency is a most polynomial in P (i.e., $\delta(P) \leq P^{O(1)}$) [KRS]. But, without taking latency into consideration our model would

predict that a linear array can solve many problems in polylogarithmic time. This seems too strange to ignore.

Our second parameter is related to the bandwidth. Consider the problem of permuting elements of an array of size N. For a DCM or sparse network, assume that each processor holds N/P elements of the array. Let $\tau(N)$ be the time it takes a machine to perform the permutation. The bandwidth of the machine is $\lim_{N\to\infty} N/\tau(N)$. The PRAM and DCM models have bandwidth $\Theta(P)$; the hypercube, shuffle-exchange, and butterfly models have bandwidth $\Theta(P/\log(P))$; the mesh has bandwidth $\Theta(\sqrt{P})$; and the tree and linear array models have bandwidth $\Theta(1)$. Bandwidth $\Theta(P)$ is optimal. We actually prefer to measure the ratio of how far from optimal a model is to make the bandwidth metric symmetric to the latency metric: The *bandwidth* bound, $\beta(P)$, is the ratio of P to the bandwidth, i.e., $\beta(P) = \lim_{N\to\infty} P\tau(N)/N$. So, for the PRAM and DCM models $\beta(P) = \Theta(1)$; for the hypercube, shuffle-exchange, and butterfly models $\beta(P) = \Theta(\log(P))$; for the mesh $\beta(P) = \Theta(\sqrt{P})$; and for the tree and linear array models $\beta(P) = \Theta(P)$. Different models obtain their bandwidths at different levels of parallelism, and the level depends on whether the routing algorithm is deterministic or probabilistic (see, for example, [21, 24] and references therein).

Although we have given here the standard latency and bandwidth bounds for the above models, these values should be considered flexible. For example, sometimes one considers PRAM's with a network between processors and memory, which have more than constant latency. Also, if wire lengths are taken into account, many models will have higher latency than indicated. Sometimes, each processor of hypercube is assumed to be able to send or receive messages on all of its wires at the same cycle. This version has bandwidth $\Theta(P)$, or $\beta(P) = \Theta(1)$.

2 Note on analysis

Sums of the form

$$\Theta(\sum_{j=1}^{\lg P} f(2^j)),$$

where the function $f(P)$ is typically constant, polylogarithmic, or polynomial in P, will often emerge in our analyses. For $f(P)$ of the form P^r, i.e., polynomial, the last term dominates, so the sum is $\Theta(P^r)$. For $f(P)$ of the form $(\log P)^s$, i.e., polylogarithmic, each term counts almost as much as the last, up to a constant factor, so the sum is $\Theta((\log P)^{s+1})$.

To keep track of these three main cases simultaneously we define

$$L[f(P)] = \frac{1}{f(P)} \sum_{j=1}^{\lg P} f(2^j).$$

Thus for $f(P)$ polynomial, $L[f(P)]$ is $\Theta(1)$, and for $f(P)$ constant or polylogarithmic, $L[f(P)]$ is $\Theta(\log P)$.

3 Parallel prefix

Given an array $A[0], \ldots, A[N-1]$, the parallel prefix problem is to compute in parallel all of the initial sums $\sum_{j=0}^{i} A[j]$ for $0 \leq i \leq N-1$. This is one of the most basic problems in parallel computation. Shared memory machines and machines with $\Theta(\log P)$ diameter can solve the problem in $\Theta(N/P + \log P)$ time. Typically, a diameter $\Theta(d)$ machine can solve the problem in $\Theta(N/P + d)$ time. A mesh, for example, requires $\Theta(N/P + \sqrt{P})$ time.

Consider the simpler problem of merely finding the sum of just P numbers. We use the standard tree algorithm:

for $j = 0, \ldots, \lg P - 1$ **do**
  for all $i = 2^j - 1, ..., N \bmod 2^j$ **do in parallel**
    $A[i] := A[i - 2^j] + A[i]$

At step j, the machine sends just one item within each submachine of size 2^{j+1}, so the throughput cost of the step is $\Theta(\beta(2^{j+1})/2^{j+1})$, which equals $\Theta(1)$, and the distance cost is $\Theta(\delta(2^{j+1}))$. So the total cost is

$$\Theta(L[\delta(P)]\delta(P)) + \sum_{j=0}^{\lg P - 1} O(1) = \Theta(L[\delta(P)]\delta(P)).$$

Now consider, finding the sum of $N \geq P$ numbers. First, each processor locally finds the sum of N/P numbers in time $\Theta(N/P)$, reducing the problem to finding the sum of P numbers. Thus, the time to sum $N \geq P$ numbers is

$$\Theta(N/P + L[\delta(P)]\delta(P)).$$

Parallel prefix can be computed within the same time bounds, by computing the partial sums while traversing up the tree and then sending the initial sums down the tree and appropriately adding them to partial sums.

Theorem 1 *The parallel prefix problem can be solved by a P-processor Y-PRAM[β, δ] in time*

$$\Theta\left(N/P + L[\delta(P)]\delta(P)\right).$$

We now substitute typical values for $\beta(P)$ and $\delta(P)$:

SHARED MEMORY MACHINE. Setting the parameters to match a shared memory machine gives, i.e., $\beta(P) = \delta(P) = \Theta(1)$:

$$\Theta((N/P + \log(P)).$$

HYPERCUBE. Setting the parameters to match a hypercube (or butterfly or shuffle-exchange), i.e., $\beta(P) = \delta(P) = \log P$, gives:

$$\Theta(N/P + \log^2 P).$$

MESH. Setting the parameters to match a mesh, i.e., $\beta(P) = \delta(P) = \sqrt{(P)}$ gives:

$$\Theta(N/\sqrt{P}).$$

TREE. Setting the parameters to match a tree, i.e., $\beta(P) = \Theta(P)$ and $\delta(P) = \Theta(\log P)$, gives:

$$\Theta(N/P + \log^2 P).$$

Notice that our time bounds are exact for the shared memory machine and the mesh, but off by a factor of $\log P$ in the second term for the hypercube and tree. This is because our algorithm assumes that the time to send a data item within a submachine of size P is $\Theta(\log P)$, whereas the hypercube is able to send items between machines whose sizes differ by a factor of two in constant time. In other words, some logarthmic distance machines can implement ascend-descend algorithms [20] especially efficiently.

4 Merge sort

We implement here a fairly simple and straightforward sorting algorithm. The algorithm locally sorts groups of N/P items, forming P sorted lists that need to be merged into a single sorted list. It pairwise merges the lists until only a single sorted list remains. At step j, $1 \leq j \leq \log P$, the algorithm merges two sorted lists each of size $(N/P)2^{j-1}$ into a single sorted list of size $(N/P)2^j$, using all the processors of a submachine of size 2^j.

The merging can be accomplished in a variety of ways. One method is to have one processor use binary search to split the merging problem into two independent merging problems of equal size: To merge the lists $a_1 \le a_2 \le \cdots \le a_m$ and $b_1 \le b_2 \le \cdots \le b_n$, the algorithms finds i, j so that, $i+j = (m+n)/2$, $b_1, \ldots, b_{j-1} \le a_i$ and $b_{j+1}, \ldots, b_n \ge a_i$, and similarly $a_1, \ldots, a_{i-1} \le b_j$ and $a_{i+1}, \ldots, a_m \ge b_i$. The processors split into two groups of equal size and work recursively on each half until there is only one processor per merging problem. Then, the sublists can be merged sequentially.

At step j, the time to find the first splitting item is $\Theta(\delta(2^j)\log((N/P)2^j))$, and (by the previous algorithm) the time to broadcast it is $\Theta(\sum_{k=0}^{j-1}\delta(2^k))$. Notice that when we find the second two splitting items the total list size has been halved and the submachine within which a broadcast needs to be done is also halved. After each processor determines the two sublists that it needs to merge, the processor must access all of its N/P items. This process is global within the submachine of size 2^{j+1}, so it takes time $(N/P)\beta(2^j) + \delta(2^j)$. Thus, the total time for step j is

$$\Theta\left(\sum_{h=0}^{j-1}\left(\delta(2^{j-h})\log((N/P)2^{j-h}) + \sum_{k=0}^{j-h-1}\delta(2^k)\right) + (N/P)\beta(2^j) + \delta(2^j)\right).$$

Summing over all $\log P$ steps and including the initial sorting time gives

$$\Theta\left(\sum_{j=1}^{\log P}\left(\sum_{h=0}^{j-1}\left(\delta(2^{j-h})\log((N/P)2^{j-h}) + \sum_{k=0}^{j-h-1}\delta(2^k)\right) + (N/P)\beta(2^j) + \delta(2^j)\right)\right)$$
$$+\Theta\left((N/P)\log(N/P)\right).$$

This sums to

$$\Theta\left(L^3[\delta(P)]\delta(P) + L^2[\delta(P)]\delta(P)\log N + L[\beta(P)]\beta(P)N/P + (N/P)\log(N/P)\right),$$

for the parameter ranges of main interest here; namely, $N \ge 2P$ and $\beta(P)$ and $\delta(P)$ constant, polylogarithmic, or polynomial in P. This proves,

Theorem 2 *The sorting problem of size N can be solved by a P processor Y-PRAM$[\beta, \delta]$ in time*

$$\Theta\left(L^3[\delta(P)]\delta(P) + L^2[\delta(P)]\delta(P)\log N + L[\beta(P)]\beta(P)N/P + (N/P)\log(N/P)\right),$$

where $N \ge 2P$ and $\beta(P)$ and $\delta(P)$ are constant, polylogarithmic, or polynomial in P.

For comparison, we consider a few models.

SHARED MEMORY MACHINE. For parameter values matching a shared memory machine, $\beta(P) = \delta(P) = \Theta(1)$, the above result simplifies to

$$\Theta(\frac{N \log N}{P} + \log^2 P \log N).$$

This is not quite as good as the best shared memory sorting results [4], but is still efficient for $N = \Omega(P^{\log^2 P})$.

HYPERCUBE. For parameter values matching a hypercube (or butterfly or shuffle-exchange) machine, $\beta(P) = \delta(P) = \Theta(\log P)$, the above result simplifies to

$$\Theta\left(\frac{N \log N}{P} + \frac{N \log^2 P}{P} + \log^3 P \log N\right).$$

This is efficient for $N = P^{\Omega(\log P)}$, which is not as good as the best hypercube sorting results [6, 7].

It seems that no parallel merge sort can obtain extremely good performance on our model for parameters matching the hypercube: There must be $\Theta(\log P)$ merge steps after locally sorting lists of size N/P. Each merge step will permute a total of N items, and each permutation will require an overhead of $\Theta(\beta(P))$ (on average). Thus, a term of the form $\Theta(L[\beta(P)]\beta(P)N/P)$ seems inherent.

MESH. For parameter values matching a mesh machine, $\beta(P) = \delta(P) = \Theta(\sqrt{P})$, the above result simplifies to

$$\Theta(\frac{N \log N}{P} + \frac{N}{\sqrt{P}}).$$

This is optimal for a mesh: the first term accounts for the $N \log N$ comparisons that must be performed in the worst case, and the second term accounts for the data movement that must be performed. The algorithm is efficient for

$$N \geq 2^{\Omega(\sqrt{P})}.$$

TREE. For parameter values matching a tree, $\beta(P) = \Theta(P)$ and $\delta(P) = \Theta(\log P)$, the above result simplifies to

$$\Theta(\frac{N \log N}{P} + N).$$

This algorithm is also optimal for the same reasons as the mesh algorithm is optimal. Notice that it is not quite as simple as the the algorithm that sorts lists of size N/P at the leaves of the tree and pairwise merges the lists up the tree using only one processor per merge. However, our algorithm was developed in a more general context.

We could obtain slightly better results by implementing Cole's merge sorting algorithm [4] which possesses much inherent locality, but the algorithm is much more complicated.

5 Column sort

We present an alternative parallel sorting algorithm that captures the performance of the best known algorithm for the hypercube. The general idea, which is based on Leighton's column sort [17], is due to Han [10].

Column sort works by partitioning the N items to be sorted into a two dimensional matrix. The columns are sorted and the matrix is permuted (by an appropriate permutation). These two steps are executed a total of eight times, at which point the entire list is sorted. The only requirement is that the number of rows must be quadratically larger than the number of columns.

One could use this as a parallel sorting algorithm by partitioning the list to be sorted into an $N^{2/3} \times N^{1/3}$ matrix. If the number of columns ($N^{1/3}$) is at least as large as the number of processors (P), then the algorithm can be implemented directly, by assigning an equal number of columns to each processor and sorting columns sequentially. Otherwise, we assign $P/N^{1/3}$ processors per column, and execute the algorithm recursively.

Thus, the sorting time $T_P(N)$ satisfies the following recurrence:

$$T_P(N) \leq \begin{cases} 8T_{P/N^{1/3}}(N^{2/3}) + \Theta(\frac{N}{P}(\beta(P) + \delta(P))) & \text{if } P > 1 \\ O((N/P)\log(N/P)) & \text{if } P = 1. \end{cases}$$

The depth of the recursion is $\log_{3/2}((\log N)/\log(N/P))$. So, the number of sequential sorts of N/P elements is $(\log((\log N)/\log(N/P)))^{\Theta(1)}$, which implies that to total time spent sorting sequentially is

$$(\log((\log N)/\log(N/P)))^{\Theta(1)}\Theta((N/P)\log(N/P)).$$

For ($\beta(P) = \delta(P) = \Theta(\log P)$) the total time spent permuting is

$$(\log((\log N)/\log(N/P)))^{\Theta(1)}N/P.$$

Thus, for ($\beta(P) = \delta(P) = \Theta(\log P)$) the total time for sorting is

$$(\log((\log N)/\log(N/P)))^{\Theta(1)}\Theta((N/P)\log(N/P)).$$

If in addition, $N \geq P^{1+\epsilon}$, for some constant $\epsilon > 0$, the total time for sorting is

$$\Theta((N/P)\log(N)).$$

This matches the best known hypercube sorting algorithm [6].

We have proved,

Theorem 3 *Column Sort can be implemented in the P processor Y-PRAM$[\beta, \delta]$ to sort N elements in time*

$$\Theta((N/P)\log(N)),$$

for $\beta(P) = \delta(P) = \Theta(\log P)$, and $N = \Omega(P^{1+\epsilon})$, $\epsilon > 0$.

6 Radix sort

We implement the radix sort algorithm of [14, 22, 25] on our model. Assume we wish to sort N numbers in the range $0, \ldots, M-1$ using radix R.

One pass of radix sort does the following: Each processor locally contains N/P items. Create a $P \times R$ array. Each processor locally counts how many of its N/P items belong in each of the R buckets. This takes time $\Theta(N/P)$. Performing parallel prefix on the $P \times R$ array *by columns*, and a parallel prefix of the sequence of the total for each column, determines for each item its global location in the sorted array. By the subsection on parallel prefix, this takes time $\Theta(R + L[\delta(P)]\delta(P))$. Move the items to their determined destinations, which sorts the items by radix R. This takes time $\Theta((N/P)\beta(P) + \delta(P))$. Thus, the total time for one pass is

$$\Theta((N/P)\beta(P) + R + L[\delta(P)]\delta(P)).$$

To fully implement radix sort, we need $(\log M)/(\log R)$ passes. Thus, the total time for all of the passes is

$$\Theta(\frac{\log M}{\log R}((N/P)\beta(P) + R + L[\delta(P)]\delta(P)).$$

which is equal to

$$\Theta(\frac{\log M}{\log((N/P)\beta(P))}((N/P)\beta(P) + L[\delta(P)]\delta(P))).$$

for $R = (N/P)\beta(P)$. This proves,

Theorem 4 *A P-processor Y-PRAM$[\beta, \delta]$ can sort N integers in the range $0, \ldots, M-1$ in time*

$$\Theta(\frac{\log M}{\log((N/P)\beta(P))}((N/P)\beta(P) + L[\delta(P)]\delta(P))).$$

We can now substitute values for $\beta(P)$ and $\delta(P)$:

SHARED MEMORY MACHINE. Setting the parameters to match a shared memory machine gives:

$$\Theta(\frac{(\log M)}{\log(N/P)}(N/P + \log(P)).$$

This is the same result that one gets directly for the shared memory machine [14, 22, 25]

HYPERCUBE. Setting the parameters to match a hypercube gives:

$$\Theta(\frac{\log M \log P}{\log((N/P)\log(P))}(N/P + \log(P)).$$

MESH. Setting the parameters to match a mesh gives:

$$\Theta(\frac{\log M}{\log(N/\sqrt{P})}(N/\sqrt{P}).$$

For M at most polynomially larger than N, i.e. $M \leq O(N^k)$ for some constant $k > 0$, this simplifies to $\Theta(N/\sqrt{P})$, which is optimal.

TREE. Setting the parameters to match a tree gives:

$$\Theta(\frac{\log M}{\log N}N).$$

For M at most polynomially larger than N, this simplifies to $\Theta(N)$ which is optimal for the tree, although not especially interesting.

The region of interest will typically be M at most polynomially larger than N, and N at least polynomially larger than P, i.e. $N \geq P^{1+\epsilon}$ for some constant $\epsilon > 0$. In this case, our running time simplifies to

$$\Theta((N/P)\beta(P) + L[\delta(P)]\delta(P)).$$

Since routing takes $\Theta((N/P)\beta(P) + \delta(P))$ (for $N >> P$) on our model, this result shows that radix sort is essentially equivalent to routing (in this region).

7 List ranking

A problem that is fundamental for graph algorithms is list ranking: One is given a linked list of size N and wishes to determine the distance of each node from the head of the list.

We can implement Wyllie's recursive doubling algorithm [26] for $N = P$. Assume each node points to its successor via a *next* field and starts with *count* = *1*. The *next* pointer for the head field is *nil*. Each processor is assigned a node and synchronously executes the following code on its own node:

while $next \neq nil$ **do**

$count := count + count(next)$

$next := next + next(next)$

This algorithm takes $\log P$ iterations, and each iteration takes $\Theta(\beta(P) + \delta(P))$ time, so the total time is $\Theta((\beta(P) + \delta(P)) \log P)$. One could generalize this to N nodes by assigning N/P processors per node and iterating through the loop $\log N$ times. The time will then be $\Theta((\beta(P)N/P + \delta(P)) \log N)$. Notice that this is inefficient because of the $\Theta(\beta(P))$ cost of accessing all of the nodes at each iteration, and because of the the fact that there are $\log N$ iterations.

The basic idea of all efficient parallel list ranking algorithms is to compact in parallel many pairs adjacent nodes into single nodes. The difficulty is to avoid compacting node a with node b, while at the same time compacting node b with node c. After the compaction phases are completed the steps of the algorithm are unwound to "broadcast" the node ranks.

We implement the algorithm of Kruskal, Rudolph, and Snir [13] There are $O(\log(N/P))$ phases. Each phase reduces the number of nodes by at least a constant fraction until there are less than $2P$ nodes. At that point, the recursive doubling algorithm of Wyllie [26] is used finish up the list ranking.

The algorithm partitions the nodes of the list into a $P \times N/P$ array. The processors synchronously visit the columns of the array, i.e. at step j the ith processor visits the jth element of the ith row. Each processor compacts its visited node with the node's successor as long as the successor is not in the same column. This guarantees that compaction conflicts - described above - are avoided.

In order to compact within a column, the processors partition themselves into N/P groups, each of size P^2/N, one group for each column. The algorithm is then applied recursively to each column, until each column is assigned only one processor, at which point no conflicts can occur. After finishing the recursion, at most $2/3N + O(1)$ nodes remain, since if a node has not been compacted then both of its neighbors must have been compacted. The remaining nodes are packed into an array of size $2/3N + O(1)$ (or smaller). The packing is accomplished by doing a parallel prefix on the nodes with 1

assigned to *live* (not yet compacted) nodes and 0 assigned to *dead* (already compacted) nodes, then updating the pointer values for live nodes, and finally moving the live nodes (to the location indicated by the parallel prefix value). This completes the first phase. Now the whole algorithm is applied recursively to the resulting smaller list, which has $O(N/d)$ elements, $d = 2/3$.

To analyze the algorithm we let $H_P(N)$ be the time for P processors to compact N nodes. Then, by our recursive construction,

$$H_P(N) \leq \begin{cases} H_{P^2/N}(\frac{N}{P}) + \Theta(\frac{N}{P}(\beta(P) + \delta(P))) & \text{if } P > 1 \\ \Theta(N/P) & \text{if } P = 1. \end{cases}$$

Let $U_P(N)$ be the time for P processors to execute the entire list ranking algorithm on N nodes. Then, by our recursive construction,

$$U_P(N) \leq U_P(N/d) + H_P(N) + N/P + L[\beta(P)]\beta(P) + \frac{N}{P}\beta(P) + \delta(P)$$

where $N > 2P$ and $d = 3/2$. Note that computing the locations for the packing is done with a parallel prefix which takes $N/P + L[\beta(P)]\beta(P)$, and moving the elements to the computed locations takes $\frac{N}{P}\beta(P) + \delta(P)$.

Then the total time for our list ranking algorithm $T_P(N)$ is $U_P(N)$ plus the time list rank on a list with at most $2P$ nodes, $T_P(N) = U_P(N) + \Theta((\beta(P) + \delta(P)) \log P)$. We have thus proved,

Theorem 5 *The list ranking problem of size N can be solved by a P-processors Y-PRAM[β, δ] in time $T_P(N)$ such that*

$$T_P(N) \leq \begin{cases} T_P(N/d) + H_P(N) + \Theta(N/P + L[\beta(P)]\beta(P)) + \frac{N}{P}\beta(P) + \delta(P) & \text{if } N > 2P \\ \Theta((\beta(P)N/P + \delta(P)) \log N) & \text{if } N = 2P, \end{cases}$$

where $d = 2/3$, and

$$H_P(N) \leq \begin{cases} H_{P^2/N}(\frac{N}{P}) + \frac{N}{P}(\beta(P) + \delta(P)) & \text{if } P > 1 \\ O(N/P) & \text{if } P = 1. \end{cases}$$

We will not solve this recurrence in general, but rather solve some special cases of interest:

SHARED MEMORY MACHINE. Setting the parameters to match a shared memory machine gives:

$$T_P(N) = \Theta(\frac{N}{P}\frac{\log N}{\log(2N/P)}).$$

This is optimal for $N \geq P^{1+\epsilon}$ for any constant $\epsilon \geq 0$. The best parallel list ranking algorithms for a P processors EREW PRAM take $O(N/P + \log P)$ which is optimal for $N = \Omega(P \log P)$ [3, 5].

HYPERCUBE. Setting the parameters to match a hypercube gives:

$$T_P(N) = \Theta(\frac{N}{P}\frac{\log N}{\log(2N/P)}\log P) \qquad (N \geq 2P).$$

Again, this is optimal for $N \geq P^{1+\epsilon}$ for any constant $\epsilon \geq 0$. This matches the performance of the fastest known list ranking algorithm for the (strict) Hypercube, whose complexity is $O(\frac{N}{P}\frac{\log^2 N}{\log(N/P)})$ and becomes optimal for $N \geq P^{1+\epsilon}$ [6].

MESH. Setting the parameters to match a mesh gives:

$$T_P(N) = \Theta(\frac{N}{\sqrt{P}} + \sqrt{P}\log P).$$

This is optimal for $N > \sqrt{P}\log P$. For the case $N = P$, the parallel list ranking problem can be solved on a $\sqrt{N} \times \sqrt{N}$ mesh in $O(\sqrt{N})$ time, which is optimal [8].

TREE. setting the parameters to match a tree gives:

$$T_P(N) = \Theta(N + P\log P),$$

which is optimal for $N > P \log P$.

8 Conclusion

We have proposed a model of parallel computation, the Y-PRAM, that takes into account latency and bandwidth. It allows parallel algorithms to be designed independently of the parallel model. We presented *Y-PRAM* algorithms for parallel prefix, sorting, sorting numbers from a bounded range, and list ranking. It seemed to be easy to write programs for this model.

The *Y-PRAM* model seems to provide a reasonably accurate prediction of actual performance. Many *Y-PRAM* time bounds exactly match the best parallel bounds, when the parameters are set to match a particular machine. Sometimes the bounds were off by a $\log P$ factor in the second term, as in the parallel prefix algorithm for the hypercube. This discrepancy occurred because our model assumes that the time to send a data item within a submachine of size P is $\Theta(\log P)$, whereas the hypercube is able to send items between machines whose sizes differ by a factor of two in constant time. In other words, some

logarthmic distance machines can implement ascend-descend algorithms [20] especially efficiently. We believe that it may unnecessarily complicate the model to try to account for this low level effect.

There are many problems left open by this work, some of which we plan to tackle in the future. For example, it would be nice to design algorithms for more problems. The *Y-PRAM* is only a first approximation; more experience designing algorithms will indicate how the model should be generalized or restricted, and also what other parameters, if any, should be included. It can be refined in a variety of ways. Many machines communicate by "randomly" sending message around the machine. In that case, a small amount of more global background traffic would not seriously degrade performance, as measured by bandwidth. Such a model would probably provide a more realistic reflection many machines including CM* and Cedar, However, it seems to be more difficult to define cleanly, and none of our current algorithms take advantage of this extra traffic.

Another variant of our model, which would also be a more realistic reflection of real machines, would be to restrict the submachines so that they can communicate only via routing, rather than allowing the full power of a shared memory. In this case, m would be maximum number of messages that any one processor sends or receives. This refinement would complicate algorithm design for questionable improvement in estimating performance. Our Y-PRAM model defines what might be called an EREW Y-PRAM. Any other PRAM model could be used for the communication within submachines.

References

[1] A. Aggarwal, A. Chandra, and M. Snir. Communication complexity of prams. Technical Report RC 14998(64644), IBM Tech. Report, 1989.

[2] A. Aggarwal, A. Chandra, and M. Snir. On communication latency of prams. Technical Report RC 14973(66882), IBM Tech. Report, 1989.

[3] R. J. Anderson and G. L. Miller. Deterministic parallel list ranking. In *Proc. 3rd AWOC*, pages 81–90, 1988.

[4] R. Cole. Parallel merge sort. In *Proc. 27th Annual IEEE Symp. on Foundations of Computer Science*, pages 511–516, 1986.

[5] R. Cole and U. Vishkin. Approximate parallel scheduling, part i: the basic technique with applications to optimal parallel list ranking in logarithmic time. *SIAM J. Comput*, 17:128–142, 1988.

[6] R. Cypher. *Efficient communication in massively parallel computers.* PhD thesis, University of Washington, 1989. Department of Computer Scienceeig.

[7] R. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *Proc. 15th Annual ACM Symp. on Theory of Computing*, pages 193–203, 1990.

[8] A. M. Gibbons and Y.N. Srikant. A class of problems efficiently solvable on mesh-connected computers including dynamic expression evaluation. *Information Processing Letters*, 32:305–311, 1989.

[9] A. Gottlieb and C. P. Kruskal. Complexity results for permuting data and other computations on parallel processors. *JACM*, 31:193–209, 1984.

[10] Y. Han. Parallel algorithms for computing linked list prefix. *J. of Parallel and Distributed Computing*, 6:537–357, 1989.

[11] R. Karp and V. Ramachandran. A survey of parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*, 1988. Ed. J. van Leeuwen, North Holland, to appear.

[12] C. P. Kruskal, T. Madej, and L. Rudolph. Parallel prefix on fully connected direct connection machine. In *Proc. International Conference on Parallel Processing*, pages 278–283, 1986.

[13] C. P. Kruskal, L. Rudolph, and M. Snir. The power of parallel prefix. *IEEE Transactions on Computers*, pages 965–968, 1985.

[14] C. P. Kruskal, L. Rudolph, and M. Snir. Efficient parallel algorithms for graph problems. In *Proceedings International Conference on Parallel Processing*, pages 869–876, 1986.

[15] C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. In *Proc. 15th Annual ICALP*. Springer Verlag Lecture Notes in Computer Science, Vol. 317, pp. 333-346, July 1988. (Theoretical Computer Science, to appear 1989).

[16] D. J. Kuck, E. S. Davidson, D. H. Lawrie, and A. H. Sameh. Parallel supercomputing today and the cedar approach. *Science*, 231:967–974, 1986.

[17] F. T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, C-34:344–354, 1985.

[18] L. Meertens. Recurrent ultracomputers are not log n-fast. Technical Report 2, New York University, 1979. Ultracomputer.

[19] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of prams by parallel machines with restricted granularity of parallel of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[20] F. P. Preparata and J. E. Vuillemin. The cube-connected cycles: A versatile network for parallel computation. *CACM*, 24:300–309, 1981.

[21] T. Leighton B. Maggs S. Rao. Universal packet routing algorithms. In *Proc. 29th Annual IEEE Symp. on Foundations of Computer Science*, pages 256–271, 1988.

[22] J. H. Reif. An optimal parallel algorithm for integer sorting. In *Proc. 26th Annual Symp. on Foundations of Computer Science*, pages 496–504, 1985.

[23] R. J. Swan, S. H. Fuller, and D. P. Siewiorek. Cm* – a modular, multi-processor. In *Proc. AFIPS Conf.*, volume 46, pages 637–644, 1977.

[24] L. G. Valiant. General purpose parallel architectures. In *A handbook of Theoretical Computer Science*. MIT Press, 1990. J. van Leeuwen (ed.).

[25] R. A. Wagner and Y. Han. Parallel algorithms for bucket sorting and the data dependent prefix problem. In *Proc. International Conference on Parallel Processing*, pages 924–930, 1986.

[26] J. C. Wyllie. *The complexity of parallel computation.* PhD thesis, Cornell University, 1979.

Neural Computing and the GALATEA Project

Philip Treleaven

Department of Computer Science
University College London

ABSTRACT

This paper reviews the fundamentals of neural computing which includes: neural network models, neural network programming environments, and neurocomputer; specialised hardware for neural networks.

It then describes the ESPRIT II GALATEA project, and its predecessor PYGMALION, which provide the focus of neural computing research in the European Community. PYGMALION has developed a general programming environment for neural networks, and the goal of GALATEA is to build upon this environment, to produce a comprehensive neurocomputing system. This neurocomputing system will comprise: a sophisticated programming environment capable of mapping a network on to a range of conventional computers, including parallel machines; a novel general-purpose neurocomputer; and an integral silicon compiler for translating a network into VLSI *chips*.

1. Artificial Neural Networks

Neural networks[1] are a novel form of computation that attempts to mimic the functionality of the human brain, in order to solve demanding pattern processing problems. The term *neural computing* spans artificial neural networks, neural programming environments and neurocomputers, applying them to a broad class of *pattern processing* applications. These applications include: control, image processing, speech processing, inexact knowledge processing, natural-language processing, sensor processing, planning, forecasting and optimisation.

The key to neural computing is understanding the ways in which the brain uses neuronal systems for pattern processing[2]. The biological neuron basically consists of a cell body called a *soma* (Figure 1a), branching complex extensions called *dendrites*, and an *axon*, the output channel of the cell, carrying an electric signal to other cells. The axon connects to the dendrites of other cells through specialized contacts called *synapses* that can change, positively or negatively, the axon potential. The traditional view is that the neuron performs a simple threshold function - "weighted" input signals are summed; if the result exceeds a certain threshold, a signal is send out from the neuron.

The artificial neuron is made up of three sections (see Figure 1b), corresponding to the simplified model of the biological neuron: the weighted input connections, the summation function and a threshold function, that generates the unit output (usually *off* or *on*). The artificial neuron operates as a simple threshold device, depending on the state *S* of its input elements and the connection weights (*W*).

In an artificial neural network, neurons are generally configured in regular and highly interconnected topologies. Programming of a neural network, firstly involves specifying the mathematical function of the artificial neurons and their connections, and secondly involves the *training* of the network to recognise a set of patterns.

Typically the artificial neurons (called processing elements - PEs) are organised into layers with each PE in one layer having a connection to each PE in the next layer, as illustrated by Figure 2. Associated with each connection is a *weight* and with each PE is a *state* (usually *off* or *on*). Together these weights and states represent the distributed *data* of the network. The weights of a network together represent an energy surface, and their actual values determine the set of patterns recognisible by the network. During pattern recognition, each PE operates as a simple threshold device. A PE sums all the weighted inputs (multiplying the connection weight by the state of the previous layer PE) and then

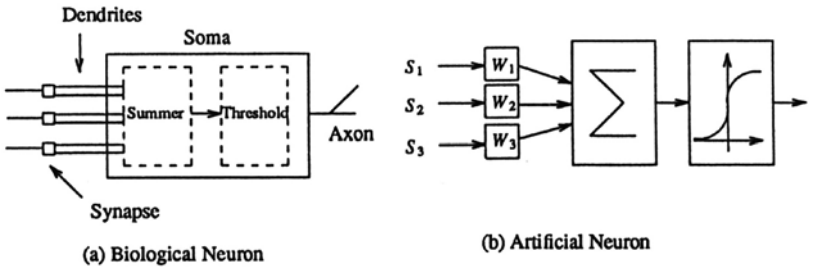


Figure 1: Simplified Models of Neurons

applies a threshold function, such as a sigmoid function.

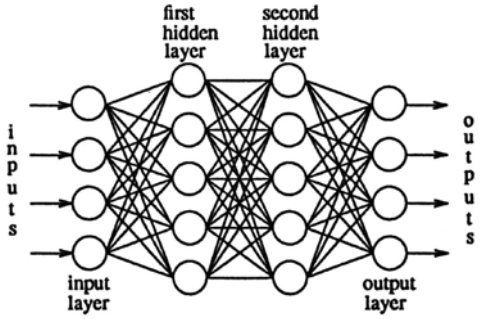


Figure 2: Multilayer Neural Network

A widely used neural network model is ***Back Propagation***[3]. A Back Propagation model is *trained* to recognise patterns by presenting example training pairs of input-target pattern (e.g. a hand-written *A* with a perfect *A*). The input pattern is presented and passes through the network to produce outputs. This output pattern is then compared with the "ideal" target pattern, and an *error* is propagated back through the network. The propagated error is used to adjust the weights of the connections. This training process is then repeated with a new training pair, and a new (hopefully smaller) error is propagated backwards. This training process is repeated with example pairs of patterns until the error is negligible, at which time the network is trained.

2. Neural Programming Environments

Interest in neural networks has stimulated the availability sophisticated software Environments for programming neural networks[3]. Environments range from commercial products from both established and Startup companies, to public-domain software, available free from university research groups.

These diverse neural programming environments share many common features. A typical neural network programming environment, as illustrated by Figure 3, might comprise:

- a ***graphic interface***, with menus and a command language: for configuring a neural network, then controlling and monitoring its execution;
- an ***algorithm library*** of common, parameterised neural network algorithms, such as Back Propagation, Hopfield, Boltzmann etc.;
- a ***high-level language***, often object-oriented, for programming or customising an algorithm or application;

- a ***network specification language***, a low-level, machine-independent, language (often based on C) defining the neural network simulation;
- ***hardware configuration*** information for defining the organisation of the target machine to run the network simulation; and
- ***translators*** for mapping the network specification language to various target machines.

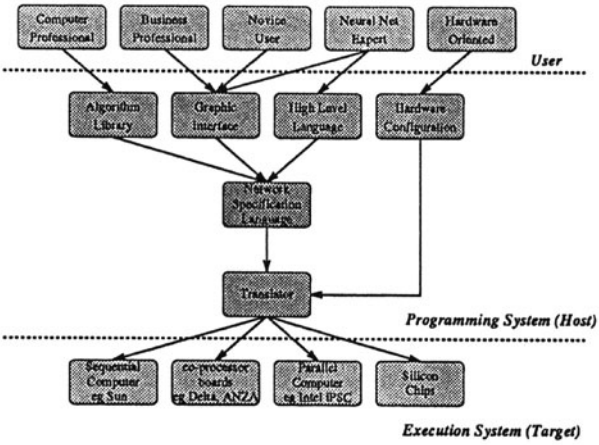


Figure 3: Typical Neural Programming Environment

Neural programming environments may be classified into three major groups: application-oriented, algorithm-oriented and general programming systems.

Applications-oriented systems
are designed for specific market domains, such as *finance* or *transportation*. These application domains form a natural subdivision. A good example is Nestor's Decision Learning System[4].

Algorithm-oriented systems
support specific neural network models. This class has two major subclasses which are: *algorithm-specific* supporting a single model, such as Back Propagation of errors, and *algorithm libraries* supporting many models coded in a common language like "C". These algorithm-oriented systems are often supplied as source code routes and therefore are easily incorporated in user applications. A popular algorithm library is OWL[5].

Programming systems
the third class, provide general "tool-kits" comprising many algorithms and programming tools, that can be used for a wide range of algorithms and applications. Programming systems can be further sub-divided into: *educational* systems for the novice user to obtain a hands-on introduction and normally support only small networks based on common algorithms; *general-purpose* systems provide comprehensive tool-kits for programming any algorithm or application; *open* systems where the user can modify any part of the system; and *hardware-oriented* systems typically supporting particular parallel computers.

As an illustration of our Taxonomy, Figure 4 lists some of the well known environments in each class.

The level of sophistication and usage of a given environment depends on the target group of users. Business professionals, interested say in financial forecasting, have little concern for the details of neural networks. For them, an applications-oriented system gives emphasis to the graphic interface and a specific, possibly proprietary, algorithm. The user merely configures a parameterised network, applies the data and monitors the network's execution.

Class	Category	Organisation	System
Application-Oriented	Financial	Nestor	Decision Learning System (DLS)
	Transportation	BehavHeuristics	Airline Marketing Tactician (AMT)
		Excalibur	Savvy
Algorithm-Oriented	Algorithm-Specific	Cal. Scient. Softw.	BrainMaker
	Algorithm Libraries	Olmstead & Watkins	Owl
		Mimetics	Galatea C-library
Programming Systems	Educational Systems	UCSD	PDP
		NeuralWare	Explorer
	General-Purpose Systems	SAIC	ANSpec
		HNC	Anza/Axon
		NeuralWare	NeuralWorks Professional II
	Open Systems	Lucid	Plexi
		UCL	Pygmalion
	Hardware-Oriented Systems	Oregon Grad. Institute	Anne

Figure 4: Neural Network Programming Environments

For the novice user wishing to learn about neural networks there are many good educational systems that allow small parameterised networks to be configured, and their execution monitored. These systems typically have an easy to use graphic interface, a library of a few common networks, such as Hopfield, Back Propagation etc. and simple demonstration applications. Essentially these environments are "skeleton" general-purpose programming systems.

For the computing professional wishing to incorporate a common neural network algorithm in an application, there are source code algorithm libraries available in C and LISP. These libraries, provide an optional rudimentary graphic interface, and any additional programming is done in C or LISP. This is a straight-forward, and popular, way of incorporating neural networks into conventional applications.

For neural network experts who need to program their own algorithms and applications there are many general-purpose programming systems. Besides providing a comprehensive graphic interface and algorithm library, they also usually provide a high level language with specialised data structures, classes and functions. These languages are usually object-oriented, possibly based on C++. Many of these general-purpose systems are also "open", allowing the programmer to modify any part of the environment.

Finally, an increasing number of users are interested in mapping neural networks on to specific hardware, such as parallel computers like Transputers, or even into silicon. Hardware-oriented programming systems emphasis the hardware configuration component, specifying the structure of the target hardware, and the translator which uses this configuration information to map the neural specification language on to the hardware. Certain hardware-oriented systems are little more than parameterised code for running a specific neural network. Others are translators for mapping the network specification language to a parallel machine or to a silicon chip.

3. Neurocomputers

The computational demands of neural networks have also stimulated the development of specialised hardware, referred to as *neurocomputers*, to speed up execution. There are two basic classes of neurocomputer:

- **General-Purpose Neurocomputers** - generalised, programmable, neural computers for emulating a range of neural network models, thus providing a framework for executing neural models in much the same way as traditional computers.
- **Special-Purpose Neurocomputers** - that are dedicated hardware implementations of a specific neural network model.

The essential difference between them is whether the neurocomputer is programmable, hence general-purpose and capable of supporting a range of neural network models, or is special-purpose, implementing a dedicated neural network.

General-purpose neurocomputers subdivide into *commercial co-processors* boards and *parallel processor arrays*[6]. Commercial co-processors are typically floating-point or signal processing accelerator boards, usually supplied with a large memory (e.g. 4M byte), that plug into the backplane of an IBM PC, or interface to a SUN Workstation. Parallel processor arrays are cellular arrays, composed of a large number of primitive processing units, connected in a regular and usually restricted topology. Their general architecture is shown in Figure 5. The structure resembles a parallel array processor, comprising identical processors connected through a parallel broadcast bus, where each physical unit executes a section of the "virtual" network. To program the neurocomputer, the virtual PEs are partitioned across the local memories of the physical processors. Updating a virtual PE implies broadcasting the update through the bus. Units that need access to that information accept and store the update in their system state memory.

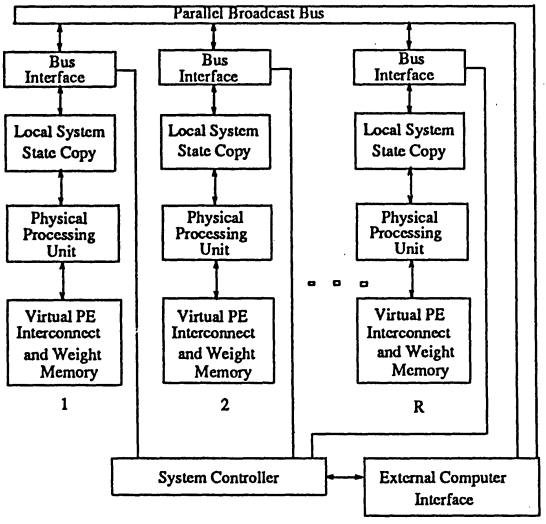


Figure 5: General-Purpose Virtual Neurocomputer Architecture
(from Hecht-Nielsen)[7]

These two categories of general-purpose systems differ basically in the number and complexity of the physical processing units employed. Parallel processor arrays aim to attain high performance and real parallelism by (mainly) increasing the number of the implemented processing units, while co-processors attempt to improve performance by strengthening the processing/storage capabilities of standard microprocessors.

Hardware accelerators, such as the HNC ANZA[8], the SAIC SIGMA-1[9] and, from Japan, the NEC Neuro-07[10], have allowed neural network experiments to be carried out over 100 times faster than the usual simulators of neural network models. They are programmable and can implement large networks of virtual processing elements with a limited number of hardware implemented processors. Processors are usually implemented by means of an industry-standard signal processing chip or microprocessors such as the MC68020 (and its MC68881 floating point co-processor), interconnected through a standard parallel broadcast bus, such as the VME bus. Physical processors and interconnections are multiplexed across a large number of virtual processing elements and virtual interconnections, so demanding large

memories to represent them. Performance comparisons between these products are based on capacity, meaning the maximum size of neural network, and speed, the time to process a neural network. Speed is usually expressed in network updates per second, for both the training phase and the recall phase. For instance, the ANZA Plus supports 1M PEs with 1M interconnections, and is capable of 1.5M connection updates per second during training and 6M updates during recall.

Several examples of very high performance special-purpose neurocomputers have been successfully built[6]. They are, typically, analog electronic implementations of neural networks and usually employ structures that resembles the simplified model of the neuron.

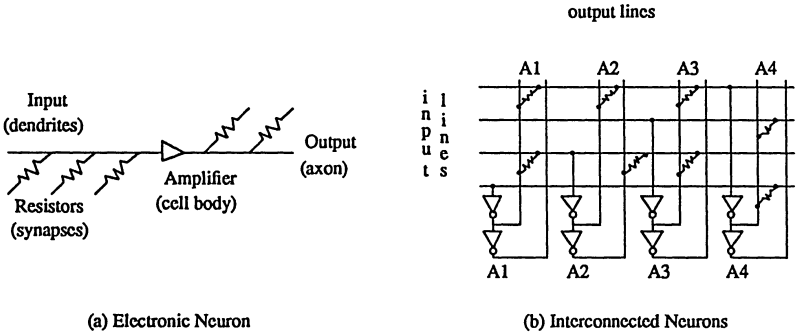
In a simple model of the neural function, the neuron state is given by:

$$s = T(\Sigma\ S.W - \theta)$$

The circuit of Figure 6a is a straightforward electronic implementation of the above equation[11]. Wires replace the input structure (dendrites) and the output (axon); the conductances (1/R) model the synaptic connections between neurons; and the amplifier models the cell body by executing the threshold function. Inputs appear as voltages to the summing wire. Using these electronic neurons, a neural network is implemented as a crossbar representation (Figure 6b). Outputs (vertical lines) are connected through resistors to the horizontal summing wires, which represent the weighted sum of the output signals of other PEs. Inverted input lines are provided when both positive and negative values are required.

For many neural network applications, however, it is necessary to change the synapses values[12]. To implement programmable connections, resistors can be replaced by a "synapse circuit", capable of providing different types of connections: excitatory, inhibitory and disabled.

In the design of special-purpose chips, the coupling network occupies most of the chip area. This is due to the difference between the silicon areas required by the PEs (containing many transistors) and by the wires (plus spacing).



(a) Electronic Neuron

(b) Interconnected Neurons

Figure 6: Special-Purpose Neurocomputer

Special-purpose neurocomputers are naturally subdivided by their implementation technology into: *silicon processors*, *optical processors* and *molecular processors*. The approach for special-purpose neurocomputer architectures is to directly implement a specific neural network model in hardware to produce a very high performance system. Any neural network model could theoretically be chosen, although currently the Kohonen or the Hopfield associative memory model[3] are typically favoured, because of their simplicity. Most silicon implementations use an analog electronic neuron model, as described above.

Further details of neural networks, neural programming environments and neurocomputers are given in the references[6].

4. The GALATEA Project

In the European Community, the focus of neural computing research is the ESPRIT II GALATEA Project, and its predecessor PYGMALION. PYGMALION and GALATEA broadly aim to promote the application of neural networks by European industry, and to develop European "standard" computational tools for programming and simulation of neural networks. The projects, as shown by Figure 7, bring together many of the leading neural computing research groups from European industry, research institutes and universities.

Partner	Laboratory	Role
Thomson-CSF	Division Outils Informatiques, Paris	*Prime Contractor* Manager integration workpackage OCR application
Philips	Lab. d'Electronique et de Physique Appliquee,Paris	Manager image processing application neurocomputer
Siemens AG	Central Research laboratories	Manager hardware workpackage neurocomputer programming environment
Mimetics SA		Manager silicon compilation Manager OCR application programming environment
UCL	University College London	Manager software workpackage programming environment silicon compilation
SGS-Thomson	Microelectronics Srl	silicon compilation
INPG	Institut National Polytechnique de Grenoble	silicon compilation
IS	Informatica Sistemi Spa	image processing application
CRAM	Consorsio per la Ricerca in Agricultra nel Mezzogiorno	image processing application
INESC	Instituto de Engenharia Sistemas e Computadores	algorithm library
CTI	Computer Technology Institute, Patras	parallel processing

Figure 7: Partners in the GALATEA Project

PYGMALION[13] has produced a rudimentary environment for programming neural networks. Its design philosophy is threefold. Firstly, to provide a general environment for neural networks with the same facilities (e.g. graphic monitor, algorithm library etc.) and functionality as commercial systems such as those of Hecht-Nielsen, NeuralWare and SAIC. Secondly, to provide a rudimentary "platform" - that can be easily extended and interfaced to other tools. For this reason the core of the environment is ***X-windows***, ***C*** and ***C++***; running on a colour workstation. Thirdly, to provide "portable" neural network applications, so that trained and partially trained networks can be easily moved from machine to machine. For this reason the (partially) trained neural network applications are specified in a subset of ***C***; essentially a ***C*** data structure.

The environment comprises 5 major parts:

- **Graphic Monitor**, the graphical software environment for controlling the execution and monitoring of a neural network application simulation.
- **Algorithm Library**, the parameterised library of common neural networks, written in the high level language ***N***.
- **High Level Language *N***, the object-oriented neural programming language based on ***C++***.
- **Intermediate Level Language *nC***, the low level machine independent network specification language, which is a subset of ***C***.
- **Compilers** to the target UNIX-based workstations and parallel Transputer-based machines.

PYGMALION embodies a hierarchical data structure encompassing: system, network, layer, cluster, plus neuron and synapse. Network information is encoded in the *C* data structure **system**, as shown in Figure 8.

```
typedef struct {
        int n_rules;                /* # of rules */
        rule_type *rules;           /* list of rules */
        int n_parameters;           /* # of user parameters */
        par_type *parameters;       /* list of user_parameter */
             (...)                  /* system variables (e.g. int clusters;) */
             (...)                  /* lower level elements (e.g. cluster_type *cluster;) */
} (...)_type;                       /* name of the level (e.g. layer) */
```

Figure 8: *nC* Data Structure for Network Information

The Graphic Monitor sits on top of the *nC* data structure and displays its contents, with a window corresponding to each level in the data structure. There are two types of window: **Top Window**, providing facilities for controlling the simulation and displaying status information plus the program text of the neural network; and **Level Windows**, providing control facilities, and displaying status information and a graphic representation, for each specific level. A window has three areas: commands, graphics/text, and parameters (i.e. status). A command area, comprises a series of button boxes (i.e. labels), each associated with a command or a "pull-down" menu. For example, at the top level, the boxes are create, start, pause, resume, input/output and quit. A graphic area presents a graphical view of the specific level and its pattern of intra-connectivity. Lastly, a parameter area displays status information in a textual form.

The PYGMALION programming environment is available free from University College London, and has been distributed to over 250 organisations.

The goal of GALATEA is to build upon this environment, to produce a "general-purpose" neurocomputing system. We visualise a general-purpose neurocomputing system as being a fully integrated environment of software and hardware components for the development and implementation of artificial neural networks. This neurocomputing system (see Figure 9) will comprise: a sophisticated programming environment capable of mapping a network on to a range of conventional computers, including parallel machines; a novel general-purpose neurocomputer; and an integral silicon compiler for translating a network into VLSI *chips*.

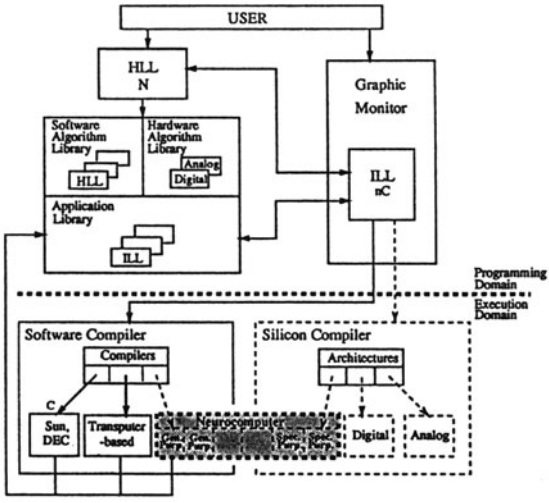


Figure 9: The GALATEA Neural Computing System

The programming environment will be similar to PYGMALION, and execute on a conventional workstation. Once programmed, a neural application can be compiled into binary (i.e. software), or silicon (i.e. hardware) for execution:

- ***workstation simulation*** - a binary representation could be executed on the workstation itself (e.g. SUN, DEC).
- ***parallel simulation*** - the binary could be down-loaded to a conventional parallel computer connected to the workstation (e.g. a Transputer accelerator).
- ***neurocomputer emulation*** - the binary could be down-loaded to general-purpose sub-systems used in a specialised neurocomputer connected to the workstation.
- ***silicon chip*** - a silicon representation in a special-purpose integrated circuit (digital or analog).
- ***special-purpose sub-system*** - special-purpose silicon chips mounted in a sub-system for use in the neurocomputer.

In addition, applications composed of multiple networks may combine the binary or silicon execution strategies specified above.

In conclusion, GALATEA aims to develop a fully integrated, general-purpose neurocomputing environment, covering all possible aspects of neural network application in real-world tasks and research. A key component of the environment is the neurocomputer, viewed as a heterogeneous system containing a number of general-purpose programmable sub-systems. It could also include one or more special-purpose, domain-specific sub-systems.

References

1. I. Aleksander and H. Morton, "Introduction to Neural Computing," in *North Oxford Press*, 1990.
2. G. Shepherd, "Synaptic Organisation of the Brain," *John Wiley & Sons*.
3. P.D. Wasserman, "Neural Computing: theory & practice," in *Van Nostrand Reinhold*, 1989.
4. Nestor, "Nestor Development System User's Guide," *Nestor Inc.*, 1988.
5. Olmsted & Watkins, "Neural Network Library," *Olmsted & Watkins Inc*, 1988.
6. P.C. Treleaven, M. Pacheco, and M. Vellasco, "VLSI Architectures for Neural Networks," *IEEE Micro*, vol. 9, no. 6, pp. 8-27, December 1989.
7. R. Hecht-Nielsen, "Performance Limits of Optical, Electro-Optical, and Electronic Neurocomputers," *Optical and Hybrid Computing SPIE*, vol. 634, pp. 277-306, 1986.
8. Hecht Nielsen, "Hecht-Nielsen Neurocomputing ANZA and AXON," *Hecht-Nielsen Neurocomputing*, 1988.
9. SAIC, "DELTA/SIGMA/ANSim, editorial," *Neurocomputers*, vol. 2, no. 1, 1988.
10. NEC, "Neuro-07 (in Japanese)," *Nikkan Kogyo*, February 1989.
11. L.D. Jackel, H.P. Graf, and R.E. Howard, "Electronic neural network chips," *Applied Optics*, vol. 26, pp. 5077-5080, December 1987.
12. H.P. Graf, L.D. Jackel, and W. Hubbard, "VLSI Implementation of a Neural Network Model," *IEEE COMPUTER*, pp. 41-49, March 1988.
13. Angeniol B., "Pygmalion: ESPRIT II Project 2059, Neurocomputing," *IEEE Micro*, vol. 10, no. 6, pp. 28-32, December 1990.

A Novel High-Speed Memory Organization for Fine-Grain Multi-Thread Computing

Herbert H.J. Hum
Centre de recherche informatique
de Montréal
3744 Jean Brillant, Bureau 500
Montreal, Canada, H3T 1P1.
and
McGill University

Guang R. Gao
McGill University
School of Computer Science
McConnell Engineering Building
3480 University St.
Montreal, Canada, H3A 2A7.

Abstract

In this paper, we propose a novel organization of high-speed memories, known as the *register-cache*, for a multi-threaded architecture. As the term suggests, it is organized both as a register file and a cache. Viewed from the execution unit, its contents are addressable similar to ordinary CPU registers using relatively short addresses. From the main memory perspective, it is content addressable, i.e., its contents are tagged just as in conventional caches. The register allocation for the register-cache is adaptively performed at runtime, resulting in a dynamically allocated register file.

A program is compiled into a number of instruction threads called *super-actors*. A super-actor becomes ready for execution only when its input data are physically residing in the register-cache and space is reserved in the register-cache to store its result. Therefore, the execution unit will never stall or 'freeze' when accessing instruction or data. Another advantage is that since registers are dynamically assigned at runtime, register allocation difficulties at compile-time, e.g., allocating registers for subscripted variables of large arrays, can be avoided. Architectural support for overlapping executions of super-actors and main memory operations are provided so that the available concurrency in the underlying machine can be better utilized. The preliminary simulation results seem to be very encouraging: with software pipelined loops, a register-cache of moderate size can keep the execution unit usefully busy.

1 Introduction

The current microelectronics technology is passing the mark of a million transistors per microprocessor chip, and computer architects are facing the increasing challenge of ULSI – *ultra large scale integration* – technology, which may boast the capability of 50—100

million transistors on a chip by the year 2000[10]! One direction to utilize such enormous hardware parallelism is to increase significantly the architectural support for fine-grain parallelism. Examples include superscalar machines which can issue multiple instructions per cycle like the Intel i860 and the IBM RS/6000, superpipelined machines which use deep instruction pipelining like the CDC-7600, or a combination of both[14].

However, conventional single-instruction-stream processors have inherent limitations in fully exploiting instruction level concurrency. This is due to the fact that a processor equipped with only the mechanism of executing a totally ordered instruction stream, lacks the capacity of tolerating long and unpredictable memory and communication latencies – latencies which are unavoidable in a multiprocessing system [4]. An alternative approach is to directly support multiple instruction threads at the processor architecture level, the so-called *multi-threaded architectures*. Multi-threaded architectures have the potential to keep the processor pipelines usefully busy by rapidly switching between threads on long-latency operations. Research on multi-threaded architectures can be found in [1, 15, 17, 13].

For a pipelined multi-threaded architecture, including the target architecture in our research, the ability to exploit the principle of locality (temporal and spatial) is both important and challenging. In conventional modern RISC architectures, the reduction in memory latencies is achieved by providing (explicit) programmable registers and (implicit) high-speed caches. A small number of programmable registers alone can only provide a partial solution – two reasons for this are: (1) the register allocation for subscript variables of array (vector) data is difficult. In fact, "most compilers fail to recognize even the simplest opportunities for re-use of subscripted variables" [7]. (2) Increasing programmable registers will increase the "context" of a thread. If the registers are to be shared between different threads, a large context may present a high context switching overhead.

For the conventional cache solutions, we point out the following important limitations: 1) the published high hit ratios were reported mostly on non-scientific benchmark programs. The effectiveness of a cache for scientific applications where large arrays (vectors) of data are accessed in the computation is less than satisfactory [6]. 2) When a cache miss occurs, the instruction pipeline usually stalls or freezes, causing considerable performance degradation [12]. This degradation will become more severe as the mismatch in processor speed and memory access times continues to grow, as witnessed in new generations of processors. 3) The fact that a conventional cache is transparent to the programmers (compilers) makes performance improvements by optimizing compilers difficult. 4) Lastly, the conventional cache memory is not designed to accomodate multi-threaded architectures. Frequent switchings between instruction threads have a negative impact on the locality of reference. Moreover, multiple active contexts contend for limited cache space, thus further eroding the benefits of the cache because of unwanted cache line replacements. Some notable examples of multi-threaded architectures have rejected the use of caches, such as in the Horizon[17].

The target architecture studied in this paper is the Super-Actor Machine (SAM) — a multi-threaded architecture based on an hybrid dataflow and von Neumann evaluation model. A number of instructions can be issued simultaneously in the SAM so that effective overlapping of floating point ALU operations with other operations can result in a

higher floating point performance. This processing model demands low latency and high throughput from its memory subsystem. In this paper, we propose a novel organization of high-speed memories, known as the *register-cache*. As the term suggests, it is organized both as a register file and a cache. Viewed from the execution unit, its contents are addressable similar to ordinary CPU registers using relatively short addresses. From the main memory perspective, it is content addressable, i.e., its contents are tagged just as in conventional caches. The register allocation for the register-cache is adaptively performed at runtime, resulting in a dynamically allocated register file.

In our execution model, a program is compiled into a number of instruction threads called *super-actors*. A super-actor becomes ready for execution only when: 1) the data dependence is satisfied, i.e., all its input data are generated and its result data from the previous activation, if any, have been used; and 2) space locality is satisfied, i.e., its input data are physically residing in the register-cache and space is reserved there to store its result. The first condition is similar to the so-called *firing rule* in a traditional dataflow machine, however each scheduling quantum in the SAM is an instruction thread instead of one instruction. The second condition, a more radical feature unique to the SAM architecture, ensures that an enabled super-actor can be scheduled for execution only when all memory accesses of its instructions are guaranteed to be in the high-speed buffer memory. Therefore, the execution unit will never freeze when accessing instructions or data. This eliminates one main source of pipeline performance degradation.

To study the multi-threaded capability of the SAM, and indirectly, the effectiveness of the register-cache, detailed simulations were performed. The simulation results are very encouraging: with software pipelined loops, a register-cache of moderate size can keep many threads in concurrent execution and effectively hides the local memory latencies and the latencies associated with fine-grain synchronization support.

In the next section, the Super-Actor Machine's abstract model will be described. Section 3 will provide an overview of the Super-Actor Machine and section 4 will describe the architecture of the register-cache along with the 'check-in process' – the process of ensuring that the required data are present in the register-cache. Section 5 will examine the effects of the register-cache on multi-threaded computing in the Super-Actor Machine via simulation results and analysis, and discussions of related work will ensue. Finally, conclusions are drawn in section 6.

2 The Abstract Model of the Super-Actor Machine

In a dataflow graph, each individual instruction ('actor') is the basic unit of work and scheduling quantum for the underlying machine, and fine-grain synchronization is performed to schedule each instruction. However, some actors can be logically grouped into *threads* so that the cost of synchronization can be reduced by performing the synchronizations only among the threads, while actors within a group can be scheduled via the conventional technique of sequencing with a program counter – à la von Neumann. Grouping instructions into threads and sequentially executing the instructions within the threads while performing dataflow-like fine-grain synchronizations at the thread level is referred to as the *hybrid dataflow/von Neumann* model of computation. It belongs to a

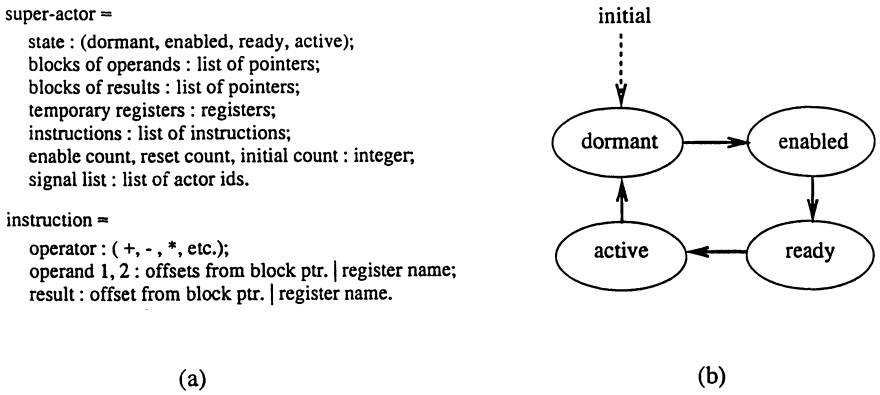


Figure 1: Definition and States of a super-actor.

subclass of the so-called *multi-threaded* architectures [11].

In our work, we call aggregates of one or more actors, *super-actors*. From this point onward, individual actors in the pure dataflow terminology will be referred to as instructions and a thread of one or more of these instructions will be called super-actors, or actors for short. The *attributes* of a super-actor are illustrated in 1 (a), and the operational semantics of a super-actor are defined by the state transition diagram in figure 1 (b).

A super-actor has 4 states, and typically, it goes through the following state transitions: 1) A super-actor is in its *dormant* state when it is waiting on its neigbouring actors to signal to it that it can be enabled. 2) For an *enabled* super-actor to make a transition into the ready state, the following must be prepared: all of the memory blocks containing its input values must be in fast memory and a block of fast memory must be reserved for its result values. 3) A *ready* super-actor enters the active state when it is assigned an available physical domain (context) for it to be executed. 4) Instructions in an *active* super-actor can be scheduled for execution (to be explained later). After execution, an active super-actor will signal its completion to all of the actors requiring notification that it has been executed, and re-enter the dormant state.

One important feature of the architecture model is the *atomicity* of the super-actor activation. Once a super-actor becomes active, it will be executed *atomically* until its completion without the possibility of suspension, i.e., all instructions in the super-actor perform operations entirely *local* to the execution unit, causing no external transactions. Furthermore, it requires no synchronizations with other super-actors during its execution.

Since super-actors are processed atomically, scheduling them based on the data-driven principle will ensure that the data dependencies among the super-actors are satisfied, thus the determinancy of the dataflow computation model is retained, where a node in the super-actor machine model is an instruction thread. A more radical feature unique to the model requires that an enabled super-actor be scheduled for execution only when all

memory accesses of its instructions are guaranteed to be in the high-speed buffer memory. Thus, not only do super-actors decrease the synchronization cost, but they also offer the opportunity to exploit the locality of reference so as to minimize the latencies in memory accesses in the execution system.

Two types of super-actors have been investigated in this paper. In a *sequential super-actor*, the data dependencies between the instructions requires that they be sequentially executed. A sequential super-actor may contain conditional branch instructions which jump to another instruction within the same super-actor. We call these instructions, *short branches*. Conditional instructions which fork multiple super-actors or alter the stream of evaluation of super-actors are restricted to being tail-instructions since the scheduling of super-actors are performed in a separate unit from the execution unit. The second type of super-actor is called a *parallel super-actor*, a special case of sequential super-actors where the instructions are data independent, i.e., instructions within a parallel super-actor does not depend on any results produced by any other instruction within the super-actor. Instructions in parallel super-actors can be executed every pipe beat.

Instructions with long and unpredictable latencies are excluded from ordinary super-actors. These instructions include non local memory access operations, explicit 'send' and 'receive' instructions which perform inter-PE communications, etc. A long-latency instruction is grouped by itself and the actor containing it is called a *long-latency actor* ('L-actor' for short). L-actors will be handled by a dedicated unit.

Furthermore, instructions which modify the memory addresses of the lines a super-actor operates on (its operand or result lines) should be grouped separately into aggregates called *support-actors*. (Instructions which modify memory addresses are used to realize address computations in the underlying machine.) The separation of these instructions from super-actors will be explained when we discuss the architecture of the machine.

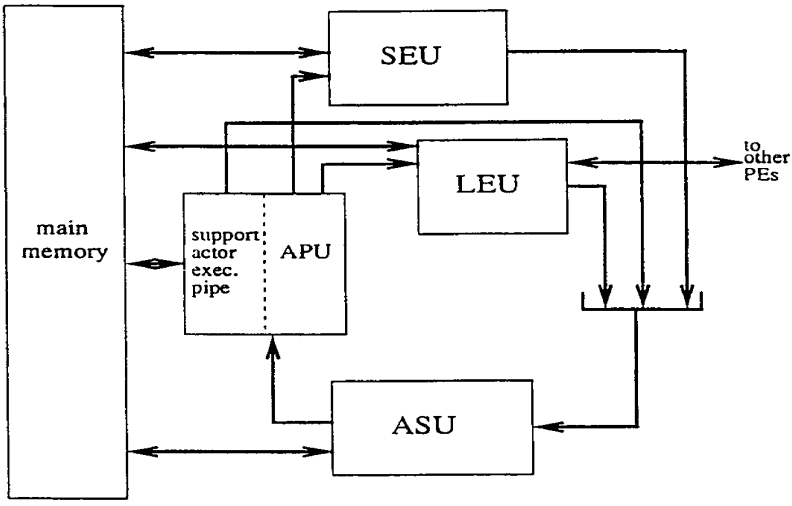
3 The Super-Actor Machine

The Super-Actor Machine is to be a multi-processor system consisting of multiple processing elements linked together by some interconnection network. Memories are distributed to each processor in the machine, and the aggregation of these memories present a global address space which is shared among all processors. In this paper, we will concentrate our discussions on one processing element. Due to space limitations, only a brief overview of the Super-Actor Machine will appear below, nonetheless we hope it provides enough background information for understanding the organization of the register-cache (R-cache).

A processing element of the Super-Actor Machine has 5 basic components: the Super-actor Execution Unit (SEU), the Actor Preparation Unit (APU) which has an adjoining support-actor execution pipe, the Actor Scheduling Unit (ASU), the L-actor Execution Unit (LEU), and the local main memory (cf. fig. 2).

The structure of the execution unit is shown in figure 3. In this section we discuss the smooth execution pipeline and the collection of physical contexts realized by multiple sets of registers. The organization of the register-caches (*i-R-cache* and *d-R-cache*) is left for a later section.

The architecture of the *smooth* execution pipeline is like any standard instruction



SEU = Super-actor Execution Unit
LEU = Long-latency actor Execution Unit
APU = Actor Preparation Unit
ASU = Actor Scheduling Unit

Figure 2: A Processing Element of the Super-Actor Machine.

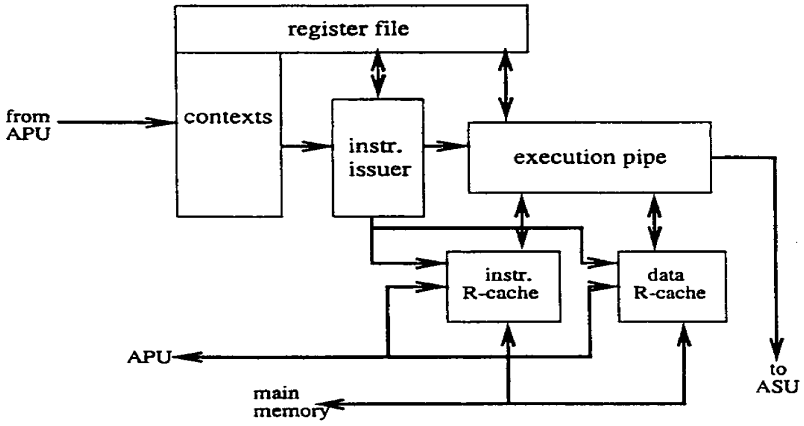


Figure 3: The Super-Actor Execution Unit.

processing pipeline except that the ALU stage is made up of sub-pipes which can handle integer and floating point operations. The other stages are the standard ones like the instruction fetch, operand fetch, etc. The aim of the smooth pipeline is to initiate an independent instruction at a sustained rate of one instruction per cycle, thus the pipeline has the following features: (1) it is *clean* or free of *structural hazards*, and (2) all stages in the pipeline have a uniform and fixed processing time for all types of instructions.

A physical *context*, realized by a set of registers, will be assigned to each super-actor when it becomes active, and will be returned to the pool of free physical contexts when it leaves the execution unit. The purposes of the set of registers are to store information of a super-actor and to be temporary scratch-pad registers for an active sequential super-actor. Values in the registers are not retained after the activation of the super-actor and cannot be used by other super-actors.

All contexts share an *instruction issuer*. The issuer chooses a ready context, increments its counter value and sends the instruction into the execution pipe. An *activation id* is associated with each context and is sent along with each instruction when it enters the execution pipe so that the proper set of registers are used. The issuer is also responsible for sending a 'decrement-reserve-counter' signal to the R-caches when a super-actor exits the SEU (The purpose of this will be explained later). If a context is assigned to a parallel super-actor, it can be ready every machine cycle. Otherwise, the instructions in the super-actor must be executed sequentially and the context must wait for a signal from the execution pipe before it can progress.

Attached to the APU is a simple RISC pipeline which is responsible for processing instructions within a support-actor. The only instructions which the pipeline can process are loads and stores, and integer add and multiply since the sole purpose of support-actors are to perform address calculations, e.g., array indexing, etc. The reason for processing address calculations in the APU is that the R-cache loader must access those calculated addresses, thus the attached RISC pipeline. The LEU is responsible for fetching the instruction and necessary operands from main memory, and processing the long-latency instructions.

Upon completion of an active super-actor, the SEU sends a done signal to the ASU indicating that that super-actor has been executed. The ASU, in turn, processes the signals and decrements the associated enable count of actors. When an actor is enabled (its enable count is zero), its enable count is reset and the enabled actor along with its attributes is sent to the APU. There the enabled actors are enqueued for entry to either the SEU, the LEU, or the support-actor execution pipe. The structure of the ASU and the handling of the signals is similar to the instruction scheduling unit as described in [9].

4 The Register-Cache Architecture

The R-caches are organized both as a register file and a cache. Viewed from the execution unit (SEU), its contents are directly accessible using relatively short addresses; a process similar to the addressing of general registers in conventional CPUs. Moreover, from the APU's perspective, it is content addressable, i.e., its contents are tagged just as in conventional caches (cf. fig. 4). To make effective use of all R-cache lines, the APU will

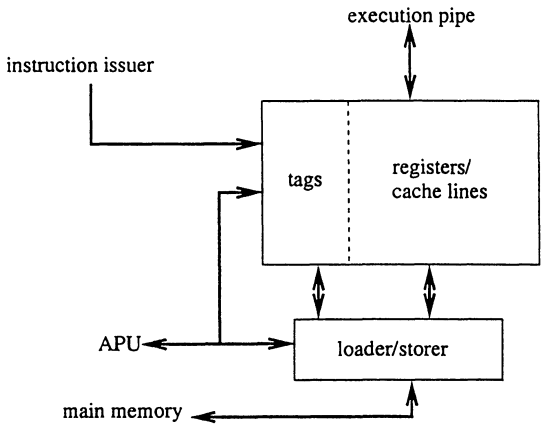


Figure 4: A register-cache.

see a fully-associative cache.

The R-cache retains the transparency feature of conventional caches in the sense that it is not visible to the programmers or compilers; thus, no register allocation by the compiler is required for the R-cache. The allocation of an R-cache line for a block of memory is done entirely at runtime and is performed using cache update and replacement algorithms. Once this is done, the R-cache locations within a line can be accessed by the SEU directly using short addresses, just as if they were general registers (cf. fig. 5). This binding process is called *registering*.

4.1 The Check-In Process for the R-Caches

The APU consists, among others, a queue for enabled parallel and sequential super-actors (PSA/SSA) and a queue for *ready* super-actors (cf. fig. 6).

The APU also contains an R-cache loader which is responsible for *checking-in* enabled super-actors, i.e., ensuring that all the necessary data for the operation of the super-actor (SA) is in the R-cache and that space is reserved in R-cache for its results. The checking-in algorithm is shown in figure 7. An analogy of this check-in process can be found at the ticket counter in an airport. Before boarding the airplane (execution unit), the tour group (super-actor) must first receive their boarding passes indicating that a block of seats are reserved (set of cache lines L it requires is in place). The assignment of seats within the block (relative locations of operands and results) to each member (instruction) of the tour group can be done statically by the tour group manager (at compile-time). However, the final row numbers are assigned dynamically prior to the departure (the processing of the super-actor) during the check-in time. The key (and a divergence from the analogy) is that the SAM architecture can overlap the check-in process of super-actors with the execution of other ready super-actors.

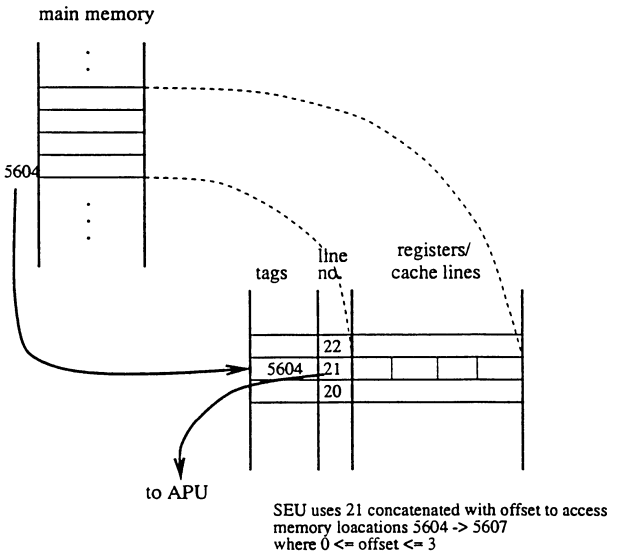


Figure 5: The registering process.

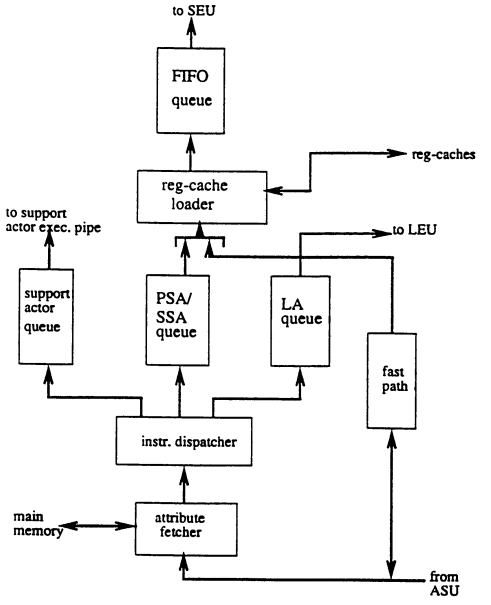


Figure 6: The APU

```
algorithm check-in
    do forever
        if PSA/SSA ready queue has an empty space then
          take next available SA from PSA/SSA enabled queue,
          fetch and/or calculate addr. ptrs. to operand and result line(s), if needed,
          send addr. ptrs. to d-R-cache and receive R-cache line nos.,
          send head-instruction addr. to i-R-cache and receive R-cache line no.,
          package R-cache line nos. with SA packet and put it in PSA/SSA
            ready queue,
        endif
    enddo.
```

Figure 7: The check-in algorithm.

Now let us elaborate on the algorithm. The addresses for the operand and result lines will need to be calculated if they are offset values from the base address. If they are pointer values, then the memory location must be calculated ($ptr + baseaddress$) and the address fetched from the data cache which is shared with the support-actor execution pipe. Otherwise, the values are absolute addresses and are sent to the d-R-cache without modification. Once the addresses are sent to the d-R-cache, the R-cache will return a line number for each address sent. When the loader has received all the line numbers from the two R-caches, it will send the super-actor, consisting of its id, base address, length, instruction and data R-cache line numbers, to the ready PSA/SSA queue. There, it will wait till a context is free in the SEU.

The algorithm describing the operation of the data R-cache is shown in figure 8. The registering process begins when a memory address is sent to the R-cache from the APU. Read-in requests are issued for operand lines and reserve requests are issued for result lines. After the registering process, we say that the instruction is *checked-in*. If the R-cache is full, the Least-Recently-Used (LRU) cache replacement policy is used on lines which are no longer needed (i.e., lines with reserved counters equal to zero) to find a replacement line. The LRU algorithm uses the "age counters" to decide which line to replace. Note that the age counters are only updated by requests from the APU, not by accesses from the SEU.

A decrement-reserved-counter signal along with the line numbers are sent from the instruction issuer in the SEU when a super-actor exits the SEU. Forced write-backs are used to handle super-actors passing their results to long-latency actors, because the LEU does not access the R-cache. Mandatory read-ins are necessary in the d-R-cache because operand lines of a super-actor which was written by a long-latency actor must be brought in since the the LEU can only write into main memory.

The i-R-cache check-in algorithm is similar except simpler due to the fact that it is read only.

```
algorithm d-R-cache
  in parallel do forever
    if read or write request from SEU then service it,
    if decrement-reserved-counter signal from SEU then service it,
    if forced write-back from SEU then
      write corresponding line to memory and send acknowledgement back
        to SEU,
    /* the registering process */
    if mandatory read-in request from SPU then
      use LRU replacement policy on not needed lines if there is no free line
        and write back dirty line if necessary,
      read in from memory, update reserved counter, and send assigned line no. to SPU,
      update age counter of not needed lines,
    else if read-in request from SPU then
      check, in parallel, tags for requested line,
      if there then update reserved counter and send line no. to SPU
      else do as in mandatory read in,
    else if reserve request from SPU then
      check, in parallel, tags for requested line,
      if there then update reserved counter and send line no. to SPU
      else use LRU replacement policy on not needed lines if there is no free line and
              write back dirty line if necessary,
           send assigned line no. to SPU,
      update age counter of not needed lines,
    endif
  enddo.
```

Figure 8: The algorithm for the data register cache.

4.2 The Size of the Register-Cache

For the tandem of the APU and R-cache (i-R-cache or d-R-cache) to function correctly, i.e., the SEU is guaranteed that its operations will always find their value in the register-cache, there is a minimum number of required R-cache lines. The minimum number of lines is $(J + K) \times L$ for $(J + K)$ active super-actors. J is the number of slots in the PSA/SSA ready queue, K is the maximum number of allowable super-actors in the SEU, and L is the maximum number of register-cache lines allocated to a super-actor. The use of the 'reserved counters' in the R-cache guarantees that the reserved or read-in register-cache lines of the $(J + K)$ active super-actors will not be replaced until the super-actor that requested it exits the SEU.

4.3 A Bypass Path to Avoid Unnecessary R-Cache Probing

The bypass path called a *fast-path* is used to avoid unneccessary probing of the R-caches. For super-actors in loop constructs which are enabled every time the loop iterates, the lines they use might still be in the register-caches when they are enabled. These super-actors can be tagged by the compiler as possible *fast-path candidates* so that when they are enabled, a small cache (called an *actor-cache*) containing the recently fired super-actors can be checked for its presence. To ensure that the super-actors present in the actor-cache would still have their lines in the R-caches, the cache should only have $J + K$ entries. If an enabled super-actor has its entry in the actor-cache, then the cache line numbers which it used previously are retrieved, the lines reserved, and the super-actor enters its ready state immediately. But if that instance of a super-actor is not present in the actor-cache, it will be sent back to the regular path where its other attributes can be fetched and the R-caches probed.

5 Performance Effects of the Register-Cache

To investigate the effects of the register-cache on multi-thread computing, a detailed simulator was written in Common Lisp with Flavors. The architectural simulator models the Super-Actor Machine down to the machine cycle level, i.e., for each object modelling a particular task, a processing time of x machine cycles was assigned for which the authors believe is attainable with today's device technology.

5.1 The Simulated Architecture

In the simulations, some of the following design parameters were arrived at arbitrarily while others were based on rough calculations of the requirements for efficient processing.

The local main memory is made up of 16 banks with access times of 6 machine cycles and a memory controller regulating access to them. Addresses are interleaved amongst the banks and each bank can service a request independently from the others. FIFO queues are used to smooth out the throughput rates between the ASU, APU, SEU and LEU.

The signal processor and the enable controller of the ASU are pipelined functional blocks with a pipe beat of 1 machine cyle. A 1K word 4-way set-associative cache with a line size of 8 words is used to buffer requests for signal lists from the signal processor to the main memory. The enable controller goes through a 512 word 4-way set-associative cache with a line size of 4 words. (Note that an enable count is only 4 bits, so 1 word contains the enable counts of 8 actors.)

There are 16 available physical contexts in the SEU, so there are 16 sets of registers where each set contains 8 32-bit registers. We chose the value 16 because the execution pipe has 10 stages through its longest path, and if the contexts contained only sequential super-actors, then we would require a minimum of 10 active super-actors to keep the pipe fully busy. The execution pipeline has a pipe beat of 1 machine cyle and the instruction issuer is also pipelined with a cycle time of 1 machine cycle. The floating point add, multiply and approximate reciprocal pipes are six stages long with a pipe beat of 1 cycle, and the integer pipe is one stage long. Fetch and stores from the register set or register-cache each take 1 cycle. The i-R-cache is 1K words with 16 words per line. This implies that a super-actor can only contain a maximum of 16 instructions. It has been found that the average grouping of dataflow actors are of size 4 [3], so 16 should be plenty[1]. The d-R-cache is also 1K words with 4 words per line and the path to main memory is 4 words wide. Each super-actor is allowed a maximum of 4 lines.

The fetching of actor attributes in the APU goes through a 1K word 4-way set-associatve cache with a line size of 8 words. The R-cache loader can load a R-cache line in a maximum of 8 cycles (one to form the address, 6 for the access and one to load). Requests to the i-R-cache and d-R-cache are performed in parallel. The actor-cache in the fast-path has space for 63 entries.

The support-actor execution pipe is a basic RISC pipe with a pipe beat of 1 cycle. The i-cache and d-cache are both 1K words 4-way set-associative with a line size of 4 words. The LEU was not modelled since the preliminary experiments were only used to investigate the impact of the register-caches in a PE of the Super-Actor Machine.

5.2 The Test Programs

Work is currently under way in producing an assembler for SAMAL (Super-Actor Machine Assembly Language) and in generating SAMAL code from the program graph form of the Id compiler[18]. For this study, we have hand-coded three small benchmark programs.

The three benchmark programs are: SAXPY, SAXPBYPC, and Lawrence Livermore Loop 1. SAXPBYPC is the same FOR-loop construct as SAXPY except the expression is $a * X[i] + b * Y[i] + c$ instead of $a * X[i] + Y[i]$. For all three benchmarks, the loops were unrolled four times so that parallel super-actors can be formed by aggregating four identical operators in the loop[2]. The index sequencing is handled by a sequential super-actor which can either trigger super-actors in the loop body or exit when it is finished. The

[1]This does not mean that the rest of the cache line goes to waste. In fact, other super-actors can share the same i-cache-line, the only requirement being that an assembler or compiler must handle the arrangement of instructions into contiguous blocks which are aligned on 16-word boundaries.

[2]Other techniques for grouping instructions into super-actors are possible, but they are not investigated in this paper.

Benchmark	1 Loop		4 Loops		
	execution time (cycles)	SEU utilization	execution time (cycles)	SEU utilization	Speedup
SAXPY	30799	13%	8621	46%	3.6
SAXPBYPC	33527	19%	10797	59%	3.1
Loop1	32151	23%	13442	56%	2.4

Table 1: Results for SAXPY, SAXPBYPC and Loop1.

arrays which the loops process are stored locally in main memory and support-actors are used to perform address calculations for the super-actors in the loop body. The indexing super-actor has 6 instructions while the parallel super-actors in the loop body have 4 instructions each. The support-actors have an average of 7 instructions each.

Dataflow software pipelining[8] was utilized to increase the amount of exposed parallelism in the programs. With dataflow software pipelining, a code body for SAXPBYPC was reconstructed with 4 stages, thus handling 2 simultaneous iterations and exposing more parallelism, while a code body for Loop1 was reconstructed with 5 stages. However, SAXPY could not really benefit from software pipelining due to its small loop body – only 2 stages were produced when it was software pipelined.

5.3 Simulation Results

Two versions of each program were written: one which only had one loop that iterated from element 1 to 1200, and the other version with 4 simultaneous loops, i.e., each loop was invoked in parallel where a loop iterated through 300 elements. The results are shown in table 1. The reader should note that several operations can be issued each cycle besides an ALU operation in the SEU, and the utilization rate shown in table 1 does not reflect the processing of those other operations. The speedup factor was calculated by dividing the execution time for the 1-loop version by that of the 4-loop version. From the table, one can conclude that more parallelism results in a higher utilization of the SEU and the more parallelism the compiler exposes in a program, the faster the execution due to the opportunity of overlapping memory loads to the R-cache with the processing in the SEU. However, there appears to be a limit as to how much exposed parallelism the current configuration of the Super-Actor Machine can exploit, as shown by the smaller speedup for Loop1 as opposed to SAXPY and SAXPBYPC, and by the leveling off of the SEU utilization.

On closer monitoring of the simulator, we realized that the register-cache loader had become a bottleneck because of the increased number of ready super-actors which must wait to enter the loader. To overcome this bottleneck for memory-intensive programs, the loading phase could be sped up by pipelining the R-cache loads or a load request can bring in a larger memory block. We decided to investigate the latter since it is less costly in terms of hardware. In the following experiment, the d-R-cache line is increased to 8

	8-word			16-word		
loop(s)	execution time (cycles)	SEU util.	Speedup	execution time (cycles)	SEU util.	Speedup
1	15759	19%	1.0	8845	30%	1.0
2	7981	38%	2.0	5321	50%	1.7
4	4897	62%	3.2	3690	73%	2.4
8	4005	76%	3.9	3422	80%	2.6

Table 2: SAXPY for 8-word and 16-word long d-R-cache line.

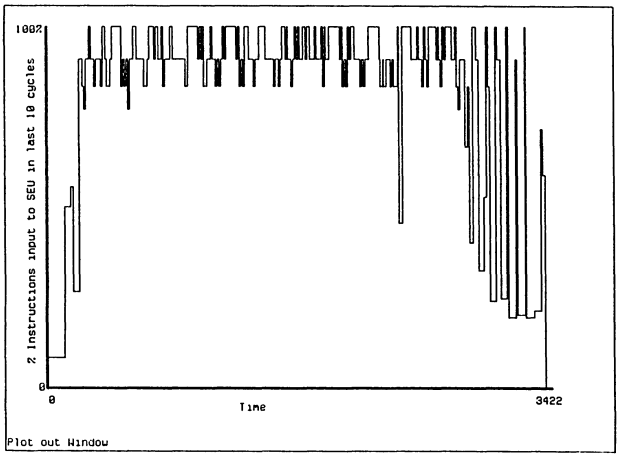


Figure 9: SEU utilization profile for the 8-loop, 16-word wide version of SAXPY.

words and 16 words[3], so the R-cache size increases accordingly. The i-R-cache organization remains the same since the real backlog is caused by the loading of array elements into the d-R-cache. To exploit the larger d-R-cache lines, SAXPY was re-written with an unrolling factor of 8 and 16 so that a parallel super-actor has 8 and 16 instructions respectively. Table 2 shows the results of this experiment where the loops iterated from 1 to 1152. The numbers in the speedup column is relative to the 1-loop case. As one may notice, the SEU utilization has increased significantly for the 16-word R-cache line case as compared to the 4-word R-cache case. Indeed, the bandwidth from R-cache to main memory is a major factor in keeping the SEU usefully busy. In fact, for the 8 simultaneous loop and 16-word R-cache line case, the SEU can be kept at about 90% busy when the start-up and wind-down phases are discounted (fig. 9).

[3]A 16-word cache line is not impossible, the IBM RS6000 has a 128 byte cache line.

5.4 Discussions and Related Work

Local memory latencies and the latencies associated with fine-grain synchronization have always been a challenge to dataflow architectures. The above simulation results show that with enough exposed instruction-level parallelism, these two sources of latencies can be effectively hidden via the actor-preparation unit and R-cache mechanism of the SAM. However, further research remains in addressing the efficiency issues and the multitude of tradeoffs present in the Super-Actor Machine.

In the SAM, a number of supporting operations can be issued at each cycle while the SEU is issuing an ALU operation. These operations may include: (1) an integer operation (+/-) in the APU for address calculation, (2) a memory load/store operation (integer or floating-point), (3) a 'fork' or 'join' operation in the ASU[4], and (4) some long-latency operation in the LEU. Moreover, the execution pipes in the SEU are fully pipelined. In this manner, one processing element of the SAM can be considered a superscalar, super-pipelined machine as defined in [14]. However, it really belongs in a superset of this class due to its support for partial order execution and the issuing of multiple instructions from multiple streams. (A superscalar machine issues multiple instructions around one program counter, i.e. from a single stream.) To roughly guage the efficiency of the SAM in hiding local memory latencies and fine-grain synchronization costs, we examined the performance of SAXPY on the IBM RS6000/530 – a superscalar, somewhat superpipelined machine. The sustained performance is about 1 floating-point operation per clock cycle[5] when the combined floating-point multiply add instruction is used. In the SAM, a 0.76 floating-point operation per cycle can be obtained on SAXPY (the 8-loop, 16-word version). Note that the floating-point pipe on the SAM is 6 stages deep whereas the RS6000 only has a 3-stage floating pipe. This crude comparison has given us great hope in that a fine-grain multi-threaded architecture can indeed hide the overhead associated with fine-grain synchronization.

Some multi-threaded architecture researchers are also investigating high-speed memory between main memory and the execution unit. In particular, Nikhil and Arvind (P-RISC)[15], Agarwal et al (APRIL)[1], and Iannucci (EMPIRE)[13]. In P-RISC as in EMPIRE, the key to exploiting temporal and spatial locality is to continuously process all the threads within an active frame (the frame contains threads of a code block) until there are no more. A set of active frames would be kept in a high speed memory to minimize the local memory latency. EMPIRE is more unique in that a frame must be in fast memory before any of its threads are picked for execution. These architectures have addressed the local memory latencies, however, their fine-grain synchronization support mechanism is within the execution unit, thus those associated costs cannot be totally hidden. In APRIL, a regular cache is put between main memory and a RISC processor. For its cache to be effective, a small amount of active threads can only be supported, otherwise cache interference can have deleterious effects. Since only a limited number of active threads are loaded into fast memory at any given time, the ominous possibility of constantly loading and unloading active threads exists. Furthermore, the fine-grain synchronization support is also embedded within the execution unit. With all RISC implementations, the question

[4]We note that the ASU performs fork/join operations implicitly through signal processing functions, while some other multi-threaded architectures may execute explicit machine instructions [16].

of floating-point performance remains.

Thus far, we have only mentioned architectures with non-superscalar processing elements. Burton Smith's Tera computer[2], however, is a multi-threaded superscalar architecture. Again, his architecture does not rely on cache memory, instead he utilizes a huge register set file. It will be very interesting to quantitatively compare the SAM with the above mentioned architectures.

6 Conclusion

The SAM architecture has put a strong demand on its memory organization in terms of low latency and high throughput requirements. In this paper, we have presented the organization of the register-cache as a high-speed buffer memory to meet such a demand. It ensures a low and fixed latency to support the highly pipelined instruction processing capability in the execution unit. Architectural support for overlapping the super-actor execution and main memory operations are introduced to provide high memory bandwidth. An adaptive "check-in" process which dynamically registers and binds the keys to access register-cache lines helps to avoid some of the difficulties of register allocation at compile-time, in particular, for subscript variables of large arrays containing floating-point numbers. The preliminary simulation results have provided evidence of the effectiveness of the register-cache in hiding local memory latencies and the latencies associated with supporting fine-grain synchronization.

7 Acknowledgment

We would like to thank the Natural Sciences and Engineering Research Council for their support. We are very grateful to the members of the ACAPS group for their interesting discussions on this subject, in particular, Erik Altman, Russ Olsen, Philip Wong, and Kevin Theobald. Finally, we would like to thank Dr. J.B. Dennis for his comments on an earlier draft of this paper.

References

[1] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 104–114, 1990.

[2] R. Alverson et al. The Tera computer system. In *Proc. of the 1990 Int'l. Conf. on Supercomputing*, 1990.

[3] Arvind. Personal communication, 1990.

[4] Arvind and R. A. Iannucci. Two fundamental issues in multiprocessing. Computation Structures Group Memo 226, Laboratory for Computer Science, MIT, 1987.

[5] R. Bell. IBM RISC system/6000 preformance tuning for numerically intensive FORTRAN and C programs. Technical Report GG24-3611, IBM Int'l. Technical Support Center, Aug. 1990.

[6] D. Callahan and A. Porterfield. Data cache performance of supercomputer applications. In *Proc. of the Supercomputing '90 Conference*, pages 564—572, New York, New York, 1990.

[7] David Callahan, Steve Carr, and Ken Kennedy. Improving register allocation for subscripted variables. *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990. White Plains, NY.

[8] G. R. Gao, H. H. J. Hum, and Y. B. Wong. Towards efficient fine-grain software pipelining. In *Proceedings of the ACM International Conference on Supercomputing*, Amsterdam, Netherlands, June 1990.

[9] G. R. Gao, R. Tio, and H. J. Hum. Design of an efficient dataflow architecture without dataflow. In *Proceedings of the International Conference on Fifth-Generation Computers*, pages 861–868, Tokyo, Japan, December 1988.

[10] P.P. Gelsinger et al. Microprocessors circa 2000. *IEEE Spectrum*, pages 43—47, Oct. 1989.

[11] R. H. Halstead Jr and T. Fujita. MASA: A multithreaded processor architecture for parallel symbolic computing. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 443–451, 1988.

[12] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgain Kaufman Publishers Inc., San Mateo, CA, 1990.

[13] R. A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140, 1988.

[14] N.P. Jouppi and D.W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Third Int'l. Conf. on Arch. Support for Prog. Lang. and Operating Sys.*, pages 272—282, 1988.

[15] R. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 262–272, Israel, 1989.

[16] R. S. Nikhil and Arvind. Id: A language with implicit parallelism. Computation Structures Group Memo 305, Laboratory for Computer Science, MIT, 1990.

[17] M.R. Thistle and B.J. Smith. A processor architecture for Horizon. In *Proc. of the Supercomputing Conference '88*, Florida, 1988.

[18] K. R. Traub. Sequential implementation of lenient programming languages. Technical Report MIT/LCS/TR-417, Laboratory for Computer Science, MIT, 1988.

Evaluation of Futurebus hierarchical caching

K.G. Langendoen H.L. Muller
L.O. Hertzberger
University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
koen@fwi.uva.nl

Abstract

This paper presents a simulation model for hierarchically structured multiprocessors based on the Futurebus+. The model simulates the behaviour of the buses and caches at the level of individual memory references. These memory references are generated by a set of "stochastical processes" which are based on measured statistics of actual programs. The model is validated with published trace driven simulations of single and two level cache systems.

We have used the model in some experiments to study the performance effects of cache parameters in various multilevel cache hierarchies. We conclude that a two level hierarchy of caches is attractive for those applications that cause a lot of bus traffic. The parallel application of our benchmark, which heavily uses shared data, showed a performance increase of 44% when a flat bus was replaced by a two-level hierarchy. Finally we observed that 99% of the total of bus transactions in all simulations used only 5% of the Futurebus+ cache-coherency protocol. We conclude that many of the optimizations in the protocol only increase complexity without a clear performance benefit.

1 Introduction

The current generation of bus based shared-memory multiprocessors is equipped with caches for two reasons. First the performance of modern processors critically depends on the usage of caches, and secondly caches reduce bus traffic and therefore bus contention in a multiprocessor system. The caches run a cache-coherency protocol to keep the data values consistent. Many of these protocols require the caches to snoop (monitor) all traffic on the memory bus and take appropriate action when they have the requested data, they depend on the broadcast capability of the memory bus. Since this severely limits the number of processors in a machine coherency protocols have been designed to operate in systems with a *hierarchy* of buses. Caches located between two levels snoop traffic at both sides and pass requests and responses on to the other level when necessary.

The Futurebus+[1] [Futurebus89] is an industry standard bus definition for scalable

[1]Note, in the sequel we omit the '+' for readability.

shared memory multiprocessors. It is the definition of a high performance multiprocessor system bus and includes a hierarchical cache coherency protocol. Because the Futurebus is targeted as a general industry standard for several generations of computer systems, it is defined in architecture, processor, and technology independent terms. For example the cache protocol is described in terms of lines and sets. As a consequence the Futurebus specification has many degrees of freedom (transfer rate, cache size, etc) which have to be fixed for an actual implementation. In general it is quite difficult to find the "optimal" set of parameter values because many of the relations between the parameters are unclear. Since experimenting with real hardware is expensive we need a performance model to evaluate various design alternatives.

Despite the great interest in caches, hence the volume of literature, we were not able to find a suitable multiprocessor model which includes both hierarchical and snooping caches. Many different snooping protocols for single bus systems have been described and compared, see for example [Archibald86] and [Eggers89]. Hierarchical cache memories have been studied in the context of fast single processor system; [Bugge90] is an interesting example because it uses the Futurebus. Some analytical models of hierarchical snooping cache systems do exist, but either these models are too restrictive or the parameters are too high level. For example, in [Vernon89] one has to specify the probability that a cache miss at some level can be satisfied by a neighboring cache at the same level. We do not want to specify such a parameter, but rather derive it from results obtained with the model.

The simulation model presented in this paper is based on the Futurebus specification and can model a wide range of hierarchically structured multiprocessors. The model is heavily parameterized so it can be used as an experimentation tool to gain insight in the behaviour and capacities of the Futurebus protocol. To capture the effects of some low level parameters the simulator operates at the level of individual bus cycles and records the complete status information of cached data. Although this requires large amounts of memory and computation the model is still capable of simulating an eight 20 MIPS processor system; a simulation run representing 50 millisecond takes 1 hour on a SUN-4.

The remainder of the paper starts with a detailed description of the simulation model in section 2. Next, some experiments to validate the model with published measurements are described in section 3. In section 4, we discuss some experiments. We determine the optimal line sizes of caches in a simple hierarchical system, and study the effects of various degrees of associativity. Then we fix the line size and associativity, and determine the performance effects of different multiprocessor topologies. The bus hierarchy varies from a flat bus up to a three level bus system. The conclusions about these experiments are summarized in section 5.

2 Model

The multiprocessor model covers the complete trajectory from application down to bus cycles and consists of two parts as depicted in figure 1. The upper layer models applications running on multiple processors and generates a sequence of memory read/write requests (an address trace), which is serviced by the lower layer. This memory layer models a hierarchy of caches and Futurebuses.

We have imposed some restrictions on the application-memory interface to keep the

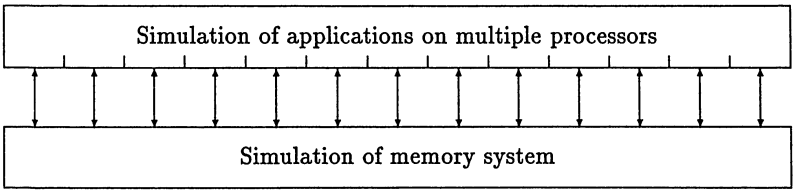


Figure 1: Simulation layering.

simulation model feasible:

- Only the addresses of memory locations are considered, not their values. This excludes the interpretation of the applications at assembly level, but the amount of memory required to store the contents of all simulated memory and caches of a multiprocessor would simply exceed the capacity of our computer systems. Besides, interpretation of assembly code is also unfeasible in this case because of computational demands.
- The interface is limited to simple read/write requests, no I/O instructions or read-modify-write cycles are considered.
- We do not consider the mapping of virtual to physical addresses. Instead we assume that processors directly generate physical addresses and that all applications fit into main memory: there are no page faults.

These limitations make it feasible to exactly simulate the Futurebus cache coherency protocol in the memory layer. The simulator only has to record the status information of each cache line. The applications, on the contrary, do have to be modeled as simple processes generating memory references and may not depend on the result values. This is elaborated in the following sections.

2.1 Application layer

The application layer is responsible for generating memory references to drive the memory layer and models both applications and processors. Since the memory layer only handles memory addresses, the application layer just has to generate an address trace. Although appealing, the usage of real multiprocessor traces of existing programs has a few drawbacks. First the traces will be huge because of the number of processors and the amount of references needed to avoid the long-term cold start effects of second level caches. The measurement and handling of such large traces is a research topic in its own right. Secondly a multiprocessor address trace lacks flexibility because it is only valid for one specific multiprocessor configuration.

To avoid the drawbacks of real multiprocessor traces we use the alternative of stochastical simulation. Applications are modeled as simple processes that generate 'random' addresses. To capture the locality in access patterns of real world programs, a model is used which distinguishes instruction, stack, and data access, each with their typical locality, see section 2.1.1. By tuning a few parameters, like grainsize and data access rate, a

diversity of applications can be specified. The major drawback of stochastical simulation is its inaccuracy. Therefor we have validated the model with the experiments described in section 3.

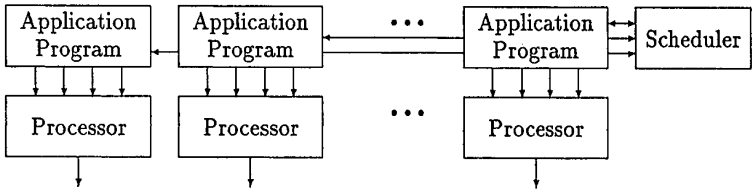


Figure 2: Structure of application layer.

The application layer of figure 1 is detailed in figure 2. Each application process generates an address trace based on measurements of a specific real world application. The trace is fed to a processor filter, which tailors it to a certain type of processor. This enables us to model processor speed, instruction density, register banks (by masking off some of the stack accesses), memory pipelines, etcetera. The scheduler maintains a FIFO queue of ready applications and allocates them to processors on context switches. Note that during a single simulation run, a program can be scheduled in on a different processor from where it was scheduled out last time. Like in a real machine, more than one instance of an application can run at the same time.

2.1.1 Application model

An application is modeled as a simple loop process. Each iteration generates an instruction fetch and possibly some stack and data references. We distinguish three types of instructions to take care of locality in the instruction stream:

1. Jump instructions. The next instruction is an instruction in the neighborhood of this instruction. The jump target is randomly selected from a normal distribution. Both the average distance of jumps and the probability of forward jumps are parameters of the application. It turns out that a normal distribution reasonably matches the real world jump behaviour.

2. Call/return subroutine instructions. The program control transfers to one of the subroutine entry points. The selection of an entry point is based on an exponential distribution to model the typical runtime preference of a program. The subroutines themselves are uniformly distributed over the text segment.

3. Arithmetic instructions. This category includes all other instructions and the program simply continues with the next instruction.

The text segment is shared by all instances of the same application. Besides instruction fetches, an application generates data references as well. An application contains three different data segments with their own locality:

Stack. The application maintains a stackpointer, accesses to the stack are in the neighborhood of this stackpointer. The stackpointer is updated when a call- or return-subroutine instruction is simulated. The average size of the stack frame, and the chance of doing a stack access are application specific parameters.

Each application has its own private stack segment.

Private data. When an instruction references this segment the memory address is selected from a normal distribution around the previous reference. This simple scheme is not very realistic and we plan to improve upon it by using a set of datapointers analogous to the subroutine entry points in the text segment. That would be a simplified version of the model described in [Archibald86], who uses an exponential distribution over a LRU-chain of data references.

Each application has its own private data segment.

Shared data. The address selection is the same as for the private data segment, but this segment is shared with all instances of that application. Different types of shared applications can be modeled by varying the access rate and load/store ratio of this segment.

For each of the segments a pair of parameters specifies the access rate and load/store ratio of the data references. After a sequence of instructions an application process will do a context switch to make a system call. The scheduler puts the application at the end of the ready list, runs a kernel job for a short time, and allocates a new application to the idle processor. The context switch interval is a model parameter and controls the grainsize of the application.

2.1.2 Processor model

The task of the processor model is to adapt the traces coming from the application to a specific processor. RISC and CISC architectures show their own particularities in the address trace. RISCs tend to do more instruction accesses at a higher clock frequency and have larger code sizes, while CISCs do more data accesses, because of less register usage. The following parameters are used to transform the universal application trace into a processor trace:

Speed. The instruction fetches of the application trace are fed to the memory layer at the MIPS rate of the specific processor. Note that the MIPS rate influences the stack and data references as well.

Instruction size, Instruction power. A CISC uses fewer instructions than a RISC to perform the same application and instructions are coded more dense as well. Hence, the average jump distance and text size of an application have to be scaled accordingly. Therefore the processor model includes two scaling parameters: instruction power and instruction density. These can not be merged into one parameter because they scale differently: The density operates on all instructions, whereas the instruction power only affects the arithmetic instructions because the absolute number of jump instructions doesn't vary between RISCs and CISCs.

Register usage. The usage of register windows, large register banks, and compiler optimizations leads to reduced data traffic. Especially the accesses to the stack segment are greatly reduced. The processor model accounts for this effect by masking off some of the stack references from the application trace, they are simply discarded. The register-usage parameter specifies the percentage of masked references.

Context size. The size of the data that should be saved and restored on context switches. This is related to the number of registers.

2.2 Memory layer

The memory layer of figure 1 simulates buses, caches, and shared memory at bus cycle level. To exactly simulate the behaviour of the Futurebus cache coherency protocol, the simulator maintains the tags and states of all cache lines. The simulator is written in Pearl [Muller90], an object oriented language for architectural simulation and evaluation. By modeling the buses, caches, and memory as individual objects it is easy to specify arbitrary multiprocessor configurations.

Some examples of multiprocessor cache hierarchies are shown in figure 3. The top level caches are connected to the address generators found in the application layer. In the remainder of the paper we often name caches by their level number. We start counting at the processor, which is connected to a first-level or primary cache. When referring to other caches in the hierarchy relative to a specific cache, we use downwards to denote caches closer to memory, and upwards for the ones closer to the processors. Likewise we refer to buses.

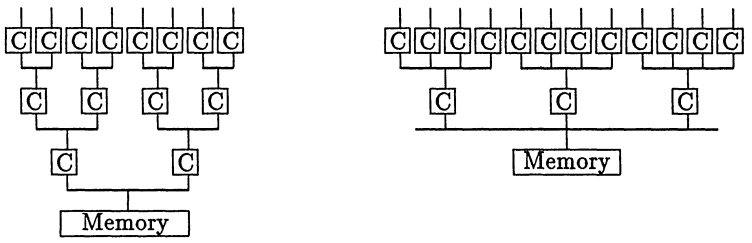


Figure 3: Two example memory hierarchies.

To simplify the simulator all caches in the memory hierarchy have to use the same line size. Furthermore all data transport in the memory layer occurs as line transfers. The performance deviation in case of a write back of a single word will be quite small because of the large setup time compared to the data transmission speed.

2.2.1 Bus model

In general bus specifications define the electrical behaviour of the bus, and the protocol how to drive the bus wires to perform basic actions like read and write. The Futurebus specification, however, also includes a cache coherency protocol, and even a complete

message protocol. In this article, we ignore the message protocol and the electrical layer. We concentrate on the transaction layer and the cache coherency protocol. The bus model implements the transaction layer, it handles bus requests from caches in FIFO order and broadcasts the transactions to all other devices connected to the bus. The speed of the bus transactions is specified with two parameters: the setup (arbitration) time and the transmission speed. The cache coherency protocol is implemented by the cache model.

2.2.2 Cache model

The Futurebus specifies a MOESI-like[Sweazy86] cache coherency protocol for hierarchical caches. Only three states are used in the Futurebus protocol: *Exclusive* (modified), *Shared* (unmodified), and *Invalid*. The protocol is described with four basic bus transactions: `read_shared`, `read_modify`, `write`, and `invalidate`. These actions are for reading (shared) data, for reading/writing exclusive data, for writing modified data back and for invalidating a cache line. In normal operation, when there are no cache coherency conflicts, reads and writes of a bus are handled by the cache or memory downwards in the hierarchy. However, when a cache monitors on the downwards bus that a neighbouring cache wants to access data it owns exclusively then it intervenes in the bus transaction and passes the data to the requesting cache. An example of the cache protocol is shown in figure 4: cache `A` tries to read data that is exclusively owned by `B`. Cache `B` intervenes in the bus transaction and supplies the data, thereby preventing the memory to answer the read request (with outdated data).

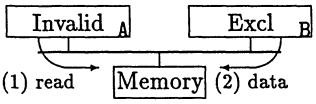


Figure 4: An example of an intervening cache.

In a multi-level hierarchical system, it frequently happens that a cache has to intervene a bus request, but cannot immediately answer the request because it has to get the data from another bus segment. To prevent the first bus from being blocked for a long time, the Futurebus introduces the concept of a *split*. The *split* of a bus transaction causes the requester of the bus action to suspend itself and release the bus for other transactions. The splitting cache issues a response transfer as soon as the answer is available. The originator of the request catches this response, and continues with the original transaction. In the meantime, both the cache and the bus are free to be used for requests of other cache lines.

Figure 5 shows two example cases of a split transaction. In the first simple case, cache `B` issues a split because it does not have the data requested by `A` and repeats the read on the lower bus. In the second case, cache `C` tries to read a value. Cache `D` intervenes this read because it registered that some cache up in the hierarchy has the actual data. It splits the read and issues a read transfer on the upper bus where cache `E` will respond because it has the data. The data is then propagated downwards by cache `D` which issues a response transfer on the bus connected with cache `C`. Cache `C` catches the response and finishes its pending read request by supplying the data to the upper bus.

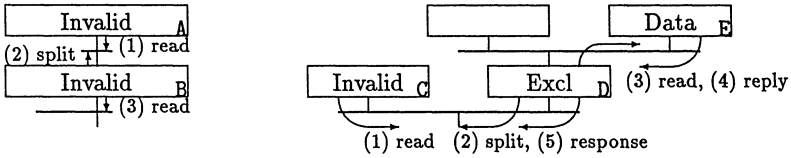


Figure 5: Two examples of *splits*.

The Futurebus protocol does not specify which cache line to replace when a set is full, nor what actions to take during the replacement. Things get really hairy when neighboring caches start referencing either the line selected for replacement, or the line that caused the replacement. Our cache model randomly selects a line for replacement and carefully records the status of both lines.

The cache simulator is implemented as a state machine that reacts to bus transfers. A cache snoops all bus cycles on both buses and has to be capable of handling two transactions in parallel, so the transition table becomes quite large. In practice, however, only a small fraction of the table is used; specifying 252 entries out of the 8620 suffices to run all simulations described in the following sections. The transition-table usage is discussed further in section 4.4.

2.2.3 Memory model

The task of the memory model is easy: Just listen to the bus and service read/write requests unless one of the caches on the bus intervenes. The memory is characterized by its service time. Since we do not store data values, the size of the memory is not relevant.

3 Validation

As noted, the stochastical application model, described in section 2.1.1, introduces inaccuracies. Especially the locality in data and text references is a potentially weak point. To verify the stochastical application model we made simulation-runs to compare the model results with experiments described in literature. The published results have been derived from real world address traces, and cover important aspects of our hierarchical multiprocessor model:

- [Hennessy90, Hill87]: A uniprocessor system with one single level cache. The trace is derived from a UNIX environment.
- [Bugge90]: A uniprocessor system with a two-level cache hierarchy. The trace is derived from some data manipulation programs running under SINTRAN III.

Running the same experiments with our simulation model required the specification of the model parameters of section 2. The articles provide values for many of the memory-layer parameters, but none for the application parameters since these are hidden in the address traces. The next section discusses the application benchmark we have used as

workload in various experiments. The two sections thereafter present the validation results.

3.1 Benchmark

To get realistic application parameters we have taken values from [Hennessy90], which lists some program specific parameters like instruction usage distribution and percentage of data references. We did additional measurements of UNIX programs to obtain the remaining parameters. This included parameters like size of text/data segment, number of procedures, and size of stackframe. To better quantify the effect of sharing on system performance, we have defined two job mixes:

UNIX mix: A set of jobs consisting of multiple editors (vi), TeX formatters and C-compilers together with a UNIX kernel. All instances (jobs) of one application program share one text segment, but have their private stack and data segments. The UNIX kernel is a small process duplicated on each processor, which shares both text and data segments.

Fine grained parallel program: A set of identical jobs that generate many read/write references to a small shared data segment. The grainsize is controlled by setting the context-switch rate (which is typically much higher than for the UNIX jobs) and by setting the access rate to the shared data. Since locality and access rate of shared data is essential to performance, we took parameters (10Kbyte segment, one access per 10 instructions) which give comparable write-broadcast ratios as reported in [Eggers89]. In this way, our parameters are in the right order of magnitude, but we emphasize that real parallel programs may behave quite different.

In the future we would like to include the concept of locking in our simulator, to model synchronization phases in a fine grain application. At this moment the memory layer can not support locks because it handles only addresses of memory locations, not their contents. We foresee to include a special lock manager which keeps the lock value. A lock operation in an application will first issue a read/write to the memory layer to get access to the lock, and then get/set the lock by consulting the lock manager.

3.2 Single processor, one-level cache validation.

This experiment is described in [Hennessy90, Hill87]. It is a trace driven simulation of a VAX processor with a single cache, running a multi programming workload. The cache size and associativity are variable. The study of [Hill87] has decomposed the cache miss rate into three fractions, but we will only use the reported total miss rates. We have run the same experiments on our simulator with the UNIX-mix benchmark as workload.

The data in figure 6 shows that our simulation model compares well to the measurements of [Hill87]. The miss rates for small caches are a bit too high, and the miss rates for larger caches are a bit too low. For caches with increased associativity the miss rates show similar results with a slightly higher deviation for large cache sizes.

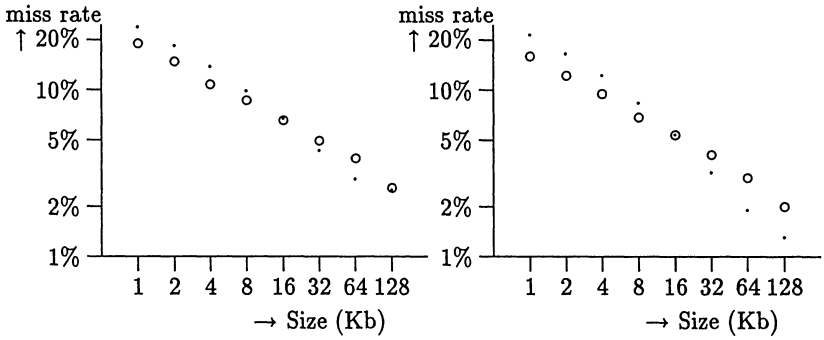


Figure 6: Miss rates measured by [Hill] (circles) and simulated miss rates (dots). A direct mapped cache (left) and a 2-way associative cache.

3.3 Single processor, two-level cache validation.

This experiment considers a uniprocessor with a small fast primary cache, followed by a large secondary cache. [Bugge90] reports the miss rates of the secondary cache with various parameter settings of line size, associativity, and total cache size. The primary cache is fixed as a 128Kb direct-mapped cache with 16 byte lines. To deal with the effects of cold start misses in the address trace, that paper contains three miss rates: a *worst*, *best*, and *estimate* case. The worst case assumes that cold start misses are indeed real misses, whereas the best case counts those misses as hits. The estimated miss rate simply ignores cold start misses (nor miss, nor hit).

We did the same experiment using our stochastical address-trace synthesizer. Since our model is limited to equal line sizes in all levels of the hierarchy, and [Bugge90] uses a 16 byte line for the primary cache, we only report miss rates for the cases with a 16 byte line size (this is different from the Futurebus standard of 64 bytes). To reduce the effects of cold start misses we report the miss rate over the last 30% of the synthesized address trace. The trace stems from a mix of UNIX jobs (see section 3.1) and contains 46M references to the primary cache, which has a miss rate of 3%.

The results in table 1 show that the miss rates of our simulation model are close to the figures reported in [Bugge90]. The simulated figures of the 8 Mbyte secondary cache are inaccurate because the cache did not reach a steady state before the end of the simulation run[2]. The 8 Mbyte figures of Bugge are also distorted by cold start effects, as can be seen from the relatively large difference between the worst and best miss rates. Our optimistic 1 Mbyte results are presumably caused by the usage of shared text segments in the UNIX mix. We observed that the miss rates, especially those for large caches, are quite sensitive to the exact configuration of the workload (number of applications, context switch rate). The simulated miss rates of the caches up to 4 Mbyte however, show the same trend as the miss rates of Bugge: the influence of the cache size is larger than the effects of the associativity. This indicates that our stochastical application model has the same long-term locality behaviour.

[2]Longer simulation runs are underway to improve the 8 Mbyte figures.

		[Bugge90]			Simulation
Size	Associativity	Worst	Est	Best	model
1M	2-way	22.6	22.2	22.1	20.9
	4-way	18.6	18.2	18.1	17.9
	8-way	16.4	16.0	15.9	15.5
2M	2-way	11.6	10.7	10.6	12.0
	4-way	9.0	8.1	8.0	9.4
	8-way	8.0	7.1	7.0	9.4
4M	2-way	7.3	5.6	5.5	7.3
	4-way	6.0	4.2	4.2	6.2
	8-way	5.9	4.0	4.0	5.8
8M	2-way	5.0	2.0	1.9	5.9
	4-way	4.6	1.3	1.3	5.4
	8-way	4.4	1.0	1.0	5.3

Table 1: Miss rate of secondary cache (in %); [Bugge90] and model values

4 Simulation results

We have used our simulation model to study the performance effects of different multiprocessor topologies. The experiments include a flat, a two-level, and a a three level deep bus hierarchy with a constant number of processors. Before running the experiments we had to determine some reasonable parameter values for the associativity and line size of the caches. Reasonable values for these parameters are determined in the experiments reported in the next two subsections. First we determine the optimal associativity of the caches with a fixed linesize of 64 bytes. Given this associativity we determine the best linesize.

Throughout the remainder of the paper we use the instruction execution rate, expressed in MIPS, as measure of system performance. All the experiments have used the two benchmarks from section 3.1 that consists of a mix of UNIX jobs and a fine grained parallel program. The simulated processors are single-cycle processors running at 20 MIPS. The parameters of the Futurebuses are set to 110 ns arbitration time and a transfer rate of 700 Mb/s. All experiments run for 500 (simulated) milliseconds.

4.1 Associativity

The associativity in a first-level cache has a known important effect on the miss rate, see for example [Hennessy90]. What associativity should we pick for lower level caches in a hierarchical multiprocessor? A common line of reasoning is that low-level caches should be at least large enough to hold all lines of the caches upward in the hierarchy (inclusion property, [Baer88]). Otherwise the high level caches will have to compete with neighboring caches for space in the low-level cache, which causes the low-level caches to frequently invalidate upward copies to service requests that hit a full set. The performance of high-level caches will decrease because they have to invalidate useful lines, which results in a low hit-rate. In [Baer88] it is proved that to enforce the inclusion property, the associativity

of a low-level cache memory should be at least the sum of the associativity of its upward connected caches. Then one set in the low-level cache can hold all lines in the primary caches which fall into that set. Since the paper does not quantify the performance effects of obeying the inclusion property, we performed an experiment with various associativities around the inclusion value.

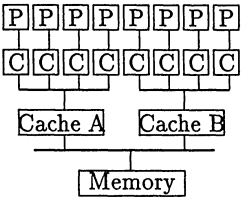


Figure 7: The memory hierarchy used in the associativity and line size experiments

The experiment uses the architecture depicted in figure 7. The first-level caches have a size of 64 Kbyte, and are 2 or 4-way associative. The line size is 64 bytes according to the Futurebus specification. The size of caches A and B is fixed at 2 Mbytes, while the associativity ranges from 4 to 64. The inclusion property requires at least an associativity of 8 respectively 16 for the second-level caches.

second-level	2-way			4-way		
associativity	MIPS	missrate	bus-util	MIPS	missrate	bus-util
4	13.4	3.2%	65%	13.6	3.0%	64%
8	13.3	3.3%	66%	13.8	3.0%	63%
16	13.5	3.2%	65%	13.7	3.0%	64%
32	13.5	3.2%	65%	13.7	3.0%	63%
64	13.5	3.1%	65%	13.7	3.0%	64%

Table 2: The MIPS and miss rates of 2-way and 4-way first-level caches for varying associativity of the second-level cache.

The results of the UNIX mix in table 2 show that the performance is hardly influenced by the associativity of the second level cache. This unexpected result is probably caused by the size of the second-level cache. The large number of sets compared to the primary caches effectively increases the associativity of the second-level cache. Possibly colliding lines between different primary caches usually fall into different second-level sets because of the spatial locality in the applications. Higher associativity does decrease the miss-rate of the second-level cache (as observed in section 3.3), but the overall performance is dictated by the miss rates of the first-level caches. Only architectures with saturated busses will benefit from high associative second-level caches. In the remaining experiments we use the lowest possible associativity that satisfies the inclusion property.

4.2 Line size

In this experiment, the line size is varied to find an "optimal" value. We have used the same architecture as in the previous experiment, drawn in figure 7. The associativity of the first level caches is set to 2, and the second level caches have a fixed associativity of 8. The second- and first-level caches use the same line size, which is varied between 32 and 1024. The results of the two benchmark applications are shown in figure 8.

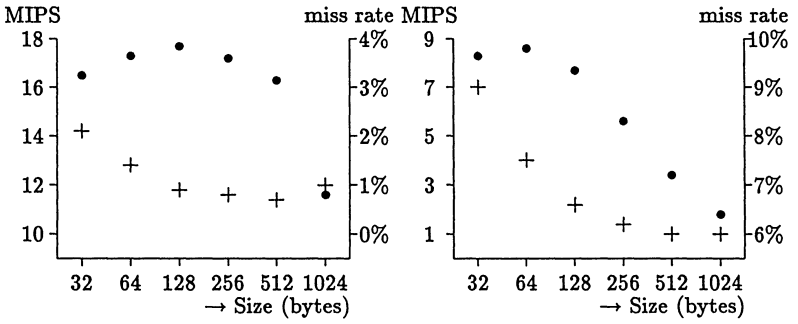


Figure 8: Processing power (dots) and miss rates (+) of various line sizes in the case of a UNIX application (left figure) and a parallel application (right figure).

The data in figure 8 shows that the line size has different effects on the system performance and the second-level cache miss rate. For a UNIX mix of programs, the optimal performance is reached with a 128 bytes line, whereas a 512 bytes line yields the lowest miss rate. Apparently, the decreased miss rate does not outweigh the increased miss penalty. The parallel application results are similar to the UNIX mix, only the optimal performance is reached with a smaller line size of 64 bytes. This optimum depends on the way the shared data is used. As stated in [Eggers89] applications can be specifically coded for a certain line size. The 64 bytes line size of the Futurebus seems a reasonable choice, and we used it in the rest of the experiments.

4.3 Different architectures

This experiment studies the performance differences of comparable multiprocessor systems with different cache hierarchies. We have defined several architectures consisting of 8 processors and a comparable amount of cache memory. These are the architectures labeled a through d in figure 9. The architectures e and f have been added to the experiment when we observed a low bus utilization for the UNIX-workload. The line size is 64 bytes in all architectures. The first-level caches are 2-way associative, while the other caches have the associativity enforced by the inclusion property. For example, the third-level caches in figure 9-d are 8-way associative.

The performance of the multiprocessors is listed in table 3 and shows a large difference between the two benchmark programs. The UNIX-mix performs best on an architecture with a single bus, whereas the parallel application prefers a two-level hierarchy.

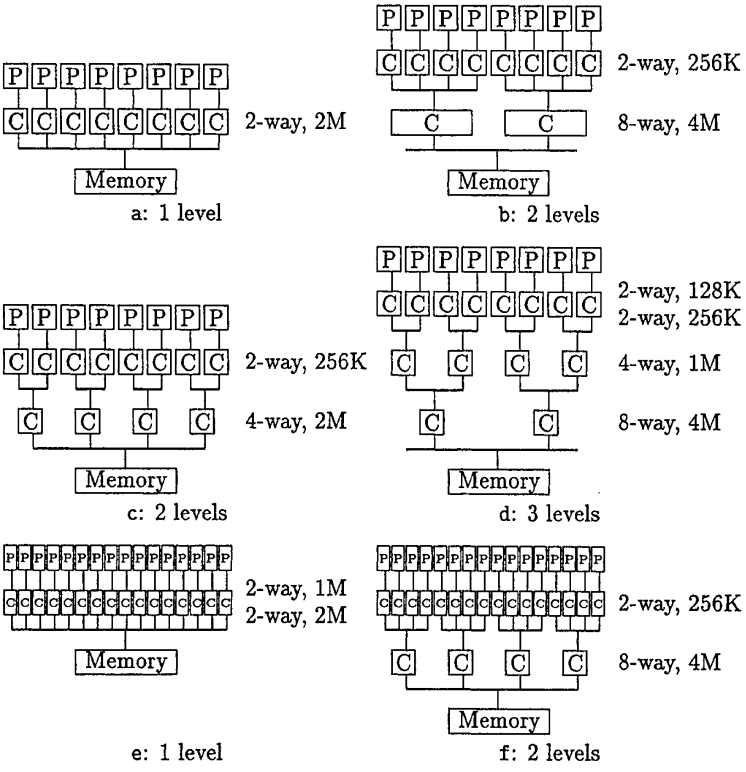


Figure 9: Hierarchical architectures, a-f. The associativity and size of the caches are denoted at the right hand side of the caches. Architectures d and e are evaluated with two different sizes.

The performance of the UNIX-mix largely depends on the effectiveness of the first level cache since jobs are independent and do not share writable data. A cached line of one job will *never* be invalidated by a job running on another processor; the cache line is replaced when the job itself causes a cache miss in a full set. The big caches of architecture-a are well suited for the UNIX-mix and only cause a moderate load on the main-memory bus, therefore the UNIX-mix does not benefit from a hierarchy which potentially decreases the effects of bus contention. In fact the performance degrades because of the smaller primary caches (increased miss rate) combined with a higher miss penalty. The results of doubling the number of processors to 16 are shown in table 4. This number of processors saturates the bus and favors a two-level hierarchy. The memory bus is still the performance bottleneck, larger second-level caches may pay off.

The bus utilization figures show that the parallel application generates more bus traffic than the UNIX-mix to handle the updates of shared data. All hierarchical architectures succeed in diminishing bus contention and substantially increase the overall performance,

	total cache size	UNIX mix					parallel application				
		MIPS	miss-rate	bus util (%) 1^{st}	2^{nd}	3^{rd}	MIPS	miss-rate	bus util (%) 1^{st}	2^{nd}	3^{rd}
a	16Mb	16.4	1.6%	69	-	-	5.0	7.0%	99	-	-
b	10Mb	15.0	2.2%	59	60	-	7.2	7.2%	93	87	-
c	10Mb	14.7	2.2%	32	70	-	6.4	7.2%	52	99	-
d	13Mb	13.5	2.7%	34	50	59	6.6	7.5%	55	83	83
	14Mb	13.8	2.3%	33	50	57	6.7	7.2%	54	83	82

Table 3: Performance of the 8 processor architectures (a-d).

architecture	total		miss	bus utilization	
architecture	cache size	MIPS	rate	^{1}st	2^{nd}
e	16Mb	8.8	1.9%	99%	-
	32Mb	9.5	1.8%	99%	-
f	20Mb	11.9	2.3%	56%	98%

Table 4: Performance of the 16 processor e and f architectures for the UNIX work-load.

up to 44%. Note the remarkable difference between the two-level hierarchies b and c. The configuration with two large second-level caches has a clear advantage over four smaller caches, even the three level deep architecture d outperforms architecture c. This is a consequence of the saturation of the memory bus (99%) in architecture c, which is most likely due to sharing in the low level caches.

4.4 The usage of the transition table

During the simulation runs we did not only measure performance parameters like miss rates and bus utilization, but also the usage of the Futurebus cache coherency protocol. The protocol has been implemented as a state machine, and the measurements show that 90% of the transitions are executed in just 17 states. As to be expected, these states correspond to cache hits on reads/writes and snooping hits. Only 38 states cover 99%, while the remaining 1% requires another 214 states. These 252 states out of a total of 8620 sufficed to run all simulation experiments described in this paper, which issued a total of 3.10^9 instructions. Unfortunately we had already specified over 300 states before trying to run the simulator.

5 Discussion and Conclusions

This paper has reported on the evaluation of hierarchical multiprocessor architectures based on the Futurebus. In section 2 a simulation model was presented to study the performance effects of various system parameters. The experiments include various line sizes, associativities, and hierarchical cache configurations.

The model does not use multiprocessor address traces as a workload, but contains a set of stochastical processes instead. These processes generate memory references based on measured statistics of actual programs. Throughout the experiments we have used a benchmark of two different synthesized workloads: a multiprogrammed set of UNIX applications, and a fine grained parallel program which uses shared data. We have run some simulations to validate the model with published trace-driven results.

The first experiments with the model studied the effects of line size and associativity of caches in a hierarchical multiprocessor architecture. The simulation results show that a line size of 64 bytes, as chosen by the Futurebus, will give good performance for both the UNIX and parallel applications. Higher associativity of (large) caches down in the hierarchy (close to memory) will decrease the low-level cache miss rates which only has an effect on the overall performance when the lower busses are saturated. In general the size of the low-level caches is more important than the associativity.

The benchmark was run on various multiprocessor architectures, see figure 9, to study the performance effects of various hierarchical topologies. It shows that a hierarchy of caches decreases the bus utilization, but raises the miss penalty. The performance of the architecture critically depends on the type of program. For the UNIX applications, which do not share writable data, a flat hierarchy is optimal up to at least 8 processors. A multilevel cache hierarchy is beneficial for a large number of processors, the break-even point lies between 8 and 16. Our fine-grained parallel program benefits from a hierarchy even with a small number of processors. With 8 processors we noted a performance difference between a flat and a two-level hierarchy of 44%. It is difficult to generalize this result because the performance benefits of a cache hierarchy for parallel programs heavily depends on the locality in the application. A hierarchical cache system only succeeds in diminishing bus contention if most data sharing is between adjacent processors in the tree. To better quantify the effects of data sharing on system performance we will improve our synthesized application model to include synchronization patterns, see section 2.1.1.

The usage of the future bus transition table is quite remarkable: 38 states covered 99% of the transactions, 252 covered all transactions. The definition of the bus contains a lot of small optimizations to save some bus cycles. Many of these optimizations were rarely executed, or never at all. We expect that only systems with much rapidly changing shared data will exercise these optimizations. Therefore we question the value of these optimizations since they do increase the complexity without a clear performance benefit.

The simulation tool has shown to be useful in evaluating the performance of hierarchical multiprocessors. Additional work will have to be performed to obtain better understanding about the performance effects of locality in fine grained parallelism programs, and its impact on the stochastical application model.

6 Acknowledgements

We like to thank E. Odijk for his stimulation to start work on the design of the Pearl/Oyster simulation system, which has been developed in the PRISMA project. Marius Schoorel of ACE provided us with the Future-bus problem. Furthermore we thank Rutger Hofman, Pieter Hartel and Wim Vree for reading and commenting on a draft version of this paper.

References

[Archibald86] J. Archibald and J. Baer, "*Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model*", ACM Transactions on Computer Systems, Vol 4, No. 4, November 1986, pp 273-298.

[Baer88] J. Baer and W. Wang, "*On the inclusion properties for multi-level cache hierarchies*", Proceedings of the 15th Annual International Symposium on Computer Architecture, 1988, pp 73-80.

[Bugge90] H.O. Bugge, E.H. Kristiansen and B.O. Bakka, "*Trace-driven simulations for a two-level cache design in open bus systems*", Proceedings of the 17th Annual Int. Symposium on Computer Architecture, 1990, pp 250-259.

[Eggers89] S.J. Eggers and R.H. Katz, "*Evaluating the performance of four snooping cache coherency protocols*", Proceedings of the 16th Annual International Symposium on Computer Architecture, 1989, pp 2-15.

[Futurebus89] Futurebus+, Logical Layer Specifications, Draft 8.1, P896.1 Working Group of the IEEE Computer Society, December 1989.

[Hennessy90] J.L. Hennessy and D.A. Patterson, "*Computer architecture: a quantitative approach*", Morgan Kaufmann Publishers, Palo Alto, California, 1990.

[Hill87] M.D. Hill, "*Aspects of Cache Memory and Instruction Buffer Performance*", Ph.D. Thesis, Univ. of California at Berkeley Computer Science Division, Tech. Rep. UCB/CSD 87/381, November 1987.

[Muller90] H.L. Muller, "*Evaluation of a communication architecture by means of simulation*", Proceedings of the PRISMA workshop on Parallel Database Systems, Noordwijk, The Netherlands, September 24-26, 1990.

[Sweazy86] P. Sweazy and A.J. Smith, "*A class of compatible cache consistency protocols and their support by the IEEE Futurebus*", Proceedings of the 13th Annual International Symposium on Computer Architecture, 1986, pp 414-423.

[Vernon89] M.K. Vernon, R. Jog and G.S. Sohi, "*Performance Analysis of Hierarchical Cache-Consistent Multiprocessors*", Performance evaluation 9, 1989, pp 287-302.

Efficient Global Computations on a Processor Network with Programmable Logic *

Jean Marie Filloque

LIBr-ENST Bretagne, Kernevent-Plouzane, 29285 Brest, FRANCE

Eric Gautrin

IRISA-INRIA, Campus de Beaulieu, 35042 Rennes Cedex, FRANCE

Bernard Pottier

LIBr-UBO, UFR Sciences, av. Le Gorgeu, 29287 Brest, FRANCE

Abstract

A new parallel MIMD architecture is described each node of which is tightly coupled to a global programmable logic layer. This layer gives local acceleration to the node processors by massive micro-grain parallelism. It also provides fast computation services to distributed algorithms by synthesis of global dedicated units operating directly on node operands. As a result, fine approximations of global states become transparently visible in each node, in contrast with usual difficulties and delays in sharing and computing control data.

This point is emphasized by the description of two parallel virtual time mechanisms. The first study involves increasing virtual clocks, and the second one takes into account time counter overflows in a time warp environment. Implementations are based on global systolic networks, fed by the array of local operands and controlled by a small automaton. Thus, global states are handled for each cycle of the mechanism, and results become visible after one pipeline delay with no cost for the accelerated parallel machine.

To summarize general characteristics of this architecture are: general purpose, reconfigurability, cheapness, extensibility.

Introduction

Conventional MIMD architectures are usually built using Von Neumann processors, communication links, or external addressing facilities. Machines belong to the *shared memory class* or the *distributed memory class*, depending on the way the nodes communicate. Algorithms must take care of global computation states for various purposes: termination or deadlock detections, calculation convergences, minimum of local stamps, ...

*This work is supported by *Région Bretagne* and *Municipalité de Brest*. The Armen machine implementation is supported by *ANVAR*.

Shared memory multiprocessors can use shared variables to compute these conditions. Distributed memory computers must implement periodic visits of the whole network to calculate global flags or values. In each case, control operations are merged with computation operations. Periodic accesses to shared variables from the whole network, processor and communication links preemption and network delays raise barriers to the repetition of such operations.

Thus, it appears that MIMD machines cannot execute simultaneously control and computation operations. Moreover, it is evident that the computation bandwidth is dependent on the frequency of control: MIMD machines cannot provide adequate global state visibility.

As an example in the distributed simulation field, the *Time Warp* algorithm requires that the minimum of local *Time Stamps* will be available to garbage memories. Bellenot[2] has reported that such an operation takes about 150ms on a 32 nodes hypercube with the application being stopped.

Beside the strong interest for fast and flexible interconnection networks, a very attractive hardware support in an MIMD machine could be some global unit receiving data from node interfaces and sending computation results back to these nodes. To cover large fields of applications, it is necessary to define such a unit with a technology allowing deep modifications of its behavior. Recent advances in reconfigurable logic technology have given the opportunity to investigate all kinds of global hardware supports to accelerate control and computation in parallel architectures.

This paper presents the architectural concept of *global reconfigurable coprocessors* for MIMD machines. For this purpose, local *reconfigurable logic sockets* are added to each node and connected together to build a *linear logic layer*. The topology of the parallel machine does not need to be specified, but there is a requirement for some stable primary communication services. To get additional hardware support, the operating system must synthesize services into the logic layer. This task is achieved by sending first configuration specifications to each node, and then writing them into the configuration memory of the local socket. Delays for this last task are currently from 0,1s to 1s, and this process can be repeated and interleaved with execution.

The addition of a reconfigurable logic layer to an MIMD machine has two strong advantages with respect to technology and architecture. First, reconfigurable logic *is* an integration technology and allows very efficient circuits to be synthesized and used. Second, the coprocessor has a strategic position in the MIMD machine. It is strongly tied to each node of the machine, but conserves properties of a dedicated centralized functional unit. It can improve intensive computations as a local accelerator, or distributed computations as a global coprocessor.

The objectives of this paper are twofold:

1. the configurable layer use is illustrated and demonstrated by a description of two Global Virtual Time coprocessors for distributed algorithm support. It is shown by these simple examples that inefficient software tasks can be improved in a smart way by the reconfigurable layer.

2. an original algorithm is proposed where a global controller is synthesized to synchronize the nodes periodically. The period is an application dependent tunable constant.

The paper is organized as follows:

- The first part is a short description of architecture principles. The general coprocessor status is emphasized by the notion of dedicated synthesized architecture.
- In the second part we introduce some notions from the distributed simulation field, and describe an implementation of global virtual time computation on the configurable layer.
  - A minimum approach is first investigated without any attempt to manage the time counter overflow. The synthesized service can be used to prevent mutual drift between the logical local clocks.
  - A second approach is virtual time management processes on user-specified time slice boundaries.

Information relative to a practical implementation and the whole project is given, and we conclude by general considerations and fields of application.

1 Accelerated parallel architecture

1.1 Node architecture

The proposed architecture principle involves a general purpose parallel machine with a shared or distributed memory, and a complementary global synthesized coprocessor. Figure 1 shows a node with a processor, local memory and a configurable socket. The socket interconnection has a ring topology.

The socket can be implemented with a large commercial reconfigurable logic array providing at least three data ports to the local system bus and the two adjacent sockets. The local interface of the socket is connected to the processor interrupt and arbitration signals, as well as to local memory control lines. Access to the configuration memory of the socket is mapped into the processor address space, and normal memory processor transactions are passed to the socket logic to be internally interpreted. Therefore the socket can be seen as a second processor rather than a slave unit.

1.2 Synthesized coprocessors properties

Coprocessors can operate on control information, instructions and data[15]. They can be synthesized to yield three distinct classes of computation:

- local coprocessing, examples of which are : instruction set emulations, data intensive algorithms, support for heavily used functions.
- massive parallel computations on the reconfigurable layer which may be used as a large operator controlled and fed by the parallel machine. Two fields of application are systolic signal processing and cellular automata.
- architectural support for global control of the parallel machine. Expected applications are load balancing, fast termination detection, global synchronization and virtual clock support.

The two following paragraphs describe the architecture's design and properties of the reconfigurable logic technology respectively.

Sockets are tightly coupled to node processors and embedded into the reconfigurable layer, thus providing local interfaces to global synthesized operators. Data are directly

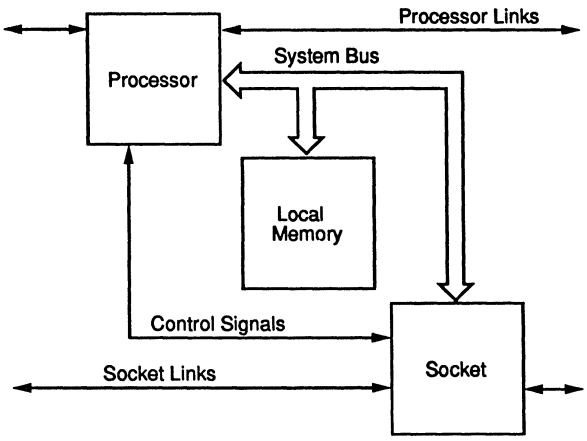


Figure 1: Node Architecture

used as operands for calculations, avoiding the very heavy control process of an MIMD machine, where objects must be carried from node to node to be processed on Von Neumann processors. Therefore there is no longer a bottleneck from bus or network contentions, and there are no prohibitive delays from transport layers. Global intensive computations can be achieved on networks of self-synchronized operators, possibly controlled by a small automaton on a special node. This property applies either to highly regular algorithms or to the control support of irregular applications. This last point has a predecessor in the *Fetch-And-Op* primitive operation of the Ultracomputer [4], where a dedicated network provides a global service with implicit mutual exclusion. Another accurate comparison is the systolic architecture, which is known to minimize regular application control by allowing fast communication between interconnected processors. In a similar way, reconfigurable coprocessors provide tight coupling between nodes and global services, thereby greatly improving control and synchronization for irregular distributed applications.

Von Neumann processors use fixed size general operative units, sequence test and execution, reject constants into data or program space. Accessing data involves the use of memory tables or register files. In contrast to these processors, a synthesized operative unit matches the operand size, implements test and execution in parallel and integrates temporarily stable data into the operators. Memory tables can be mapped into internal trees with very fast access time. As a result, synthesized logic efficiently implements massive micro-grain parallelism. Previous work from various authors has taken advantage of these properties to allow considerable speedups for many applications like image processing, encryption, data compression, long integer arithmetics [1, 8, 14, 18]. The proposed architecture will obviously benefit from the technology, enlarging node fields of application.

The following section shows the need for development activities to build synthesized dedicated architectures on very general purpose hardware.

1.3 Coprocessor development model

Considering a conventional working application we can distinguish three logical components:

3	Application software
2	System support
1	Machine hardware

Each layer in the machine brings services and constraints to the upper layers. As a result application software must deal with the characteristics of underlying components. A common alternative to general purpose machines are specialized ones with the following problems: (i) small market segments involve higher costs, (ii) hardware and software investments are more difficult to preserve. The programmable logic layer architecture introduces an additional flexible component into the usual decomposition, allowing temporarily specialized machines to be synthesized on general hardware :

4	Application software
3	System support
2	Programmable logic layer
1	Machine hardware

Such a machine inherits properties of the conventional initial hardware, because of the transparency of *layer 2*. The behavior of a specialization is similar to the addition of optional arithmetic or dedicated[1] coprocessors to existent machines. It becomes possible to obtain efficient dedicated services by rejecting some difficult points of software implementation into programmable logic. Another point of interest is that new applications are more independent of technology : a specific configuration is defined to match the problem exactly, and the influence of processor integration advance is minimized.

Layer 2 definitions come from creations of *configuration files*. This is currently a CAD activity, similar to peripheral driver writing at operating system level, but there is room to design more dynamic schemes. The next section will show two virtual time services for distributed algorithms. It is envisioned that such services can be part of independent resource libraries to be released for application developments. Speed-ups and additional supports are two benefits from the proposed architecture on the quantitative and qualitative sides.

2 A first example of a global virtual time service

Introduction

Distributed systems with pure message passing communication usually use *logical clocks* to timestamp events and messages used to bring them from one process to another. Lamport has shown in [12] that it is possible to construct a total order over their occurrences by using strictly increasing counters, incremented on each emission and updated at reception. So, throughout the system, reception always occurs *after* emission.

In the distributed simulation domain, the problem of virtual clocks (another name for logical clocks) is a little bit more complicated because these clocks are not completely unrelated. Jefferson [6] has proposed the paradigm of *virtual time* that coordinates execution with an imaginary virtual clock. *Virtual time* represents global information and each

[1] An example is the coprocessor board for parallel simulation proposed in [3]

site can have only an approximation of it. *Virtual time* can be implemented with either a pessimistic or an optimistic approach. With the first, a process on a site can safely increase its local clock only if it is sure that it will receive no message in its past. The respect of this *causality* constraint may lead to deadlock. This approach is presented in [13]. It consists in avoiding or resolving deadlocks. Local virtual clocks, as well as *virtual time* never decrease. The second approach assures only the growth of *virtual time* but not of local clocks. So it allows rollbacks in the past to occur on a site. This is described in [5, 6].

It is, a priori, impossible to have a consistent view of global state and time in such an environment without a shared memory and a common clock. So, processes must content themselves with a best possible approximation of this global information. The construction of a global time approximation is proposed by several authors. It consists in a steady evaluation of a lower bound of all the local clocks in the network. This approximation is used to prevent mutual drift between logical clocks like in [16], to update queues and to avoid memory saturation in time warp systems like in [5, 17], to estimate load ratio of processors for load balancing... This type of computation is suitable for implementation in the programmable layer of the machine, and the following sections describe two applications used to support this assertion.

2.1 A global computation for increasing virtual time

The goal of this section is to emphasize the use of basic mechanisms to build coprocessors. This presentation is driven by the example of a global computation for increasing virtual time. A coprocessor is synthesized to swiftly calculate a lower bound of all *Local Virtual Time* with a circulating token and so, to deliver either this bound (a fine approximation of the global virtual time) or an upper limit for message emission timestamps to each node . This limit may also be evaluated in the programmable layer. It is a simple addition with a constant.

For the sake of simplicity, this first proposal does not manage time counter overflows. The coprocessor must compute a *Global Virtual Time* as the minimum of each node *Local Virtual Time*. For the following, let us define *GVT* to be an evaluated *Global Virtual Time*, and *LVT* to be a node *Local Virtual Time*.

2.1.1 Coprocessor Architecture

To achieve global evaluation, the coprocessor will receive local data, like node LVTs, and send back results, like a GVT. Transparency of coprocessor parallel services is given by *asynchronous channels* with nodes. These channels are implemented with double-register directional mechanisms connecting the coprocessor to a node. The coprocessor periodically reads or writes channels while node processors execute less intensive write or read operations respectively.

An asynchronous channel from coprocessor to node works as follows. The coprocessor is always allowed to write its own register, and the node to read its own register. Data are transferred from a coprocessor register to a node register when the node does not execute a read operation. In our example (see figure 2), the interface consists of two asynchronous channels:

- **LVT**: from the node to the coprocessor;
- **GVT**: from the coprocessor to the node.

To obtain fast computation cycles, the coprocessor has a pipeline topology in which one stage is associated to one socket. Partial results between stages are embodied in

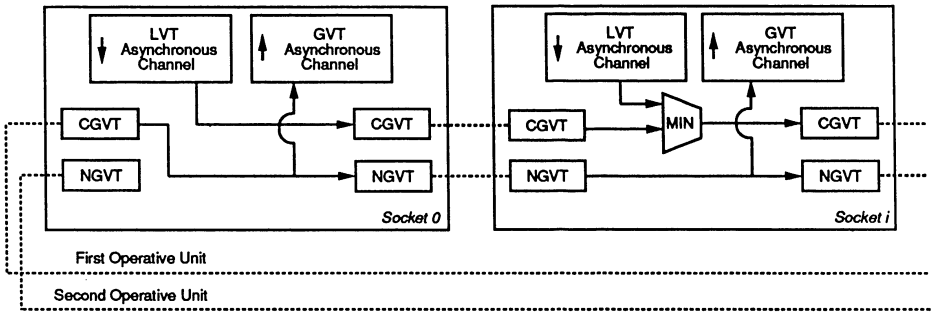


Figure 2: Socket Internal Configuration

so-called *tokens*. Token communications are asynchronous. After completion of its task, a socket writes a modified token to its righthand neighbor.

The coprocessor pipeline architecture is composed of two parts:

- A *large operative unit* distributed across every socket. This unit executes a systolic computation on an array of values from asynchronous channels. Results are fed back to the pipeline head.
- A *control unit* implemented at the pipeline head. This controller is in charge of the initialization of the operative unit, and the token generation. It also receives computation results from the operative part.

In practical implementation, the control unit and the first operative unit stage can be merged on the same socket. Furthermore, the control unit automaton can drive several operative units.

2.1.2 Coprocessor Service

The coprocessor service is defined by two successive global operations:

$$\begin{array}{lll} NewGVT & := & min_{i=1..n}(LVT_i), \quad \text{where } LVT_i \text{ is defined to be LVT of node } i. \\ GVT_i & := & NewGVT \ for \ i = 1..n \quad \text{where } GVT_i \text{ is defined to be GVT of node } i. \end{array}$$

Each operation is implemented by an operative unit. The first one completes the systolic computation of the minimum by passing the $NewGVT$ result to the control unit. The second operative unit broadcasts the $NewGVT$ value to every node.
Partial results of the two operative units are embodied in a single token:

Computed Global Virtual Time (CGVT) : partial $NewGVT$ value.
New Global Virtual Time (NGVT) : broadcasted $NewGVT$ value.

The control unit and the first operative unit stages are merged, as shown in *socket 0* of figure 2. It can be seen that the first operative unit is initialized with the LVT value of its socket asynchronous channel at each pipeline cycle. The control unit also feeds the input of the broadcast unit with the output of the minimum computation.

This figure also shows, in the block at socket i , the parallelism between the local minimum and broadcast operations which overlap with one pipeline latency.

3 A hardware service for Time Warp Simulation

This section gives a brief description of *Time Warp* principles as defined in [5] and shows the interest of knowing *Global Virtual Time.*

In the *Time Warp*, all processes are independent and there is no constraint on their asynchronous evolution. Each message is timestamped with the addressed simulation time, t_r. If the LVT of the receiver is already higher than t_r when reception occurs, then the process must roll back to time t_r, and must undo actions between t_r and LVT. All the messages it has sent must be unsent using *anti-messages.* The roll-back mechanism imposes that processes retain the history of states and lists of all messages sent and received. Maintaining all previous information obviously requires an unbounded amount of memory. It has been proved that there is a lower bound on virtual time which the system will never roll-back to [17]. Knowing this lower bound, it is possible to forget older information. This time is called *Global Virtual Time* and is defined as $GVT = min(LVT_i, t_{ri})$, where t_{ri} is the timestamp of message not yet received. GVT must be computed regularly and generally freezes the simulation progress for *one network diffusion time at least*[17]. Notice that LVT has not the same signification as in the previous section: here, LVT is the minimum of all timestamps of one node.

3.1 Algorithm presentation

To implement the Time Warp Simulation, the nodes need to know the Global Virtual Time. Our previous example presents two restrictions: first, there is no provision for a roll back mechanism; secondly, it does not consider time counter overflows. In this section we present a practical solution taking into account these restrictions.

Instead of computing the GVT, this approach tracks a condition where all nodes have overtaken an LVT bound. When this condition is verified, an approximation of GVT has occurred, and then memory garbage collection is possible. To minimize simulation process freezing on memory saturation, the application must tune the GVT progress intervals to deal with node memory capacities and application characteristics. For the sake of simplicity, the intervals between bounds are equal and the same for all processors.

The configurable coprocessor will compute this condition of a global bound overtake.

3.2 Node Message Passing

On message reception, an advanced process can roll back its LVT to a time less than the next bound to overtake. The computation of the condition must take care of unreceived messages. This problem is solved by message acknowledgment.

Each node is supposed to have its current simulation time, and two queues for input and output messages. The current LVT is deduced from the minimum of all time stamps on the node including messages in the input and output queues [5]. Message deletion from an output queue requires an acknowledgment from the communication service to ensure the visibility of the minimum LVT on the coprocessor.Thus, if the bound is not overtaken, it guarantees that there is at least one node which discards this state.

Message passing from node A to B must respect the following protocol:

Node A sends a message from its output queue;
Node B receives this message then places it in its input queue;
Node B computes its new LVT;
Node B sends an acknowledgment to Node A;
Node A receives the acknowledgment;
Node A deletes the message from its output queue;
Node A computes its new LVT.

3.3 Global Condition Computation

A global computation is an operation on an array of values distributed on every socket. This operation can not be instantaneous because of propagation delays. In the previous proposition, local virtual clocks are strictly increasing. So, GVT is evaluated in a systolic way.

In Time Warp Simulation, the condition of a global bound overtake can be expressed as follows. Let us define GO as the Global Overtake condition, and NB the next bound to overtake.

$$GO = min(LVT_{i\ i=1..N}) > NB$$

This expression could be calculated in a systolic way by a distributed operative unit, where PO is defined as a Partial Overtake :

$$PO_0 = true;$$
$$PO_i = PO_{i-1}\ and\ (LVT_i > NB);$$
$$GO = PO_N;$$

Note that the comparison $(LVT_i > NB)$ can be carried out by the node. Only the boolean result is discarded to the socket through a flag O (*Overtake*). So, the operative unit computes the boolean product of the flags O in a systolic way.

With the roll back effect, local virtual clocks are not strictly increasing. A simple systolic computation can provide an erroneous result. The following sequence on message passing from node i to node j illustrates this problem:

Initial conditions: $i > j$; $LVT_j > NB$; $LVT_i < NB$;

1. Computation of PO_j;
2. Node j receives a message from node i producing a roll back, and decreases LVT_j such $LVT_j < NB$;
3. Node i, receiving the acknowledgment from node j, updates LVT_i such $LVT_i > NB$;
4. Computation of PO_i.

In this example, the operative unit delivers a true GO value to the control unit, but $LVT_j < NB$. Note that at least one of the N following GO will discard a false value.

Property: If N consecutive true GO values are received, the condition of a global bound overtake is true.

Proof: A proof by contradiction can be given. Suppose $W(c)$ is true and $\exists\ O_i(c)$ false with i in $1..N$.

Note $O_i(c)$ the boolean value of flag O of node i at pipeline cycle c.

Then:

$$GO(c) = \bigwedge_{i=1}^{N} O_i(c - N + i - 1)$$

And $W(c)$ which is equal to

$$\bigwedge_{\gamma=1}^{N} GO(c - \gamma)$$

can be rewritten as :

$$W(c) = \bigwedge_{\gamma=1}^{N} \bigwedge_{i=1}^{N} O_i(c - \gamma - N + i - 1)$$

so

$$\neg O_i(c) \Rightarrow \exists \neg O_j(c_\delta) \mid c_\delta \in]c - N, c]$$

which is in contradiction with $W(c)$.

To detect this condition two implementations are proposed: when detecting a first true GO, the control unit can either push a marker CO (*Confirm Overtake*) into the pipeline through an operative unit and wait for its return, or count the pipeline cycles to ensure a total dump. The first solution is chosen for the sake of simplicity.

Systolic arrays cannot exactly implement a computation over the array of operands because of the technological depth limitation of reconfigurable sockets. Each token does not operate on simultaneous sample when circulating in the operator. Therefore it is necessary to observe full pipeline results to get accurate conclusions about what has occurred one pipeline delay before.

3.4 Coprocessor Behavior

The general behavior of the coprocessor can be described in three stages:

1. Tracking N consecutive true GO values;
2. Broadcasting the condition of the bound overtake to every node;
3. Waiting for a global acknowledgment from every node.

To implement the second stage, the control unit can push a marker W (*Wave*) through an operative unit and wait for it to come back.

Assuming that each node acknowledges the coprocessor through a flag A, the third stage can be implemented with an operative unit which calculates the boolean product of all the flags in a systolic way. Note that this computation is carried out in a single systolic pass.

In conclusion, the coprocessor consists of four operative units:

PO : computes the boolean product of flags O in a systolic way.
CO : pushes the marker CO through the pipeline.
W : broadcasts the condition of the global bound overtake.
PA : computes the boolean product of flags A in a systolic way.

and a control unit feeding the pipeline with tokens. Figure 3 illustrates the control unit automaton. The transition conditions are flag values from input tokens.

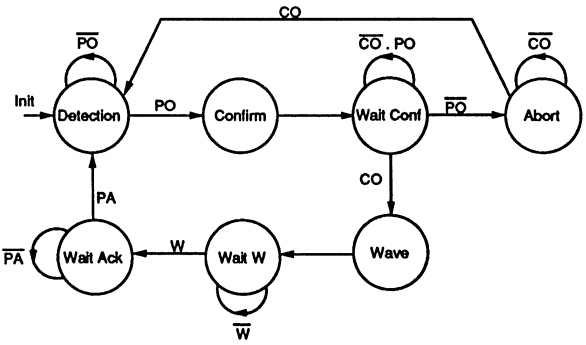


Figure 3: Node 0 Automaton

The token structure is:

PO : the partial boolean product of previous node flags O.
CO : a boolean marker to indicate the *Confirm* condition.
W : a boolean marker to indicate the *Wave* condition.
PA : the partial boolean product of previous node flags A.

The control unit can be implemented in the socket 0. The values for the output token of the control unit are deduced from the automata state:

Detection : (PO = node.O, CO = false, W = false, PA = node.A)
Confirm : (PO = node.O, CO = true, W = false, PA = node.A)
Wait Conf : (PO = node.O, CO = false, W = false, PA = node.A)
Abort : (PO = node.O, CO = false, W = false, PA = node.A)
Wave : (PO = node.O, CO = false, W = true, PA = node.A)
Wait W : (PO = node.O, CO = false, W = false, PA = node.A)
Wait Ack : (PO = node.O, CO = false, W = false, PA = node.A)

The operative units are distributed on each socket executing the following operations on the tokens:

tokenOut.PO = tokenIn.PO *and* flag O
tokenOut.CO = tokenIn.CO
tokenOut.W = tokenIn.W
tokenOut.PA = tokenIn.PA *and* flag A
if tokenIn.W *then* reset flag A

This second example illustrates the use of the coprocessor to compute global conditions in a systolic way by a simple pass, or a pipeline dump. Moreover, the coprocessor controls and sequences actions over the whole network, like broadcasting a condition to every node. Implementation obviously requires very few logic resources, giving way to additive functionalities.

4 Further work : the Armen project

An implementation of the reconfigurable logic layer parallel architecture is currently being built by the LIBr[2]. An MIMD experimental machine called *ArMen* has been designed to investigate most of the capabilities of the architecture. An INMOS T800 has been chosen as the processor node and a Xilinx 3090 LCA [19] as the reconfigurable socket. This leads to small and affordable modules where the socket can operate on addresses, instructions and data from the 32-bit processor multiplexed bus. This is not the most powerful design one could create today, but it is sufficient as a test vehicle. On the other hand, no commercial processor exactly matches our requirements, and there remains a real problem in that we cannot experiment on processor to processor exchanges with the first machine.

Another goal of the ArMen project is to build a software environment for the architecture. Applications can be either specific or general. In the first class, signal processing with use of generic tools, fixed global services like virtual clock support and cellular automaton are considered. Support for these applications can be currently designed as parallel programs and configuration file libraries. The second class of applications is a challenge involving the production of High Level Language development tools for coprocessor synthesis. Given a coprocessor model involving a *node 0* automaton, regular pipelined operative parts and standard interfaces into the node, it is expected that coprocessor generations could be considerably facilitated.

We have shown that the proposed architecture with its accelerated network layer is able to compute global information all over the system with low time cost[3]. It is of obvious interest for the efficient implementation of many applications requiring multiprocessor computation like large logic simulations, signal or image processing, etc...

More generally, one can take advantage of the active communication layer to implement every algorithm requiring global knowledge computation. This can be done very simply on the hypothesis of always empty communication channels (of the application layer). Another use of the active layer can be found in the out-of-band communication between sites. Urgent messages can be routed via this layer by a token containing data and destination. The token can be used either for point-to-point communication, partial diffusion (with an associated list) or complete diffusion.

These points are currently being studied in the framework of distributed discrete event simulators.

5 Conclusion

The proposed architecture complements current parallel designs on many levels.

A first important property of configurable logic is its ability to synthesize small dataflow sequenced operators, and thus to increase the level of parallelism within the nodes.

The connection of adjacent logic arrays provides a global programmable logic resource, on which very large operative parts with arrays of input/output ports are implemented. These ports handle the whole state of the machine repeatedly by feeding systolic arrays with it. We have shown some internal points of the coprocessors, with *global operators* which are small automata controlling systolic linear parts, as well as asynchronous channels and interrupt waves to interfere with node behavior. These tools are useful in computing global resources, or controlling the whole network behavior.

[2]Laboratoire d'Informatique de Brest is a common structure to *Université de Bretagne Occidentale* and *Ecole Nationale Supérieure des Télécommunications de Bretagne*

[3]pipeline delays are in the order of 50ns

Accessing global conditions over a distributed system has often been considered to require heavy local computation and communication or synchronization tasks. Pure distributed implementations can fail[7] because of the inefficiency of these mechanisms: communications and local computations are involved in calculating results which must be dispatched back to the nodes. It is expected that the logic layer architecture will encourage the use of efficient global services within MIMD machines for distributed systems, languages or algorithms.

References

[1] P.Bertin, D.Roncin, J.Vuillemin, "Introduction to programmable active memories", in *Systolic Array Processors*, Prentice Hall, pp. 301, 1989.

[2] S. Bellenot, "Global Virtual Time Algorithms", in *Proc. of the SCS multiconference on Distributed Simulation*, San Diego, Californie, pp. 122, 19 January 1990.

[3] C.Buzzell, M.J.Robb, R.Fujimoto, "Modular VME rollback approach for Time Warp", in *Proc. of the SCS multiconference on Distributed Simulation*, San Diego, Californie, pp. 153, 19 January 1990.

[4] A.Gottlieb and al., "The NYU Ultracomputer — Designing an MIMD shared memory parallel computer", in *Proc. International Conference on Computer Architecture*, ACM, pp. 175. 1982.

[5] D.Jefferson, H.Sowizral, "Fast concurrent simulation using the time warp mechanism", in *Proc. of the SCS Conference on Distributed Simulation*, San Diego, pp. 63-69, Jan. 1985.

[6] D.Jefferson, "Virtual Time", *Transactions on Programming, Languages and Systems*, ACM, vol. 7, no. 3, pp. 404, 1985.

[7] R.Fujimoto, J-J.Tsai, G.Gopalakrishnan, "Design and performance of special purpose hardware for Time Warp", in *Proc. of International Symposium on Computer Architecture*, IEEE, pp. 401, 1988.

[8] T.Kean, J.Gray, "Configurable hardware: two case studies of micro-grain calculation", in *Systolic Array Processors*, Prentice Hall, pp. 310, 1989.

[9] B.W. Lampson, K.A.Pier, "A Processor for a High-Performance Personal Computer", in *Proc. of Computer Architecture Symposium*, IEEE-ACM, pp. 146, 1980.

[10] H.T.Kung, "Why systolic architectures?", *IEEE Computer*, vol. 15, no. 1, pp. 37, 1982.

[11] H.T.Kung, "Network-based multicomputers : redefining high performance computing in the 1990s", in *Proc. of the Decennial Caltech Conference on VLSI*, The MIT Press, pp. 49, 1989.

[12] L.Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol. 21, no. 7, pp.558, 1978.

[13] J.Misra, "Distributed Discrete Event Simulation", *Computing Surveys*, vol. 18, no. 1, pp. 39, March 1986.

[14] B.Pottier, D.Lavenier, "High rate sigma filtering, feasibility studies on processor networks", in *Proc. of IFIP Workshop "Parallel architectures on Silicon"*, INP Grenoble, pp. 182, 1989.

[15] B.Pottier, "Machines parallèles à accélérateurs reconfigurables", *Thèse de l'Université de Rennes 1*, Dec. 1990.

[16] M. Raynal, A distributed algorithm to prevent mutual drift between N logical clocks, *Information Processing Letters*, vol. 24, no. 3, pp. 199-202, Feb 1987.

[17] B. Samadi, "Distributed Simulation, Algorithms and Performance Analysis", PhD. Num 8513157, University of California, Los Angeles, pp. 35-64, 1985.

[18] J.Viitanen, T.Kean, "Image pattern recognition using configurable Logic Cell Arrays", in *Proc. of Computer Graphic International '89*, Springer-Verlag, pp. 355, 1989.

[19] Xilinx, The Programmable Gate Array Data Book, ***Xilinx***, San Jose, 1990.

POMP* or How to design a massively parallel machine with small developments

Philippe Hoogvorst Ronan Keryell Philippe Matherat
Nicolas Paris

—

Laboratoire d'Informatique de l'Ecole Normale Supérieure
URA 1327 CNRS
45 rue d'Ulm, 75005 PARIS
Tel: (+ 33 1) 43.26.58.85, fax: (+ 33 1) 46.34.05.31.
E.mail: ...@dmi.ens.fr
...@frulm63.bitnet

—

Abstract

The design of a SIMD machine is usually complex because it leads to developping an efficient Processing Element and to writing all the softwares required by the chip and the control of the machine. We propose a different approach by using an efficient 32-bit off-the-shelf processor with its software environment (compiler and assembler) and a programmable gate array for the network. It limits the development to the minimum and leads to a rather general SIMD cluster built with off-the-shelf chips which can be considered as a SIMD transputer.

1 Motivation

In this article we propose a methodology for the development of a SIMD machine. The philosophy of the development consists in minimizing the development effort. The excessive complexity of parallel machines is probably the major cause of failure in academic projects. The first quality of a machine is its existence at the end of a project. In this article, we show that it is possible to develop a coarse-grain SIMD machine that offers good performance with very little effort on both hardware and software aspects.

Even if the specificity of this article is to show development reduction, it is important to explain why we have decided to develop this kind of machine. Our main field of interest is image synthesis.

**un Petit Ordinateur Massivement Parallèle*: a small massively parallel computer. Project supported by the French Ministry of Research and Technology, in collaboration with Thomson Digital Image.

2 Why choose SIMD for image synthesis?

Commercial machines for image synthesis are often very specialized with dedicated hardware to speed up the computation of a single algorithm [JHH80,AJ88]. This specialization is the major drawback of this approach. Machines become rapidly obsolete because new rendering algorithms require ever hardware. Only large companies are able to invest large amounts of money and man-power to develop custom machines that will be obsolete in a few months' time. We propose POMP as a non-specialized architecture (with no hardware dedicated to any special algorithm), which is a step beyond the other alternatives (partially non-specialized) proposed in [KV90,FPE*89].

We have to balance the loss of power due to this non-specialization by a massively parallel approach (up to 256 32-bit processors, in fact 8,192 bits of data-paths). This massively parallel organization prohibits the organization in a multiprocessor with shared memory. Each processor has its own local memory and an interconnection network enables data interchange.

This class of architecture contains 2 major subclasses:

- The MIMD machines. Each processor runs its own program on its own data.
- The SIMD machines. Each processor executes the same instruction at the same time on its own data. We do not need a program memory for each processor.

In the graphic pipeline, the last stage is rasterization, which requires most of the computation. A SIMD structure offers the best performance on these computation. [FP81] introduces the concept of *smart memory* which are a set of SIMD memory-PE[1] clusters.

The POMP project tries to generalize this *smart memory* concept to all the algorithms of the whole pipeline. We need for each cluster a general purpose PE, which is able to handle 32-bit integer numbers, floating point numbers, pointer data types, etc.

We also prefer the SIMD structure because a lot of synchronization problems are avoided [BCJ89] and a high MFlop/dm^3 ratio can be reached [BDW85]. Furthermore we can build a very simple programming model which enables to develop debugging environments.

3 The basis of the machine: the programming model

The efficiency and the programmability are the final targets of computer designing. The relationship between hardware and software is the main problem. Most of our choices for the architecture of POMP are consequences of the programming model.

3.1 The programming model

Variables belong to two classes:

- scalar variables (for standard calculation and flow control),

[1] Processing Element.

- parallel variables (also called vectors).

An n-PE SIMD machine is able to simultaneously perform the same operation on a vector of size n. Some SIMD programming environments try to hide the number of processors behind the concept of virtual processors (for instance the Connection Machine). The size of massively parallel variables is assumed to be larger than the number of PEs. Each physical processor emulates one or more virtual processors (*vp*). Vectors are not broken into individual elements but into smaller arrays equally distributed over the PEs.

In a typical massively parallel application, vectors of different sizes are required and need to interact. The vectors must be partitioned into classes called *vp-set* for the CM and *collection* for POMPC. Each *collection* corresponds to one set of virtual processors.

The size is the first attribute shared by the vectors of a same *collection*. The other attributes of the *collection* are:

- the activity. This vector of boolean elements (also called *context*) is the mask which indicates which elements of the vectors of the collection are active.
- the topologic organization. These information describes the topologic relative organization of the virtual processors and the mapping of these virtual processors on the PEs.

3.2 The POMPC language

A detailed description of this language can be found in [Par90]. This model has led to designing of a programming language which is called POMPC. POMPC must be considered as a symbolic macroassembler for SIMD machine as is C for general computers. This language is the direct translation of a programming model and emphasizes the SIMD aspect of the machine: we do not provide an autovectorizing language which hides the structure of the machine from the programmer. This kind of higher level languages can be implemented over POMPC.

Most of the SIMD machines provide this kind of basic language and a taxonomy of many SIMD languages and machines can be found in [Tuc90]. POMPC has been inspired by the previous version of C* [Tmc87, pages 35–41] and is rather similar to the new version of this language. MPL [Chr90] and MultiC [Mlc90] are also alike (without the collection mechanism).

To implement this model, we must define at any time what the different processors (the scalar one and the different virtual SIMD machines) are doing. As only the scalar processor has the control over the program flow and as the PEs are slaves, the best way to express this dependency is to include the instruction for the PE into the sequential program of the scalar processor (it leads to the definition of a very simple and very convenient controller explained in the next section).

As we expect to write the addition of 2 vectors like the addition of 2 scalars, the major problem is to determine from the source file the location of each calculation. Typechecking on expressions and statements provides these informations.

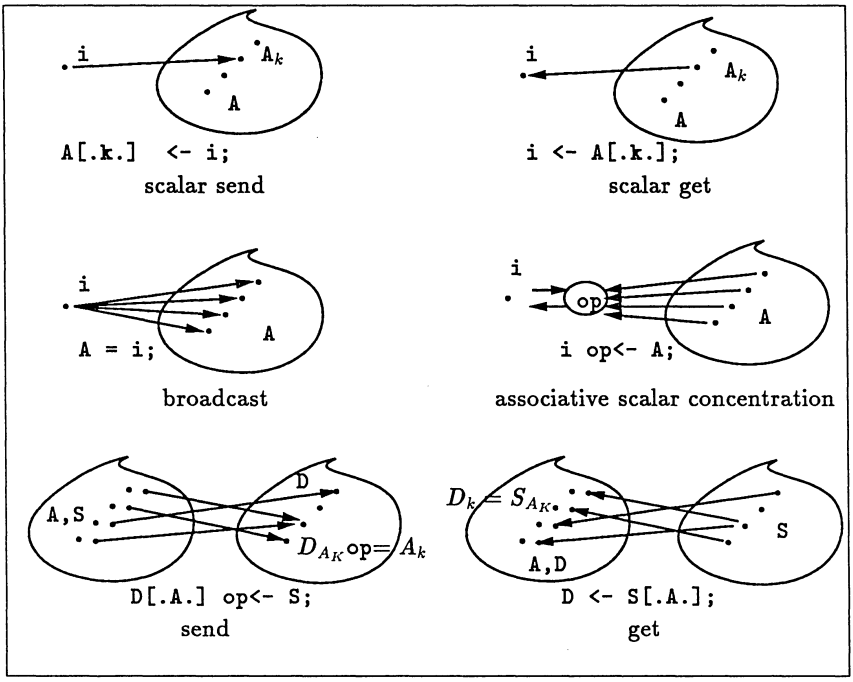


Figure 1: The communications used by POMPC.

POMPC is an extension of the Kernighan & Ritchie C [KR78]. The extensions are as follows:

- It is possible to define collections.
- Each variable can be either a scalar (like in C), a vector of the particular `processor` collection (one datum per PE) or a vector belonging to another collection. Each vector is declared as a member of a collection. Thus it is possible to associate a collection with each vectorial statement or each vectorial expression.
- The `where/elsewhere` operators allow to change the activity of a collection, during the execution of a block. This activity is modified according to the value of a boolean vector of the collection. The `where` statement is the equivalent of the `if` statement except that the block is always executed (it mays contains scalar statements or statements concerning other collections). Every other flow-control statements (even `break`, `continue` and `return` but not `goto`) has been translated for a parallel usage.
- Communications are required to perform non-local interactions. Most of the communications are expressed in the syntax of the language because they require only

```
/*************************************************************************
  Mapping a picture on a scrambled surface :
      The view of an underwater chessboard under a dripping tap
*************************************************************************/
#include "pompc.h"                      /* pompc standard include file */
#include "pc_math.h"                    /* pompc math include file */
collection [256,256] pixel;             /* pixel is a 2D 256 x 256 collection */

pixel chessboard()                      /* returns a chessboard picture */
{                                                                       10
    pixel x,y;                          /* x,y : the local coordinates */
    x = pc_coord(0);y = pc_coord(1);
    where((x & 16) ^ (y & 16)) return 255;
    elsewhere return 0;
}

main()
{
    pixel char color,picture;           /* two pictures */
    pixel int x0,y0;                    /* local coordinates */             20
    pixel int u,v;                      /* mapping coordinates */
    int time,screen;                    /* current time and screen number */

    screen = gr_open_graphic();         /* gets a window where to display the movie */
    gr_set_cmap(screen,0,0,0,-1);       /*      sets a standard color map table... */
    color = chessboard();               /* gets the picture of a chessboard */
    x0 = pc_coord(0) - 128;             /* x0,y0 : coordinate system from */
    y0 = pc_coord(1) - 128;             /*       the center of the chessboard */
    for(time=0;;time++) {               /* and let's go forever... */
    {                                                                   30
        pixel double X,Y,d,d1,phi;
        X = x0;Y = y0;
        d = pc_sqrt(X*X+Y*Y);           /* d : distance from the origin */
        phi = time - d/16.0;            /* phi : phase delay */
        where(phi < 0) phi = 0;         /* drop touches the surface at phi=0 */
        d1 = 1 + 8*pc_sin(phi)/d;       /* d1 : new distance from center */
        u = X * d1 + 128;               /* coordinates where to get the */
        v = Y * d1 + 128;               /*      color of the local pixel */
    }
    picture <- [u,v]color;              /* global indirection */            40
    gr_flash(screen,&picture,0,0,1,1);  /* displays the result */
    }
}
```

Table 1: Example of a POMPC program.

standard network specificities, the rest being carried out by library functions.

Figure 1 summarizes the different syntactical constructions for POMPC communications. The first 4 types of communications are interactions between scalar variables and vectorials ones. The 2 last types are interactions between collections. The `[. .]` operator specifies transformations on the rank of the elements. When a communication may send more than one datum on a given element, a accumulative operator can be specified to accumulate different data in the resulting element. Accumulative operators are addition, subtraction, multiplication, bit-wise and, bit-wise or, exclusive-or, minimum and maximum.

Table 1 shows an example of the POMPC language.

This program computes a chessboard picture (the `chessboard` function) and distorts it (as the deformation of a water surface under a water drip) according to a mapping achieved by a *get*.

4 Architecture of POMP

4.1 Processor designing: a necessary evil?

The first choice during the design of a SIMD machine is the size of the PEs. In fact, this is the first choice because everyone considers that PEs are necessarily custom-made and that we can freely choose the width of the datapath of the PEs.

For some very special applications (mostly image processing), it is interesting to choose 1-bit PE because of the size of the data (from 1 to 8 bits). In order to be efficient these machines require full-custom processors [Gap84]: classical sequential processors are not adapted for this computation, because of the inadequation of the width of the 32-bit processors to 1-bit and 8-bit data.

In the other fields of application for the SIMD (like ours), the required data sizes are more conventional (`int`, `float`, `double`) [Hor82,AB86] and it seems easier and more efficient to use a powerfull processor than to interface a floating-point coprocessor with 1-bit processor, as in the Connection Machine 2. In this last case, 1-bit PEs are no longer used for scientifical computations...

Unfortunately, no commercial 32-bit SIMD PE exists such as the GAPP [Gap84] for 1-bit SIMD machines or the Transputer (an MIMD PE [Inm89]).

We consider that PEs for non dedicated SIMD machines must have the same qualities as classical processors. An intermediate choice could be to design a rather small PE with all the necessary hardware required to micro-code efficiently the floating-point operations (like the MasPar machine [Bla90a]).

The consequences of this coarse-grain choice are important because this seems to suppose the development of a very complex PE. We need very broad competences to be able to design competitive 32-bit PE with floating-point and only large semiconductors companies can cope with such developments. The problem is not limited to chip design but also to the development of all the software environment. Developing the PE is not the good solution.

Let us summarize the requirements for the PE:

- a lot of MIPS: an efficient integer ALU,

- a lot of MFLOPS: an efficient floating-point ALU,
- indirect access to local memory: a data address generator,
- local flow control: a local enable mechanism,
- communications: an efficient network and a routing mechanism.

In fact it is very similar to a classical processor.

4.2 Why not use an off-the-shelf processor ?

Such an approach had already be done for the PASM computer [SSIK84]. The advantages of using a commercial processor are clear:

- we need not develop a PE, it is cheaper and less time-consuming,
- a C compiler is available for our PE,
- we can benefit from every improvement of the processor (this is very important since the speed of RISC processors is regularly doubled inside a common architecture).
- if we remain rather independent of the processor (particularly in the software domain), it is possible to change processors when it appears that a more suitable architecture has been introduced on the market.

We can drastically limit our developments and provide an easy evolution for our machine. The general concept is to choose the best processor at any given time.

Four sensitive points have to be coped with:

- We have to broadcast an instruction to every processor. This is easier if the chosen PE has a Harvard architecture[2].
- We have to keep every PE synchronous. Each instruction must take the same time independently of the data processed. This is mostly the case in RISC processors (as opposed to microcoded processors), provided that all accesses to the memory last the same time. This prohibits the use of PEs with caches.
- We have to independently freeze every PE to process the `where` statement. It is possible if an "instruction not ready" mechanism is implemented on the code bus. This is the case every RISC processor with off-chip cache.
- We have to provide each PE with an access to the network and some facilities to communicate with the scalar processor. It requires special hardware and which will be discussed in the section 5.

In 1991, there exists one processor presenting the required characteristics: the Motorola 88100 [Mot88]. We chose it as the PE of POMP.

[2] a special input bus for the instructions.

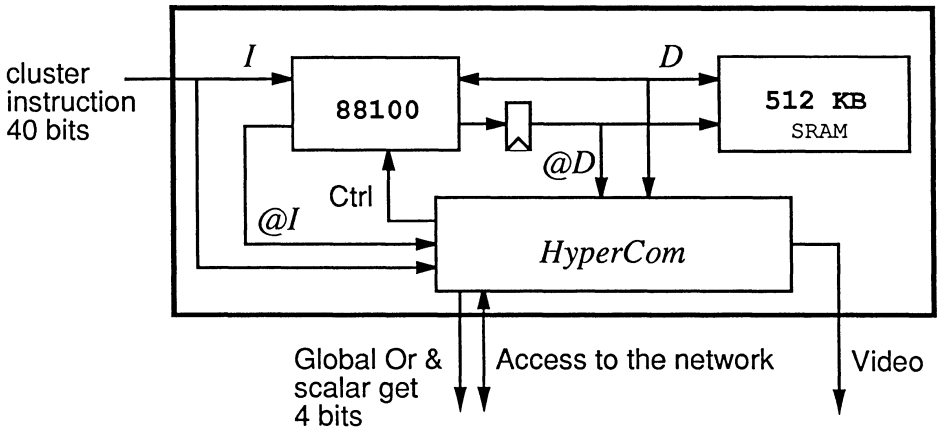


Figure 2: The basic cluster.

4.3 The Processing Element

We can now present the basic cluster for the PE (figure 2). It contains mainly:

- the 88100 at 20MHz (17 MIPS and 7 MFLOPS);
- 128K × 32-bit static RAM with a 35ns access time in 4 chips;
- *Hypercom*, a chip to customize the CPU to its SIMD environment.

The SIMD approach permits us to use only 9 integrated circuits per PE.

A 40-bit instruction bus broadcasts the instructions and the control of the *Hypercom* chip to the cluster. The *Hypercom* provides the following mechanisms:

- the activity management. Depending on the current activity loaded in the *Hypercom*, The control bus defines for each instruction if it is executed. The eventuality of n nested `where` seems to require an n-depth stack to save the current activity. In fact, it can be implemented with a counter [Ker89,Lev90] which is convenient to cope with the complicated activity handling required when using `break`, `continue`, `return`, `case` and `default`.
- the network access. It consists in limited routing capabilities and shift registers. The section 5 is dedicated to the network and will give some indications on the hardware required in each *Hypercom* chip for the access to the network.
- the hardware required for the communications between the PEs and the scalar processor: a 4-bit open-collector bus to get the *global or* of a distributed variable. This also allows to send a vector element to the scalar processor (useful for the

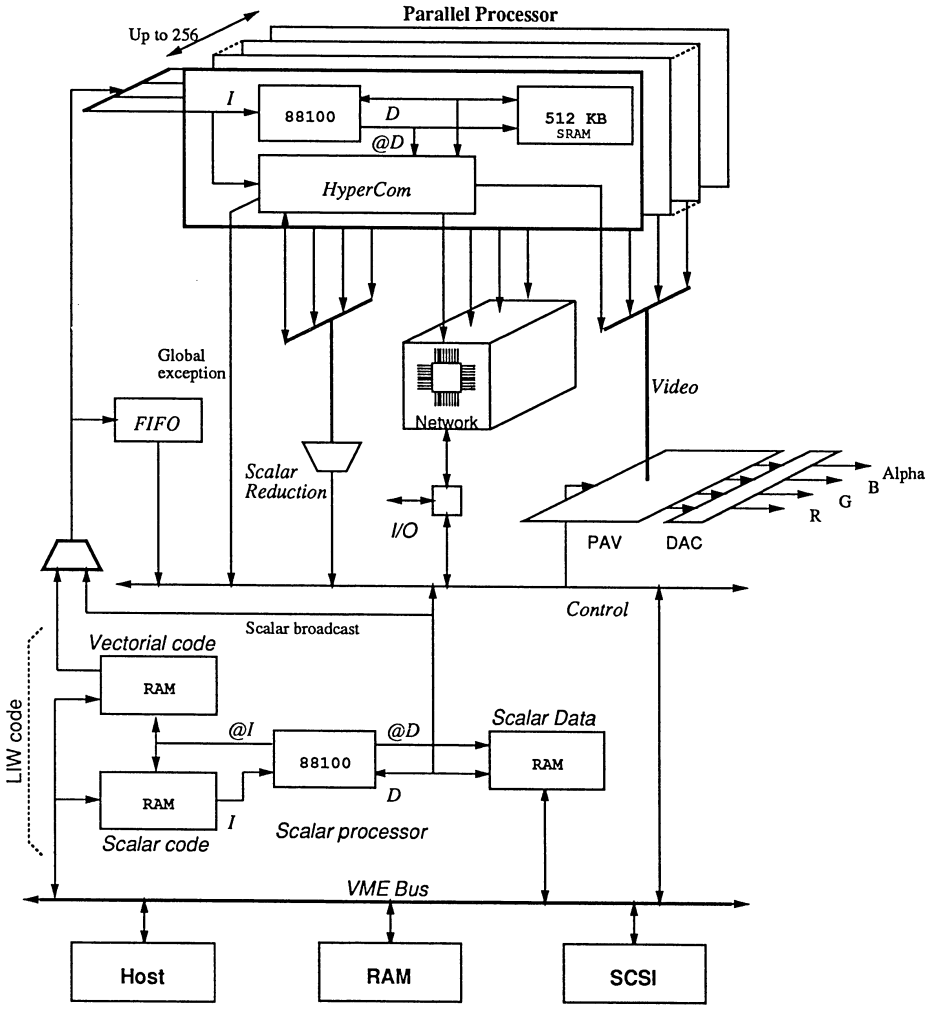


Figure 3: The global architecture of POMP.

scalar get and for the final stage of the associative scalar concentration) by nibbles of 4 bits as in [Bla90a], which is a good compromise.

- the hardware required to correctly recover from an exception or an interruption.

4.4 The controller and the scalar processor

Some SIMD machines use an independent sequencer to run scalar code or to expand microinstructions generated by an host computer [Tmc87]. The use of the host computer as scalar processor facilitates the software development but requires high input/output bandwidth for the broadcast of the code from the host to the PEs. It is possible only if an intermediate sequencer expands some high-level instructions into microcode (typically 32-bit instructions expanded into 1-bit microinstructions, when 1-bit PEs are used). We cannot use such a structure because we need a 20 MHz 40-bit instruction rate. The scalar processor must be directly located in the SIMD machine. This choice has been made in [Bla90a,AB86].

As we use a commercial processor for the PE, it is natural to use the same processor as scalar processor for code orthogonality and for easier synchronization between PEs and the scalar processor. It simultaneously fetches its own 32-bit instruction and the 40-bit instruction broadcasted to the clusters. The whole machine is driven by a 72-bit Long Instruction Word (*LIW*). The figure 3 presents the global architecture of POMP.

An history of the cluster instructions is saved in a FIFO: when an exception occurs, the PEs can correctly resume execution. The scalar processor can access a register to override some fields of vectorial instructions, enabling scalar broadcasts of values.

Since the most important argument claimed by the SIMD defenders is the removal of synchronization issues, we think that the implicit LIW is a way to go further in the synchronization of scalar code with the parallel code, allowing a more global code optimization.

5 The interconnection network

Choosing a network consists in choosing the best trade-off between performance and cost for a given class of applications.

5.1 Measuring the performance

Applications may require different communication types:

- random access,
- 1-neighbour access,
- all neighbours according to a multidimensional mesh,

and different object granularities for the network:

- size of the packets (1 bit to 1 Kbit),
- number of physical processors,
- number of virtual processors per physical processor (*vp-ratio*).

To measure the needs of a target application, we have to evaluate the occurrence of every combination above. A unit system is required for such measurements to compare

the performance of the network with the performance of the PEs. We have decided to speak in terms of:

$$\frac{\text{time required for the communication of 32 bits for each virtual processor}}{\text{time required for the addition of 32 bits for each virtual processor}}$$

5.2 Measuring the cost

The global cost of the machine depends on the cost of the PEs and the cost of the network. The latter is not easy to evaluate because it is not a linear function of the performance. This cost grows by step when the implementation must move from one technology to another at different hierarchical levels [FWT82]:

- the number of transistors required for the network by each PE,
- the number of pins required by each PE,
- the density on each motherboard (the number of routing levels on motherboards),
- the number of connections between motherboards.

5.3 Choosing the Network

Many network designs have been described in the literature and can be classified according to some criteria such as operation mode, control strategy, switching method and network topology [Fen91,Gil86,Kot87].

In our case, the network is synchronous (SIMD machine) and the control is distributed (for scalability and simplicity). The choice of the switching method is not obvious:

- packet switching requires local storage and more complex hardware,
- circuit switching needs to establish a connection through several physical links.

The network topology is probably the major issue in parallel computer designing because it depends on the applications and on almost all the machine parameters.

5.4 Implementation of a hybrid interconnection network

We propose a network for applications which require mostly random accesses (required for image synthesis with distributed data-base) but also simultaneous accesses to all neighbours on a multidimensional mesh.

These aspects seem incompatible and would require respectively a dynamic (switched) and a static network. Existing machines demonstrate it:

static network CM-2 [Tmc87], ILLIAC IV [Hor82], MPP [Bat80],

dynamic network PASM [SSIK84], OPSILA [AB86],

static network and dynamic network MasPar [Bla90b].

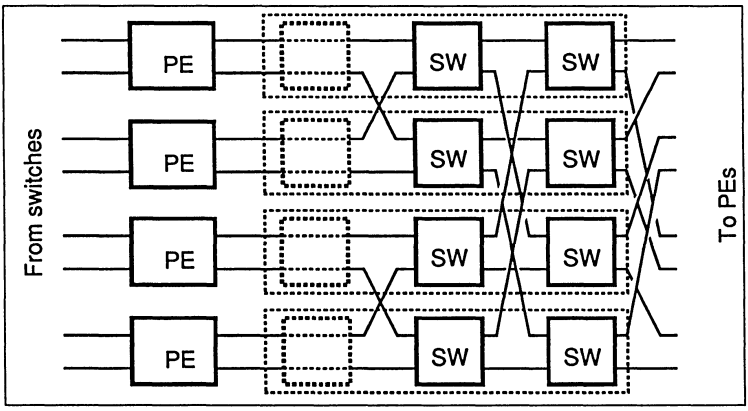


Figure 4: The network for $n = 4$ and $p = 2$.

A candidate for the static network is the hypercube network and a candidate for the dynamic network is the indirect binary cube MIN (multistage interconnection network) [Sch91].

But since a dynamic network is a spatial unfolding of a static network, it must be possible to use the physical wires between switches as a static network instead of using two separated networks like in MasPar.

Each stage of our hybrid MIN can be seen as a dimension of the hypercube. If such a MIN is built with $n \log_p n$ switches instead of $\frac{n}{p} \log_p n$ for a cube MIN with n PEs and $p \times p$-crossbars[3] (ie n lines of $\log_p n$ crossbars instead of $\frac{n}{p}$ lines), it is possible to partition the MIN into n similar subsets mapped on a hypercube, as seen on figure 4 for $n = 4$ and $p = 2$.

The classical design approach leads to the development of an ASIC (Application Specific Integrated Circuit) for the *Hypercom*. This is not convenient because we are quite obliged to redesign the ASIC if the number of processors or the network change.

In order to follow our minimalist philosophy, the *Hypercom* circuit can be implemented with some reprogrammable LCA (Logic Cell Arrays) such as the new 4000 family of Xilinx [Xil90], which offers the required performance, complexity and pin count. Each switch is reversible, offers broadcast capabilities and uses a destination tag algorithm to establish a connection.

For communications on a mesh, a control bit enable changing from the dynamic to the static network. Thus routing overheads are avoided.

5.5 Performance and cost

This study is illustrated for the case of 256 + 1 processors packaged as:

[3]We consider a generalized hypercube pattern with p PEs totally interconnected on each dimension. $p = 2$ for the standard hypercube.

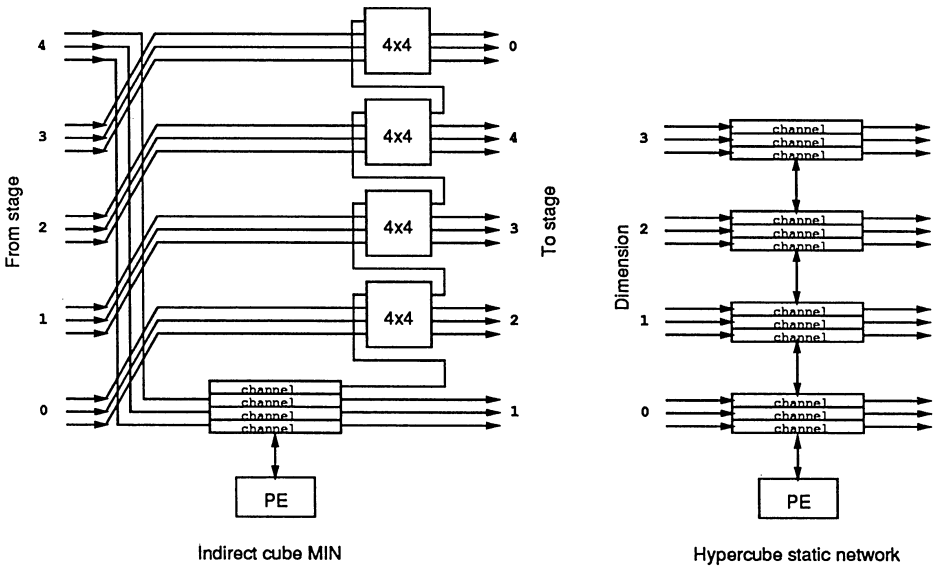


Figure 5: The two configurations of the network with $n = 256$ processors and $p = 4$.

Network type	*Average efficiency*	*Average #cycles per 32 bits*				*Peak throughput*[a]
		`int`	`double`	256 bits	∞	
8 stages, 2×2	0.30	66.8	63.4	60.8	60	1.3 GB/s
4 stages, 4×4	0.37	41.5	38.5	36.2	35.4	2.6 GB/s
2 stages, 16×16	0.48	21.4	15	10.2	8.6	10 GB/s
1 stages, 256×256	0.63	15.8	11.1	7.5	6.3	10 GB/s[b]

[a] For regular routing, like matrix multiplication.
[b] The throughput is limited by the PE data bus.

Table 2: Performance of some hybrid networks.

- 1 controller board,
- 16 motherboards of 16 PEs

in a 19" *Triple Europe* rack.

The performance evaluation of the network for random routing is complex, contrary to neighbourhood communications. We have simulated the random routing for a high *vp_ratio*, as shown in table 2, with 1-bit datapaths. Related costs are represented in table 3.

Network type	*#Links /PE*	#Communication *pins/PE*	#switches */PE*	*#Wires between motherboards*
8 stages, 2×2	2	18	32	158
4 stages, 4×4	4	30	64	282
2 stages, 16×16	16	90	512	960
1 stages, 256×256	256	960	65536	15360

Table 3: Costs of some hybrid networks.

8-stage and 4-stage (figure 5) networks present a correct trade-off between performance for random routing and cost. They are both small enough to be implemented in the *Hypercom* with a reprogrammable LCA, even with 4-bit datapath, for the 8-stage MIN, which is then very performant.

6 The code generation for POMP

Figure 6 illustrates the code generation process for POMP. The final instruction is 72-bit wide and consists in the following fields:

- a 32-bit instruction for the scalar processor;
- a 32-bit instruction broadcasted to the PEs;
- an 8-bit instruction to control the *Hypercoms*.

The global idea of our code generation consists in using commercial compiler and assembler, which is coherent with our philosophy to develop as little software as possible.

The most complex part for this generation is the splitting of the POMPC source file into two C files. This is the first phase of a compiler. It is necessary to develop a parser for POMPC. A typechecker identifies the collection of each expression and each statement. An instruction breaker cuts the different parts of expressions to separate instructions with scalar side effects (scalar assignment, scalar increments and decrements, function calls,...) from purely vectorial instructions which depend on an activity and are repeated as many times as necessary to handle the virtual processing management[4]. Consecutive instructions depending on the same collection are then associated to share the same virtual management loop. Vectorial local variables declared in block handled in a single virtual management loop are relegated to the *processor* collection: local arrays are transformed into single elements which are commonly compiled in registers (this confirms the usefulness of a RISC processor for the PEs). The final step of the program is the code generation. This part is simple because the generated code is C which is a symbolic language. Three kinds of generators are used:

[4]This is the virtual management loop.

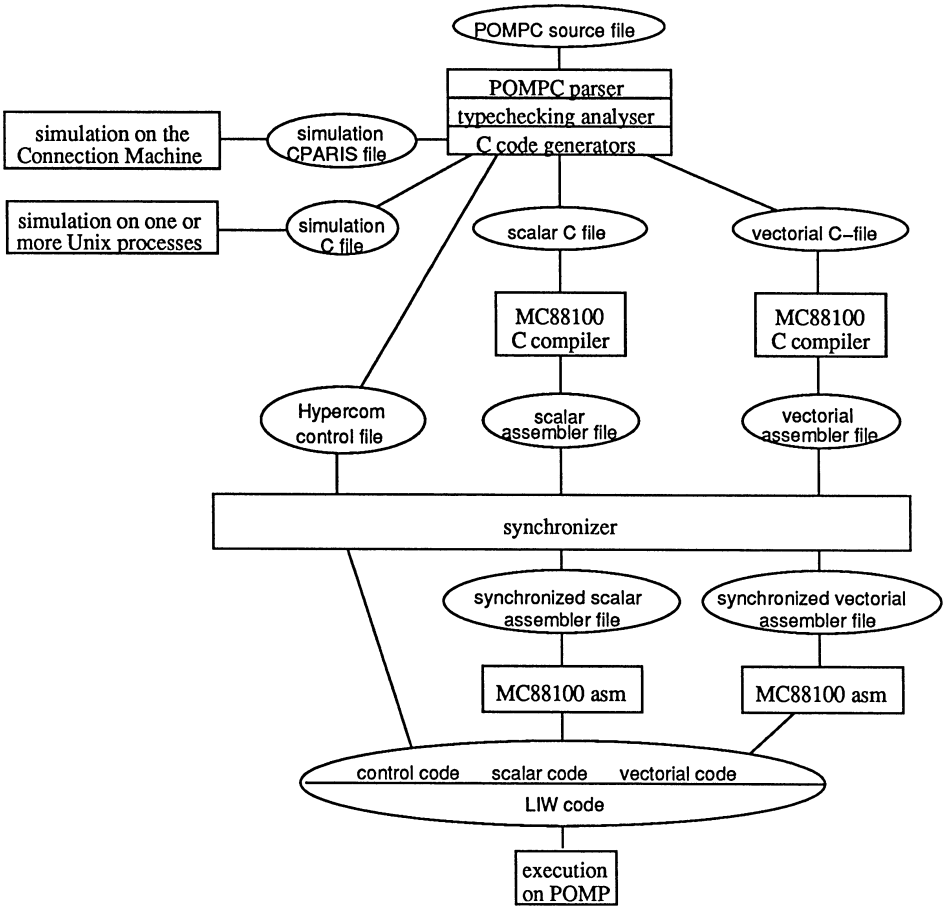


Figure 6: The code generation diagram

- the generation of the code for POMP. The different files are generated with synchronization points declared in the C codes by dummy function calls to pseudo-functions (`synchro_1()`, `synchro_2()`,...)

- the generation of a C file for one or more processes simulation: it is in fact the same generation of code but in a single file. Each PE is simulated by a Unix process which also runs the scalar code. A shared memory segment allows to synchronize the processes and to simulate the communication. Thus it is possible to develop in POMPC the communication routines for POMP and for the simulation. It allows to measure the performance of the network. The number of processes is defined by an environment variable. The monoprocess simulation is a multiprocess

simulation with only one PE.

- the generation of a CPaRIS code (the C Parallel Instruction Set of the Connection Machine [Tmc87]) which allows to perform real-time simulations on the Connection Machine[5].
- other code generators can be thought of; for instance it could be interesting to write some for the Intel Hypercube or for the Sequent Machine and to study the interest of a SPMD language to program MIMD machines.

The second program to be developed is the synchronizer. It takes as inputs the three files for each field and resynchronizes them. This program must understand the assembly code of the target processor (here the MC88100) in order to identify the synchronization pseudo-function calls. Synchronization is achieved by inserting nops in the code to be delayed. This program must also take into account the pipeline structure of the controller and the internal pipeline of the MC88100. The internal scoreboarding of the chip must be managed at compile time to avoid the desynchronization of the whole machine at run time.

The third program is a simple loader with a parallel symbolic debugger.

Only the last two programs depend on the type of the choosen processor and must be rewritten if we choose another processor. This limits the complexity of the development for today and for tomorrow.

7 Conclusion

Choosing to limit the developments does not lead to poor performances.

As concerns the hardware, we have only to develop the controller board, which is easy thanks to the use of a commercial processor, the replicated module of the cluster (a very small board with 9 circuits) and the interconnections of the 16 mother boards.

Software developments are limited to the development of a POMPC preprocessor (20,000 lines of C) and the synchronizer (5,000 lines of C).

It is possible for programs requiring little networking (at most one global indirection every 50 instructions) to reach the full efficiency of the machine: 4000 MIPS and 1700 MFLOPS with a small machine ($\approx$ 1 kW).

8 Current Work

A 3-processor machine is now under development. It will demonstrate the feasibility of the controller and of the programming concepts. As soon as credits can be found (we need 1 MFF in commercial chips) a prototype with 257 processors will be built.

The POMPC compiler is written. Simulations on the Connection Machine 2 (located at the ETCA) and on Unix work. Small applications like a One-Step-Relaxation

[5]This language is used by people programming on the Connection Machine

electrical simulator have been developed in POMPC and run on the Connection Machine and on the Unix simulators. Some aspects of the semantic of POMPC have been studied by Luc Bougé and Jean Luc Levaire [Bou90,Lev90]. A ray-tracer is under development using the spatial coherence of the rays with beam tracing techniques developed in [Thi90].

References

[AB86] M. Auguin and F. Boeri. The OPSILA Computer. In INRIA, editor, *Parallel Algorithms & Architectures*, pages 143–153, North-Holland, 1986.

[AJ88] Kurt Akeley and Tom Jermoluk. High-Performance Polygon Rendering. In *Computer Graphics (SIGGRAPH '88)*, ACM, August 1988. Volume 22, Number 4, pp 239–246.

[Bat80] Kenneth E. Batcher. Architecture of a Massively Parallel Processor. In *SIGARCH 80*, pages 168–173, IEEE, 1980.

[BCJ89] Edward C. Bronson, Thoms L. Casavant, and Leah H. Jamieson. Experimental application-driven architecture analysis of an SIMD/MIMD parallel processing system. In *International Conference on Parallel Processing*, pages 59–67, IEEE, Academic Press, 1989.

[BDW85] John Beetem, Monty Denneau, and Don Weingarten. The GF11 Supercomputer. In *SIGARCH 85*, pages 108–115, IEEE, 1985.

[Bla90a] Tom Blank. The Design of the MasPar MP-1, A Cost-Effective Massively Parallel Computer. In IEEE, editor, *IEEE Compcon Spring 1990*, February 1990.

[Bla90b] Tom Blank. The MasPar MP-1 Architecture. In IEEE, editor, *IEEE Compcon Spring 1990*, February 1990.

[Bou90] Luc Bougé. *On the Semantics of Languages for Massively Parallel SIMD Architecture*. Technical Report LIENS-90-13, Laboratoire d'Informatique de l'Ecole Normale Supérieure, June 1990.

[Chr90] Peter Christy. Software to Support Massively Parallel Computing on the MasPar MP-1. In IEEE, editor, *IEEE Compcon Spring 1990*, February 1990.

[Fen91] Tse Yun Feng. A Survey of Interconnection Networks. *Computer*, 14(12):12–27, December 1991. IEEE.

[FP81] Henry Fuchs and John Poulton. Pixel–Plane: a VLSI–oriented design for a raster graphics engine. *VLSI Design*, 2(3), 1981.

[FPE*89] Henry Fuchs, John Poulton, John Eyle, Trey Greer, Jack Goldfeather, David Ellsworth, Steve Molnar, Greg Turk, Brice Tebbs, and Laura Israel. Pixel–Plane 5: A Heterogeneous Multiprocessor Graphics System Using Processor–Enhanced Memories. In *Computer Graphics (SIGGRAPH '89)*, ACM, July 1989. Volume 23, Number 4, pp 79–88.

[FWT82] Mark A. Franklin, Donald F. Wann, and William J. Thomas. Pin Limitation and Partitionning of VLSI Interconnection Networks. *IEEE Transactions on Computers*, C-31(11):1109–1116, November 1982.

[Gap84] *Geometric arithmetic parallel processor NCR45CG72.* NCR, 1984.

[Gil86] Wolfgang K. Giloi. Interconnection networks for massively parallel computer systems. In *Future Parallel Computers*, pages 321–348, Springer-Verlag, 1986.

[Hor82] R. Michael Hord. *The ILLIAC IV, The First Supercomputer.* Computer Science Press, 1982.

[Inm89] *The Transputer Databook.* INMOS, 1989.

[JHH80] James-H.Clark and Mark-R. Hannah. Distributed processing in a high performance smart image memory. *Lambda*, 1(4), 1980.

[Ker89] Ronan Keryell. *POMP2: D'un Petit Ordinateur Massivement Parallèle.* Rapport de Magistère, LIENS — Ecole Normale Supérieure, octobre 1989.

[Kot87] S. C. Kothari. *Multistage Interconnection Networks fo Multiprocessor Systems*, pages 155–199. Volume 26, Academic Press, 1987.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice-Hall, 1978.

[KV90] David Kirk and Douglas Voorhies. The Rendering Architecture of the DN10000VS. In *Computer Graphics (SIGGRAPH '90)*, ACM, August 1990. Volume 24, Number 4, pp 299–307.

[Lev90] Jean-Luc Levaire. *Deux sémantiques opérationnelles pour POMPC.* Diplôme d'Etude Approfondie, LIENS, Paris, September 1990.

[Mlc90] *The multiC Programming Language: Extending C to Accomodate Data Parallel Processing.* Technical Report, Wavetracer Inc., 1990.

[Mot88] *MC88100 RISC processor user's manual.* MOTOROLA, 1988.

[Par90] Nicolas Paris. *Définition de POMPC (Version 1.5).* Technical Report, LIENS, février 1990.

[Sch91] Isaac D. Scherson. Orthogonal Graphs for the Construction of a Class of Interconnection Networks. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):3–19, January 1991.

[SSIK84] Howard Jay Siegel, Thomas Schwederski, Nathaniel J. Davis IV, and James T. Kuehn. PASM: A Reconfigurable Parallel System For Image Processing. *ACM SIGARCH Newsletter*, 12(4):7–19, September 1984.

[Thi90] Jean-Philippe Thirion. *Interval Arithmetic for High Resolution Ray Tracing.* Technical Report LIENS-90-4, Laboratoire d'Informatique de l'Ecole Normale Supérieure, February 1990.

[Tmc87] *Connection Machine Model CM-2 Technical Summary.* Technical Report HA87-4, Thinking Machine Corporation, April 1987.

[Tuc90] Russ Tuck. *Porta-SIMD: An Optimaly Portable SIMD Programming Language.* PhD thesis, University of North Carolina at Chapel Hill, May 1990.

[Xil90] *XC 4000 Logic Cell™ Array Family.* XILINX, 1990. Technical Data.

The Function Processor: An Architecture for Efficient Execution of Recursive Functions

Jonas Vasell Jesper Vasell
Department of Computer Engineering
Chalmers University of Technology
S–412 96 Göteborg
Sweden

Abstract

The Function Processor is a wavefront array architecture, i.e., a regular structure of locally interconnected processing elements called Function Cells, which operate according to the data flow execution principle. By means of a compilation method developed for this architecture, data flow graphs for functional programs can be created and mapped onto the processor array, so that each Function Cell is assigned the execution of one graph node. The main result presented in this paper is a Function Cell architecture which has been designed to support the functionality required by these data flow graphs. Some implementation results are also presented.

1 Introduction

Due to the rapid development of VLSI technology in recent years, array architectures have become an increasingly interesting alternative for very fast algorithm implementations. An array processor consists of a large number of simple processing elements. Each processing element is directly connected only to a few neighbour cells, and repeatedly performs a single operation on data arriving from these neighbours. Systolic arrays [Kun82, MMU87] is a well known example of these architectures. Another example is wavefront arrays [KAGER82, KLJH87, Kun88, KMPS88]. These architectures differ in the way the processing elements are synchronized. In wavefront arrays, a processing element works in a way similar to the data flow principle [Arv80, Den80, Vee86, McG89], i.e., it performs its operation as soon as all necessary operands have arrived from its neighbours.

Traditionally, array architectures have mostly been used for highly regular computations in signal processing and image analysis. A regular computation is always performed in the same way, independently of the actual input data. This limits the computations that may be performed, but it makes it possible to exploit data parallelism to a large extent. An example of a regular computation is multiplication of fixed-size matrices. Our

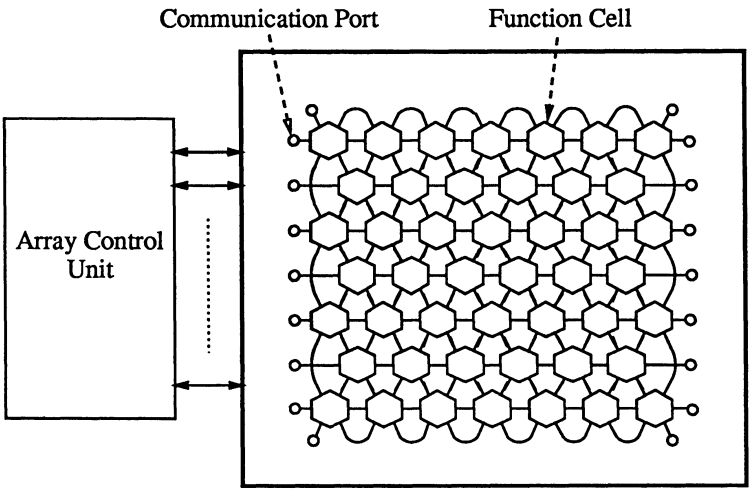


Figure 1: Function Processor Architecture

aim has instead been to try to use array architectures to support irregular computations. It is then more difficult to exploit data parallelism, but much is still to be gained from pipelining parallelism and a very low interpretation and communication overhead. We have also wanted to find support for fast execution of symbolic computations expressed in functional programming languages [Bac78, Hug89, Rea89]. Simulations have shown that typically a speed-up of between five and fifty times can be achieved with wavefront arrays compared to sequential execution on state-of-the-art workstations (see section 5).

We will here present an architecture for a wavefront array processing element suitable for these purposes. These processing elements are called *Function Cells.* The *Function Processor* (see figure 1) is an architecture consisting of an array of Function Cells and an Array Control Unit. Several different types of systems can be built around the Function Processor architecture. For instance, a system can consist of several Function Processors and a conventional processor which performs administrative tasks. Another possibility is to use the Function Processor as an accelerator for a host processor running a conventional implementation of a functional programming language. In this case the Function Processor should execute one or several critical functions in the program, while the rest of the program is executed on the host processor. The Function Processor could also be used in a specialized system, for instance a signal processing system, where it executes a limited number of different algorithms.

It is important that a new architecture is developed together with efficient programming methods. We have therefore developed a method to create data flow graphs, DFG:s, for functional programs, which can be mapped onto the Function Processor. By *mapping* a graph onto an array architecture, we mean assigning one processing element to each graph node in a way that allows intermediate results to be transferred between nodes via the processing element communication links.

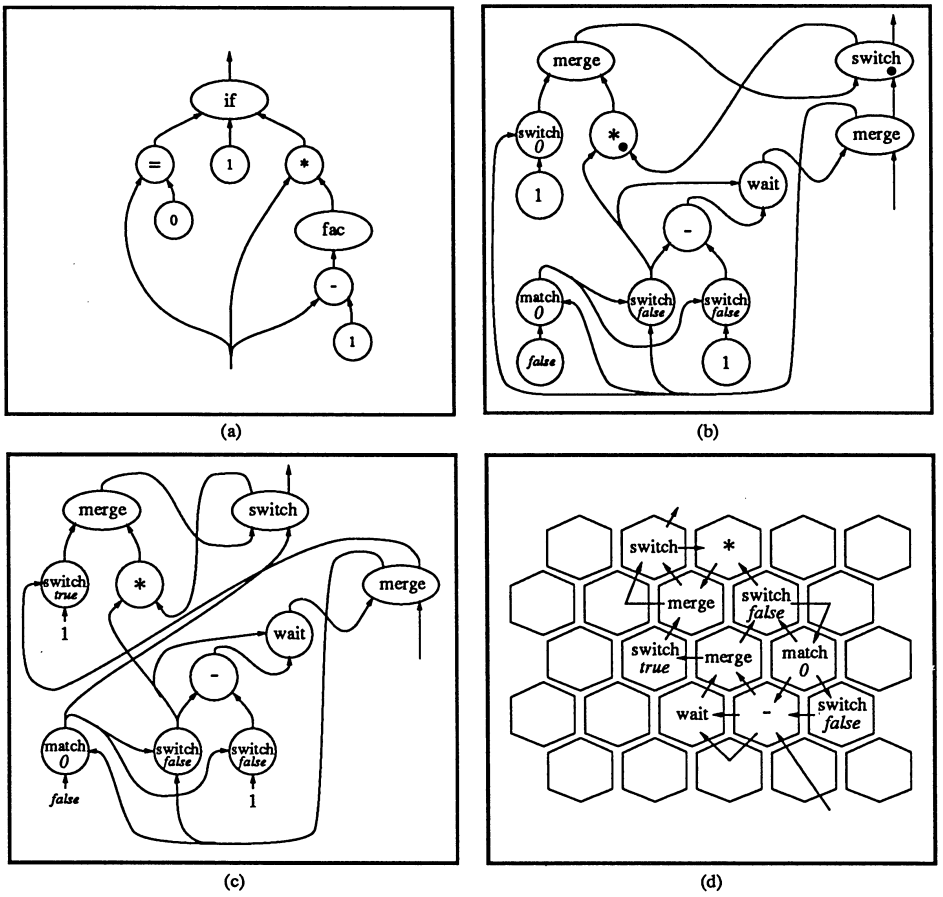


Figure 2: Programming Example

In figure 2, an overview of the whole programming process is shown for a simple example, the factorial function:

```
fac n = if n==0 then 1
        else n*fac (n-1)
```

The programming method takes a functional program, i.e., a set of (possibly mutually) recursive function definitions, as input. From these function definitions a DFG is created. The DFG for the factorial function is shown in figure 2(a).

In general, such a DFG can not be directly mapped onto the processor array. This can be for many reasons, but the most important is that the DFG has to be static, i.e., it must not contain any nodes which represent an expansion of the graph. The reason for this is naturally that the graph can not be allowed to expand during execution since

the hardware configuration is fixed and finite. We have defined four requirements that a DFG have to fulfill before it can be mapped onto the Function Processor. We say that a DFG that meet these requirements is on *hardware implementable form.* Thus, the next step in the programming or compilation process is to transform the original DFG to this form (figure 2(b)).

Usually the execution order imposed by the hardware implementable DFG is not the most efficient, and a number of optimizations are therefore applied on the graph. For the factorial function, one important optimization takes advantage of the fact that multiplication is a commutative and associative operation. Therefore, the order of the successive multiplications can be reversed. This results in a graph which corresponds to a tail-recursive version of the function in which the calculation starts immediately, rather than after all recursive calls have been made and operands have been pushed on a stack (figure 2(c)).

Finally, the optimized DFG is mapped onto the array (figure 2(d)). This sometimes requires that some Function Cells are used only to route data between other cells.

The use of array architectures for irregular functional computations has also been proposed by *I. Koren, G. Silberman et al* (see [KMPS88]). They do, however, not especially address the problems in supporting functional languages and symbolic computations. Instead, they have concentrated on the problem of mapping DFG:s onto wavefront arrays.

M. Sheeran presents a method for synthesis of algorithm-specific array processors from functional specifications in [She85]. This method has a set of predefined higher order operators as its starting point.

2 Hardware Implementable Data Flow Graphs

Traditionally, array architectures have been used for highly regular computations, i.e., computations which proceed in exactly the same way independently of input data. The DFG:s of these computations have a very regular structure and lend themselves very well to execution on an array architecture since they use each processing element in exactly the same way all the time. But if we limit ourselves to such regular computations, we also limit the usefulness of array processors.

If we compile a functional program to a DFG we are, however, not guaranteed to get a DFG which can be used directly as a program for an array processor, since usually the computations are irregular.

We say that a DFG which can be used as a program for an array architecture is *hardware implementable.* The intuitive meaning of this is that it can be seen as a description of a machine performing a specific computation. We have defined the following criteria for a Hardware Implementable DFG (HIDFG):

- *Finiteness*: The number of nodes in the DFG must be finite and statically determined, and must not exceed the number of available processing elements.
- *Realizability*: Each node in the graph must be executable in the resources available in one processing element.
- *Repeatability*: The time from the start of one terminating computation in the DFG until the next computation can be started must be finite. This means that no unnec-

essary subcomputations must be started, unless they are known to be terminating, and do not interfere with any other computation.

- *Representability*: The DFG must only handle data objects which can be represented in a way that the processing elements can interpret.

As can be seen, these criteria are defined relative to properties of a target architecture.

The DFG of the factorial function shown in figure 2(a) contains a node representing a function application. This node causes a problem since we want to statically assign each node, or rather the operation of each node, to a processor. But the application represents a dynamic expansion of the graph, which in this case is further complicated by the fact that it is a recursive application, and there is no possibility to decide how many expansions of the graph are necessary unless we know the value of the argument. This means that the DFG is not finite, since the number of expansions, and thus the number of nodes, can not be statically determined. The only possible solution to this problem is to let all applications of a function be executed in the same instance of the function graph. We could, of course, try to execute the application node in one processing element. This would, however, not be realizable in most architectures, since it would require that a processing element is capable of changing the operations of other processing elements during execution.

Another problem with the DFG in figure 2(a) is the if-node which requires a boolean value, and then chooses to output either the value of the then-branch or the else-branch. Regardless of which branch the if-node eventually will choose, both branches are computed. This means that computations are performed which might not be needed. The problem is that if an unnecessary computation is performed it might give rise to an incorrect result, e.g., if it contains a recursive application, or it might turn out to be a non-terminating computation which blocks any further computations in the graph. It can be seen from this discussion that the presence of an if-operation may give rise to a DFG which is not repeatable, since it requires unnecessary subcomputations to be started, which may lead to non-termination or even an incorrect result. In fact, handling conditionals is a major issue in the design and use of array architectures, since they are the reason why a computation becomes irregular.

The DFG in figure 2(b) solves the problems described above, and is thus hardware implementable. The problem with the recursive function application is solved by allowing all applications of the function to execute in the same graph. In order to make this possible we introduce nodes which keep track of where the results of different applications should be sent. To make this as simple as possible, the order in which different recursive applications are executed is statically determined. Executing multiple function applications in the same graph also creates a need for saving the state of a function application. This is accomplished by introducing stacks in the nodes that contain part of the state when a function application is performed. In the figure, stacks are indicated by a dot at the input of a node.

The solution described above is used for recursive applications, while non-recursive applications are handled by substituting the graph of the applied function for the application node.

Another possible approach to solve the problem with recursive applications would have been a scheme where all data is tagged with instance numbers. This is a method

which has been used in other data flow architectures, such as TTDA [Arv80, AN87] and ETS (Monsoon) [CP90]. However, though this is a very flexible solution which makes synchronization simpler, it requires much more complex hardware to be implemented in the processing elements. This would make it more difficult to build sufficiently large arrays. It would also increase the amount of data communicated between cells. Thus, we have chosen a different approach which we believe is better suited for array processors.

We have developed a method to compile functional programs into HIDFG:s. This method has been implemented in a compiler. Although it is intended to be as general as possible, the method currently places the following restrictions on the source code:

- The method assumes that the source code has been lambda-lifted [Joh85], so that it consists of a set of global recursive function definitions and a result expressions, which contain no local function definitions or lambda abstractions.
- It is also assumed that all pattern matching and case expressions, have been transformed into case-expressions using only simple patterns, i.e., patterns which are variables or constructors applied to variables. A method to do this transformation has been described by Augustsson [Aug85].
- Only a set of predefined types is available, and therefore can no new type declarations be made. The compilation method does not restrict the use of other data types, but the set is restricted by the data type support provided by the target architecture. For the Function Processor this means integers, booleans, pairs, character strings, and lists of these types.
- The methods to handle higher order functions, i.e., functions taking other functions as arguments or producing functions as results, are not yet fully developed. A set of frequently used higher order functions has been made available as predefined functions. These functions include the well-known *map*, *filter*, and *reduce* functions, and the set could easily be extended with many other functions.

Since there exist methods for lamda-lifting and compilation of pattern matching, it is only the restrictions on data types and higher order functions which can be considered to be real restrictions on the source code. The programming method has been described in [Vas90, VV91].

3 Architectural Requirements

The requirements on the Function Cell architecture are to a large extent determined by the types of nodes that may appear in a HIDFG since it must be able to perform the function of any such node.

In general, the nodes have one or two inputs and one or two outputs. They operate according to the data flow execution principle, i.e., as soon as they have received all necessary operands they perform their operation. The following are the node types that are used in mapped HIDFG:s.

Merge. This node has two inputs (left and right), and two outputs (data and selection). When a data object is available at either input it is copied to the data output of

the node. If the object came from the left input, a boolean "true" value is sent out at the selection output, and if it came from the right a boolean "false" is sent out. The selection output is not always used. In those cases, it will be omitted in the figures.

Switch. This node has two inputs (data and select), and two outputs (left and right). A data type constructor is also always specified. The node requires data to be available at both inputs. If the object available at the selection input is built with the specified constructor, the object at the data input is copied to the left output, otherwise it is copied to the right output. Sometimes only one output will be used in the figures. It is then always the left output, i.e., the output used when the selection object matches the specified constructor.

Match. This node has two inputs (match and select) and one output. A data type constructor is also specified. If the object at the select input matches the specified constructor, a boolean "true" is sent out as the result. Otherwise, the object at the match input is copied to the output.

Split. This node has one input, two outputs (first and second), and a specified constructor. All objects arriving at the input must match the specified constructor. The constructors should have two elements. The first element of the input object is copied to the first output, and the second element is copied to the second output.

Wait. This node has two inputs (data and control) and one output. When objects are available at both inputs, the object at the data input is copied to the output. In the graphs, it is always the input in line with the output that is the data input. This node type is special in that it can be specified to be in an initial state where one object already is available at the control input. These nodes are used to synchronize the flow of data at places where the order of computations is significant.

Route. This node can have a variable number of input-output pairs. A data object received at one input is copied to the corresponding output. These nodes appear in the physical mapping of a DFG when two nodes which are connected in the graph are assigned processing elements which are not neighbours in the array, and thus do not have any direct connection.

Constructors and operators. The rest of the nodes belong to this class. These nodes perform some specific operation or constructor function on its inputs, or simply produce a constant object.

Some of these node types may operate on different data types. Therefore the Function Cell must be able to recognize and distinguish the types of the data objects which are communicated in a DFG. More specifically it must be able to recognize the type of a data object received on an input as well as whether all parts of it has been received. As many data types as possible should be supported, but there are some which are especially well suited for support in an array architecture. These are data types which easily can be represented as sequences of bit-groups with a very small overhead for coding and decoding. We wish to support four such data types which will be described in detail later.

They are scalars, pairs, strings and lists. Scalars are single word objects such as integers and characters. A pair is constructed from two scalar values. Both strings and lists are sequences of other data objects.

Furthermore, some of the nodes in a DFG may have a LIFO buffer (a stack) at one of its inputs. Also, the data types which are represented by variable length sequences of data objects sometimes cause a need for FIFO buffering of some node operands.

When creating a physical mapping of a DFG, i.e., assigning a processor to each node and finding a communication path corresponding to each arc, it is necessary to introduce *route*-nodes in order to make it possible to connect nodes which are not neighbours in the array. This can be simplified if it is possible to have a Function Cell perform not only the operation of a specific node type, but at the same time route a data object between two physical ports. This can be achieved since each Function Cell has six physical ports but the node types use at most four of them. Thus, the remaining two ports could be used for routing.

Since each Function Cell within an array must be able to handle any of the node types and data types it must be reconfigurable, i.e., it must be possible to load and store a configuration within a cell.

To make it possible to adapt the array size to a given application, it is important to make the architecture truly scalable. It should be possible to enlarge an existing array simply by adding more Function Cells. The main problem that prevents scalability is that it is difficult to provide all cells with a common clock. This problem can be solved by making the communication between cells asynchronous. This implies that each Function Cell should have its own clock.

The main part of the Function Processor architecture is the array of Function Cells. The size of this array imposes a limit on the size of programs that may be executed on it, i.e., the number of nodes in a physical mapping of a HIDFG must not exceed the number of Function Cells in the array. In order to make this restriction less severe, it must also be possible to store multiple configurations within one array, and to change quickly between them.

In the following sections we will discuss the architecture of the Function Cell in more detail.

The second part of the Function Processor architecture is the Array Control Unit (ACU). The ACU makes it possible for a Function Processor to communicate with other parts of a system, e.g., a host processor, a memory or other Function Processors. The ACU is responsible for loading configuration data into the array, and for handling input and output of data during execution. It consists of a number of independently programmable communication channels, which are connected to the array.

4 Function Cell Architecture

The Function Cell architecture we are going to describe here can be divided into five separate parts, each of which is separately described below. An overview of the architecture is shown in figure 3.

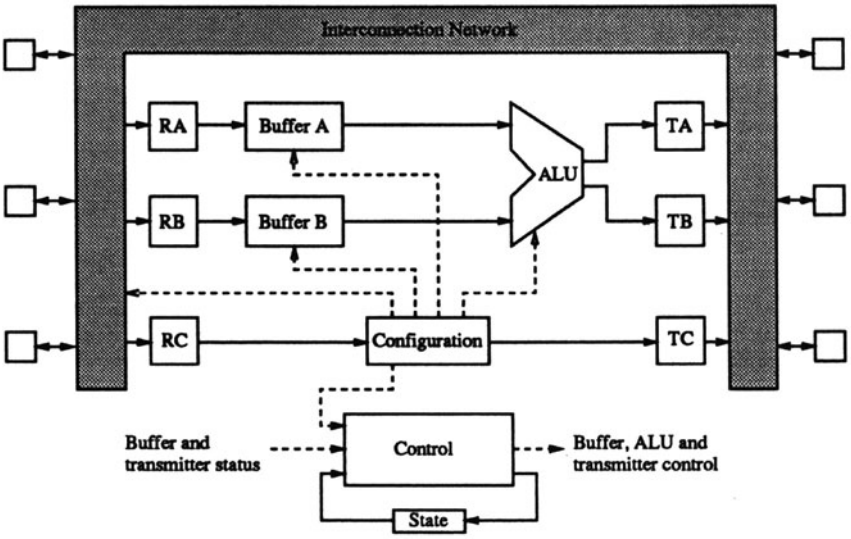


Figure 3: Function Cell Architecture

4.1 Ports

The Function Cell can be divided into two major parts; a functional unit and an interconnection network. The functional unit has three inputs called *logical input ports A, B and C*, and three outputs called *logical output ports A, B and C*. The logical input and output ports A and B are used to implement the inputs and outputs of the different DFG node types. The C ports are used for configuration and routing. At the logical ports are receivers and transmitters (RA, RB, RC, and TA, TB, TC, respectively) placed. These are responsible for the asynchronous communication between logical ports on different cells via *interconnection buses*.

The *interconnection network* is a flexible reconfigurable switch capable of connecting each of the logical ports to any one of six *physical ports*. Any physical port can act either as an input port or as an output port. In the Function Processor, every physical port on one cell is connected to exactly one physical port on another cell, or to one of the external communication ports (see figure 1). Despite this very rigid physical structure, the flexible interconnection network in the cells makes the task of mapping DFG:s onto the array much easier. The interconnection network configuration is programmed at the same time as the functionality of the cell is programmed.

Data are communicated between cells in units of *words*. A word is also the amount of data that the ALU can process in one operation. The number of bits in a word, the *wordwidth*, will not be fixed in this architecture description. It can vary between different implementations and applications, but a practical lower limit is probably 16 bits.

4.2 Configuration

The Function Cell is programmable, or rather, reconfigurable. Each configuration specifies the DFG node operation performed by the cell, as well as through which physical ports the cell communicates with its neighbours. The configuration is stored as a set of control words in a number of configuration registers. Several configurations can be stored in the cell, but only one configuration can be active during the execution of one function, i.e., the configurations should not be seen as instructions in a sequential program. By storing several configurations in the cells, the Function Processor can quickly switch between different tasks by means of a global signal telling the cells which configuration to use. The exact number of configurations and control words that the cell can store is implementation dependent, but the total number of bits required for one configuration can be estimated to 48 plus the width of one word. One of the words in the configuration is always a constant value used by the input buffers, either to recognize data objects, or as a constant input.

The Function Cell can be either in *execution mode* or in *configuration mode.* The mode is selected by means of an external signal. In configuration mode, the cell shifts configuration data from the logical C input into the currently selected configuration registers, whose earlier contents simultaneously are shifted out via the logical C output. This operation is independent of the current configuration, and the logical C ports are routed to predetermined physical ports. This means that, by putting all cells in an array in configuration mode, their C input and output ports form a chain through which configuration data can be shifted to all cells. After a new configuration has been stored or selected, the input buffers and the control unit are emptied and reset. The cell can then start its new operation in execution mode. In this mode, the C input is directly connected to the C output, bypassing the configuration registers. This direct link can be used to route data through the cell independently of its other functions. Thus, mapping of DFG:s onto a cell array is made easier.

4.3 Input Buffers

The input buffers are the parts that contribute the most to the special characteristics of the Function Cell. Their main purpose is to asynchronously receive operands to the node function implemented by the cell, and to inform the control unit when operands are available. In accordance with the architectural requirements, one of the buffers can be configured as a multi-word FIFO or LIFO (stack) buffer. Simulations of benchmark functions indicate that the buffer size should be at least 256 words. An overview of the input buffer can be seen in figure 4.

We have, however, also chosen to let the input buffers provide support for a number of different data types. The idea is to let the input buffers be responsible for recognizing objects of any specified data type, thereby making the control algorithms for different node types independent of the operand types. The *merge* node, for instance, is implemented by an algorithm that only has to specify that as soon as one data object is available at any input it should be copied to one of the outputs. It does not have to be concerned about whether the object is a list that consists of several words, or a simple scalar value represented by a single word. These functions are performed by a special part of the input buffer called *object detector.* The object types to be expected at the input buffers

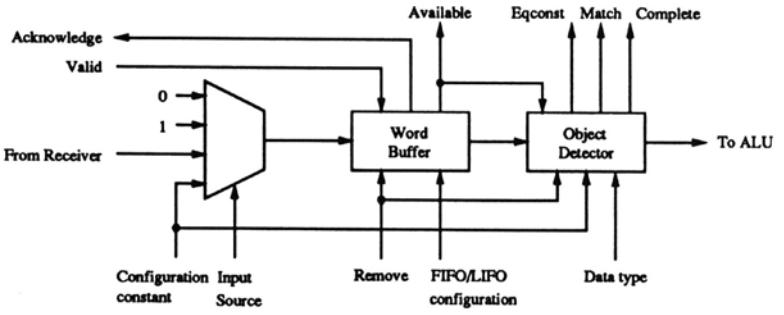


Figure 4: Input Buffer

are specified as a part of the cell configuration. This information is taken from an analysis of the DFG that has been mapped onto the Function Processor.

The object detectors can be configured to recognize one of four different data types. Objects of all types are represented by sequences of elements, sometimes terminated by a word with a special value. The elements can be either objects of another type less complex than the type of the object they are part of, or single words. The objects of a type can be built up in different ways, with different constructors. One of the constructors for each type is designated the *primary constructor* of that type. The object detector is capable of recognizing objects built with the primary constructor.

The least complex data type supported is the *scalar type*. A scalar object always consists of a single word. If this word is equal to the configuration constant, the object is recognized as the primary constructor. The next more complex data type is the *pair type*. A pair is built up either by two scalar values, in which case it is formed with the primary pair constructor, or a single word equal to the configuration constant.

The third data type is the *string type* which can have elements of scalar or pair type. A string object consists of zero or more elements followed by the configuration constant (note that no element must contain the configuration constant as this will be interpreted as the string terminator). Typically a string can be a null-terminated character string, i.e., a sequence of character codes followed by character code zero. A string containing at least one element is built up with the primary string constructor. In a way, the string type is many different types, each with a different termination word. Therefore, a string can also contain elements of any other string type as long as no string element contains the termination word of the string it is an element of. The most complex type is the *list type*. A list is very similar to a string, except that it can contain elements of any of the other types, and that the termination word for lists has a predefined value called *nil-token*, which is equal to the smallest two-complement value a word can contain, i.e., $-2^{wordwidth-1}$. A non-empty list is built with the primary list constructor (often called *cons*). The list has the advantage over other types that the object detector does not depend on the configuration constant which therefore can be used for other purposes.

The object detectors output four status signals used by the control unit to determine its actions. The first signal, *available*, indicates if any word (any part of an object) is available in the buffer. The second signal, *eqconst*, indicates if the word available at the

buffer output (if any) is equal to the constant specified in the configuration. The third signal, *complete*, indicates when all the words in a complete data object of the specified type have been received. The fourth signal, *match*, indicates if a complete object was constructed with the primary constructor of the specified type.

The input buffers are so autonomous that the control algorithms only have to control a single operation on them. This operation is to remove the word currently available at the buffer output, called the topmost element, and replace it with the next word in the buffer if there is one. This is controlled by the *remove* signal. When, for instance, an addition node has added two scalar operands, these are removed from the input buffer, and the cell starts to wait for two new operands to become available.

Usually, an input buffer receives data asynchronously from a receiver with which it communicates via a handshaking protocol (*valid, acknowledge*). It can, however, also be configured to continuously receive a constant value; zero, one or the configuration constant. Zero and one are frequently used constants, and are therefore made available even in configurations where the configuration constant is used for other purposes.

4.4 ALU

The Arithmetic and Logic Unit (ALU) is responsible for all data processing in the Function Cell. It takes the two operands coming from input buffers A and B as input, and it produces two results on separate, independent outputs. Each result can be equal to either of the two operands, or be the result of an arithmetic or logic operation on the two operands. The results are selected by the control unit. The operation performed by the ALU is determined directly by the contents of the configuration registers.

In this way, the control algorithms become independent of the ALU operation and the number of different node types supported by the Function Cell is kept down. The set of operations that the ALU can perform can vary between different implementations, but traditional arithmetic operators (add, subtract, multiply), comparison operators (which produce boolean results), and the boolean constants true (-1) and false (0) should be supported. There should be no need to support unary operators since either operand can be set to a constant value in the input buffers.

The ALU outputs are connected to the transmitters at the logical outputs A and B. The transmitters are controlled directly by the control unit which informs them when valid data are available at the ALU outputs. The transmitters also produce status signals informing the control unit when the last valid data have been sent out. Data are not sent by a transmitter before all the receivers it is connected to have informed it via the interconnection buses that they have passed on the last transmitted word to their input buffers. This information is exchanged between the transmitters and the control unit by means of *valid* and *ready* signals.

4.5 Control Unit

The control unit is a finite state machine that implements control algorithms for all node types supported by the Function Cell. After a cell has been configured, it is reset. This means that the control unit enters a special start state in which it inspects the configuration registers to determine what node type has been selected. It then enters

```
The switch node reads an object from input A. If this object is a primary constructor object of
the input A type, one data object is read from input B and sent to output A, otherwise it is
sent to output B. State 1 is the initial state.
State 1. Read a complete object from input A:
        If any word of input A object is available
            If input A object is complete go to state 2
            Else remove one word of input A object and go to state 1
        Else go to state 1
State 2. Send a complete object from input B to the selected output
        If any word of input B object is available
            If input B object is complete
                If input A object matches primary constructor
                    If transmitter A is ready
                        Remove one word from inputs A and B
                        Output the word to transmitter A
                        Go to state 1
                    Else go to state 2
                Else
                    If transmitter B is ready
                        Remove one word from inputs A and B
                        Output the word to transmitter B
                        Go to state 1
                    Else go to state 2
            Else
                If input A object matches primary constructor
                    If transmitter A is ready
                        Remove one word from input B
                        Output the word to transmitter A
                        Go to state 2
                    Else go to state 2
                Else
                    If transmitter B is ready
                        Remove one word from input B
                        Output the word to transmitter B
                        Go to state 2
                    Else go to state 2
        Else go to state 2
```

Figure 5: Control Algortihm for the *switch* Node

the initial state in the control algorithm for this node type. A control algorithm usually consists of a few states which typically correspond to how much of the necessary operands that has arrived. In figure 5, the control algorithm for a *switch* node is given as an example.

The control unit chooses its actions according to the status signals it receives from the input buffers and the transmitters. These status signals are sampled by the control unit at regular intervals. As mentioned above, the status signals from each of the input buffers are *available, eqconst, complete and match*, and from each of the transmitters *ready*. In each state, the control unit also outputs a set of control signals which have been described above. These signals are the input buffer *remove* signals, the ALU output select signals, and the *valid* signals to the output transmitters.

5 Implementation Results

A single chip VLSI implementation of the Function Cell architecture presented here, has been made using a standard cell silicon compiler. The technology used is a double metal layer $1.5\mu m$ CMOS process. The implementation uses 16-bit words and contains a 512-word buffer. The largest parts in terms of chip area is the buffer and the interconnection network including the transmitters and receivers. The interconnection buses are only 6 bits wide (4 bits data, 2 bits handshaking) in order to reduce the size of the interconnection network and the chip pin-count. This means that a 16 bit word is communicated as four 4-bit groups. A whole 16-bit word can, however, be communicated in parallel with the completion of one operation cycle which takes approximately $100ns$.

The configuration data for this cell consist of 61 bits, so the whole configuration is divided into four words. Thus, it takes at least $400ns$ to configure each cell in an array. The number of cells in a practical implementation of the Function Processor should be at least 100–200, which would result in a total configuration time of at least $40 - 80\mu s$, assuming that cells are configured sequentially. This implementation allows only one configuration to be stored in a Function Cell. However, it can easily be extended to allow multiple configurations to be stored in each function cell. That would make it possible to quickly switch between different configurations by means of a global signal which tells the cells which configuration to use. The number of configurations stored in a cell could probably be in the range 8–16 without making the hardware significantly more complex.

The Function Processor has been simulated, executing automatically compiled and optimized HIDFG:s for a number of different functions. The simulated functions are:

fac: computes factorial number 20 (20!).

numbers: generates a list of integers from 1 to 200.

fib: computes the 15:th Fibonacci number.

sum: computes the sum of the list of integers from 1 to 20.

sort: builds the descending list of numbers from 20 to 1, and sorts that list in ascending order using the insertion sort algorithm.

Function	*Nodes*	*Cells*	*Execution Time (μs)*			*Speedup over*	
			Function Processor	*SPARC*	*RT/PC*	*SPARC*	*RT/PC*
fac	10	13	12.9	280	700	21.7	54.3
numbers	10	18	140.7	2320	6300	16.5	44.8
fib	15	17	1813.5	14085	37300	7.8	20.6
sum	21	32	64.2	437	1900	6.8	29.6
sort	36	57	873.6	3090	9300	3.5	10.6
primes	42	83	179.1	990	2400	5.5	13.4
queens	114	–	450.3	9770	24000	21.7	53.3
substitute	141	–	36.6	870	–	23.8	–

Table 1: Results from Simulated Performance Measurements.

primes: produces a list of all prime numbers smaller than 20, using the Sieve of Erathostenes method.

queens: generates a solution to the problem of placing 7 queens in safe positions on a 7 by 7 chess board.

substitute: finds and replaces the first occurrence of a search string in a text.

The main results of these simulations are summarized in table 1. The first column contains the number of nodes in the HIDFG for the function. The effect of mapping the HIDFG onto the Function Processor is shown in the second column, which contains the number of Function Cells that are needed for the mapping, i.e., the number of HIDFG nodes plus the number of cells used only for routing. These mappings, except for the mapping *sort*, have been made by hand. The mapping of *sort* has been made automatically using an experimental version of a program currently being developed especially for mapping HIDFG:s onto the Function Processor. We have not yet been able to produce any satisfactory mappings for the *queens* and *substitute* functions. More information about the problem of mapping irregular graphs onto this type of architecture can for example be found in [KMPS88, WSS89].

The next three columns show execution times for the functions on three different machines. The first execution time is for the Function Processor executing the mapped HIDFG (except for the last two functions which have been simulated without routing delays). This has been taken from a simulator and assumes a $100ns$ cycle time. To compare this with state-of-the-art single processor performance, the next two columns show execution times for two conventional workstations; the SUN Sparcstation 1 and the IBM RT/PC. On these machines, the functions were written in LML (Lazy ML) and compiled with the LML compiler [Joh87, Aug87] which produces graph reduction code. The last two columns show how much faster the Function Processor is compared to the other two machines.

For the mapped functions, the execution times for the unmapped graphs have also been measured. This shows that the routing delays caused by the mapping increases the execution time by 7–75%. There does, however, not appear to be any correlation between the execution time increase and the graph size for these examples.

6 Conclusions

The Function Cell architecture presented here fulfills the requirements stated in section 3. So far, the implementation results have shown that the architecture is realizable and in accordance with the assumptions made in earlier stages of the project.

The most important restriction in the architecture presented here has to do with the data types. There is no support for data structures which can not easily be represented as a sequence. One solution to this problem is to use the Array Control Unit to access parts of data structures to make it possible to handle them as pointers within the array. It is, however, not yet clear how this solution would affect the Function Processor performance.

The compilation method does currently not support general higher order functions. It does, however, support a set of predefined higher order functions, and it is also possible to extend this to cover some user-defined functions as well.

Since the Function Processor is not a general-purpose architecture, its usefulness is dependent on the type of system it is used in. We do, however, feel that the performance results obtained so far, are encouraging and motivate investigations of different types of systems using the Function Processor.

7 Acknowledgements

We would like to thank Tony Nordström for his contributions to this project. The project has been financially supported by the Swedish Board for Technical Development, STU.

References

[AN87] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. In *Proceedings of the PARLE Conference*, June 1987.

[Arv80] Arvind. A data flow architecture with tagged tokens. Technical report, MIT, Cambridge, Massachusetts, June 1980.

[Aug85] L. Augustsson. Compiling pattern matching. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[Aug87] L. Augustsson. *Compiling Lazy Functional Languages, Part II.* PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, Sweden, November 1987.

[Bac78] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, 21:280–294, August 1978.

[CP90] David E. Culler and Gregory M. Papadopoulos. The explicit token store. *Journal of Parallel and Distributed Computing*, 10(4):289–308, December 1990.

[Den80] Jack B. Dennis. Data flow supercomputers. *IEEE Computer*, pages 48–56, November 1980.

[Hug89] J. Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, 1989.

[Joh85] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, 1985.

[Joh87] T. Johnsson. *Compiling Lazy Functional Languages.* PhD thesis, Dept. of Computer Science, Chalmers University of Technology, Göteborg, Sweden, February 1987.

[KAGER82] S-Y. Kung, K.S. Arun, Ron J. Gal-Ezer, and D.V. Bhaskar Rao. Wavefront array processor: Language, architecture and applications. *IEEE Transactions on Computers*, C-31(11):1054–1066, November 1982.

[KLJH87] S-Y. Kung, S.C. Lo, S.N. Jean, and J.N. Hwang. Wavefront array processors – concept to implementation. *IEEE Computer*, pages 18–33, July 1987.

[KMPS88] Israel Koren, Bilha Mendelson, Irit Pedel, and Gabriel M. Silberman. A data-driven VLSI array for arbitrary algorithms. *IEEE Computer*, pages 30–43, October 1988.

[Kun82] H.T. Kung. Why systolic architectures? *IEEE Computer*, (1):37–46, January 1982.

[Kun88] S-Y. Kung. *VLSI Array Processors.* Information and System Sciences Series. Prentice Hall, Englewood Cliffs, New Jersey 07632, USA, 1988.

[McG89] J.R. McGraw. Data flow computing: System concepts and design strategies. In S.P. Kartashev and S.I. Kartashev, editors, *Designing and Programming Modern Computer Systems, Vol.III*, chapter 2, pages 73–189. Prentice Hall, 1989.

[MMU87] W. Moore, A. McCabe, and R. Urquhart, editors. *Systolic Arrays.* Adam Hilger, 1987.

[Rea89] Chris Reade. *Elements of Functional Programming.* Addison-Wesley, 1989.

[She85] Mary Sheeran. Designing regular array architectures using higher order functions. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, volume 201, pages 220–237. Springer Verlag, 1985.

[Vas90] Jesper Vasell. Implementing functional programming languages on wavefront arrays. Licentiate thesis 84L, Department of Computer Engineering, Chalmers University of Technology, 412 96 Göteborg, Sweden, April 1990.

[Vee86] Arthur H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18(4), December 1986.

[VV91] Jonas Vasell and Jesper Vasell. A functional programming technique for programmable wavefront arrays. In E.F. Deprettere, editor, *Algorithms and Parallel VLSI Architectures*. Elsevier, Amsterdam, The Netherlands, (to appear in 1991).

[WSS89] Shlomit Weiss, Ilan Spillinger, and Gabriel M. Silberman. Architectural improvements for data-driven VLSI processing arrays. In *Proceedings of the 1989 Conference on Functional Programming Languages and Computer Architecture*, pages 243–259, London, 1989.

THE G-LINE
A DISTRIBUTED PROCESSOR FOR GRAPH REDUCTION

R.Milikowski and W.G.Vree

Computer Systems Department, University of Amsterdam,
Kruislaan 409, 1098 SJ Amsterdam, the Netherlands,
milikows@fwi.uva.nl.

Abstract

The G-line is a horizontally coded graph reducer. In lazy functional languages much time is used to manage the graph. A method has been developed to perform construction of a subgraph in a single parallel access to the graph memory. With a simulation, using infinite hardware, we have shown that the architecture performs well. We argue that a realistic implementation is possible.

1. Introduction

The distributed G-processor is specially designed to perform lazy graph reduction. It is a hardware implementation of the abstract Göteborg machine [1]. In the distributed G-processor the price of laziness is small as managing the graph scarcely causes cost.

Laziness is a property in which computational work is postponed until it is really necessary. Until that moment the representation of that work is stored in the graph. Often evaluation is never called for. In the second place laziness avoids doing the same work more than once. This is implemented in the graph by means of sharing. The core of a machine which implements lazy graph reduction generally consists of a heap containing the graph and a stack containing pointers into the graph. During the reduction process data is moved between the program code, the graph and the pointerstack and inside the pointerstack and the graph. Operations on the graph are mostly seen as a burden. Constructing it, keeping it up to date, consulting it and collecting garbage in it costs a lot of time [2]. Research is done to limit the work on the graph and gains have been made in different ways. Compared to the 'standard combinators' of Turner the 'program derived combinators' as used in the G-machine are more efficient allowing larger steps in the reduction process and accessing the memory substantially less [3]. Compiler directed approaches extract as much information as possible in advance from the expression to be evaluated. By means of strictness analysis building of graphs can be prevented, and by sharing analysis unnecessary updates can be avoided. However, in programs of a realistic size only a part of graph construction can be avoided.

In recently developed abstract machines more efficient representations of expressions in the heap are used. These are the 'TIM-machine' which uses closure like representations and the 'spineless tagless machine', and the most recent version of the G-machine - the <v,G> machine [4,5,6,7]. Implementations of these reducers run on existing processors

by further compilation. However, there are also well known examples of special processors designed for graph reduction. The first machines reduced SKI-combinators: SKIM and NORMA [8,9]. To assess the performance of supercombinators simulators have been built. Kieburtz designed a RISC architecture mainly based on the Göteborg machine. It contains a special designed instruction-fetch-unit and uses caching [10,11]. It makes efficient execution of sequential G-code possible. At the moment by far most the architectural work on reduction machines is directed towards building parallel reducers and concentrates on the communication between the processors. Little attention is paid in current research projects to the processor itself [12].

The distributed G-processor, designed by the first author, is based on parallel access of the graph memory and the pointerstack. Constructing a subgraph only takes a single access to the heap. This is realized by using horizontal microcode. This micro parallelism has to be distinguished from parallel redex reduction. The latter means reduction of different redexes of the graph on different machines. The micro parallelism in the G-line is parallelism inside the reduction of a single redex. First we have implemented a brute force version in which we do not bother about the amount of hardware and in which the parallelism is unbounded. The main goal is to study the validity of the architecture idea and to gather statistics about performance. The results are presented in this paper. In a next phase of our research we apply this idea to develop a realistic distributed design that could be realized in hardware.

2. Basic graph operations

The reduction process consists of various parts. Before a function is called, pointers to the arguments are collected on the top of the pointerstack. Sometimes pointers to the arguments are present elsewhere on the pointerstack. In that case these pointers are copied to the top of the pointerstack. In other cases the pointers must be fetched from the graph by an operation called 'unwinding'. Pointers to the arguments being on the top of the pointerstack, the function is called to do its work. The called function terminates its work by updating the root of the redex with the result. The result may be a value (present in a node created recently) or the root of a constructed subgraph. It may be necessary to evaluate the arguments of the function themselves before the function can use them. In that case the evaluation mechanism is executed recursively. Many variations of these operations are possible and indeed used but we will concentrate on the simplest versions. In fig 2.1 - 2.3 reduction operations in a graph with binary nodes are shown.

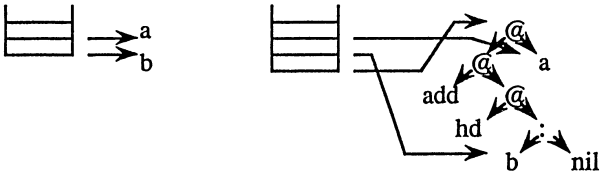


Fig 2.1 A subgraph representing the expression 'add (hd(cons b nil)) a' is constructed. The left part of the figure shows the pointerstack before this operation. Afterwards the top of the pointerstack points to the root of the subgraph just constructed.

The pointers a and b point to nodes somewhere in the graph. These nodes may themselves be the root of subgraphs. The graph construction is done by moving data

from the program and from the pointerstack to the heap. After constructing a new subgraph a pointer to its root is at the top of the pointerstack.The most extensively used graph operations (graph construction, unwind and update) are pictured in fig 2.1. In the left of the figures the situation on the pointerstack before the operation is shown and in the right the situation afterwards.

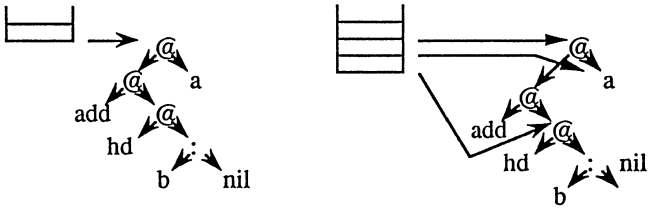


Fig 2.2 Building argument stack. On the left part of the figure the top of the pointerstack points to the root of a redex.. On the right part pointers to the arguments of the function add have been pushed.

Before unwinding starts the top of the pointerstack points to the root of a redex (fig 3.2). A redex is a function f applied to a number of arguments. These arguments must be copied to the pointerstack and will then be used by the code of the function f which is called after unwinding.

The arguments are found by inspecting the type field (tag field) of the left child node, starting at the root of the redex. If it is an apply node (@) then the right child is moved to the pointerstack and the left child is entered. This is repeated until the left child is a function node.

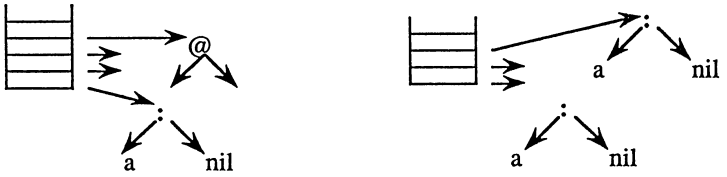


Fig 2.3 Updating; on the left is the result of some computation pointed to by the top of the pointerstack. on the right this result has been copied in the root node of the redex.

The update is done by reading both pointers of the pointerstack and then moving the contents from one node to another (fig 2.3). If not shared the source node will become garbage.

The G-code is a specially designed graph directed language to perform the graph operations [1]. Usually, this code is in a next step compiled to code for some existing machine.

2.1. Memory memory bottle-neck

Execution of the basic graph operations consumes much time of the reduction process. Different performance data have been published, using different ways to measure the performance giving an insight in the way work is spent during the reduction process.

Results have been published by Kieburtz on his S-machine - an optimized version of the G-machine concerning real execution times [13]. The ALU operations generally consume a very limited portion of time, in this benchmark from 12% to 2 % of the execution time. Nearly all the rest are moves of data from heap to heap, from stack to stack, from stack to heap from, the program to the heap and vice versa.Hartel has run a realistically sized benchmark on a variant of the G-machine [14]. He counted memory accesses, resulting in similar conclusions. We will call this the memory memory bottle-neck. One may expect parallel operations on the memory will speed up the execution.

3. Constructing a subgraph

As a running example we will take append, which is is lazy in cons.

append x y = if (null x) then y else (cons (hd x) (append (tl x) y))

Entering the else part of the expression causes construction of a suspension of append because cons is not strict in its arguments. The work stored in the else part is only done if the 'need to print' forces so. Until that moment a representation of this expression (a suspension) is stored in a graph, shown in fig 3.1. The suspensions of (hd x) and (tl x) are also constructed, though advanced compiler techniques could avoid this. Each node of the graph consists of three fields. The first field of a node - the tagfield - contains information about the node type and some additional information. Both other fields may be used to store values, e.g. floats. In function nodes the second field contains the address of the code. The second and third fields of an apply- and a cons node contain pointers. In the example the crossed fields are unused. More compact representations exist. However, we start our design from this conventional memory representation.

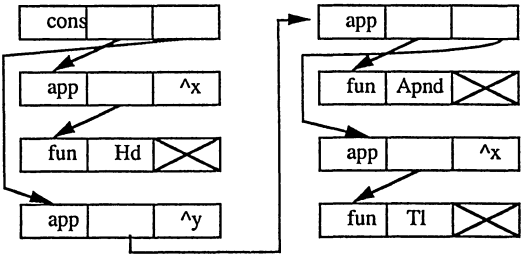


Fig 3.1. Representation of a suspension in the heap as written after execution of a G-code sequence.

The work to be done when writing this graph includes 21 accesses of the graph memory (all fields without a cross). The contents of the tag fields are known at compile time and are normally stored in the program code. The offsets inside the reduct can be computed by the compiler in advance. At the moment the address is known at which construction of this piece of graph starts all 'local pointers' (pointers inside the subgraph) can be computed. Only the pointers ^x and ^y are unknown at compile time and must be fetched from the pointerstack at runtime. Writing this subgraph in a conventional machine requires multiple accesses to the program code, the heap and the pointerstack.

4. Micro parallel architecture

A subgraph to be written will be considered as a vector of which the elements are the fields of the subgraph. The architecture of the G-line is organized in such a way the elements of this vector can be written in parallel. In contrast to vectors in numerical computing the fields of these vectors are not constructed uniformly. The contents of the various fields of a single subgraph may originate from the operand of the code, from the pointerstack or from an address computed at runtime. The mechanism to write a single field is rather complex, actually nearly as complex as a whole conventional G-machine.

In the G-line the graph memory is divided in memory banks. All memory banks are managed by their own G-machine. The global buses that connect these G-machines are at the same time the address and data bus of the heap. These G-machines are equal and execute synchronous nearly the same code when a graph access has to be done. A write on the memory can be handled locally by the appropriate machine. No communication with the other G-machines is required. In that way it is possible that a number of writes can be done in parallel which is the case in graph construction. A read on the heap in the conventional architecture means the data is moved from the heap via the bus to the G-machine. In the G-line data is moved from one of the memory banks via the global buses to all replicas of the G-machine. That G-machine that manages the specific memory bank to be read performs locally the addressing of the memory bank and takes control of the global buses. All local machines are *nearly* complete copies of the whole G-machine in hardware and in software. They differ by the fact they manage different memory banks. Furthermore those sections of the program that deal with graph writing are distributed over the different machines, as the graph writing itself is distributed over the different machines. Local G-machines contain replicas of the valuestack, the pointerstack, the ALU , the program counter (PC) and the control store. Care has to be taken that the replicas of the stacks and the PC remain identical in the local G-machines. This invariant will be called the *shadow condition* which has to hold always after execution of an instruction. There is no hierarchy between the machines and no scheduler. In the next paragraphs this architecture and the way a program runs on this distributed processor will be described.

4.1. G-line

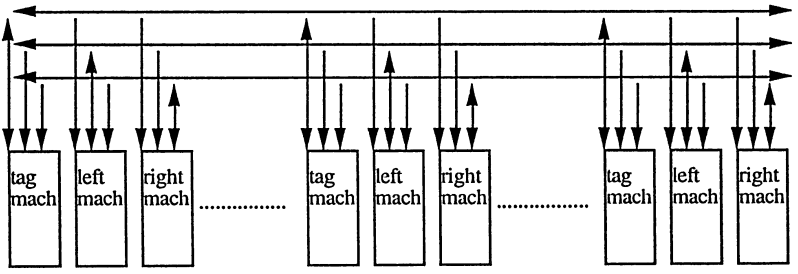


Fig 4.1 G-line. Each of the machines is a replica of the G-machine containing a memory bank of the heap.

The number of G-machines in the G-line architecture of fog 4.1 is equal to the maximum number of fields of a subgraph that will possibly be written in a specific program.

As each node consists of three fields, three G-machines together forming a cluster, are needed to write those fields in parallel.

First, we discuss the different units of the G-machines. The tag, left and right machines are not exactly equal. Fig 4.2 shows a left machine.

The heap: The heap in fig 4.2 contains the graph and consists of 3n memory banks (sometimes called local memories), in which n is the number of clusters. In an alternating order these are memories containing tag fields, left fields and right fields. A heap cell consists of a word from all three local memories in a cluster on a single global address. The tagfield normally contains the tag and the number of arguments of a function, and some runtime information. According to the shadow condition the heap pointers are equal in all local machines. To guarantee that the heap pointers in all machines are equal after garbage collection, a compacting garbage collector will be used. The heap has bi-directional communication with the global buses. The tag memory with the tag bus etc.

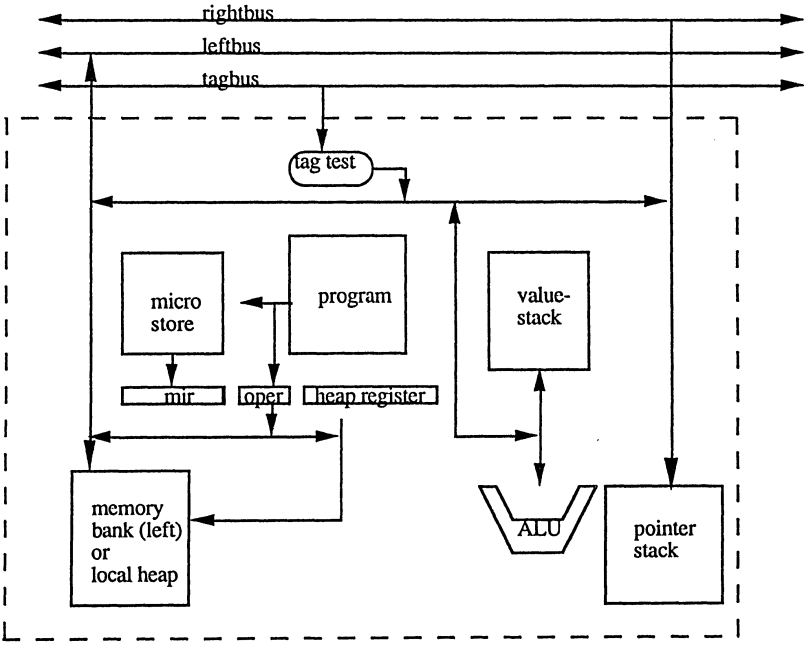


Fig 4.2 Left machine

Program memories are identical in each machine. The start address of a specific function are the same everywhere. The code is equal, except for the MKGRAPH instruction, which is introduced in the next section.

The microstore: Dividing the execution of instructions over a number of machines, makes the operation to be done in each G-machine very simple, especially after some minor optimizations, which we treat in the final paragraph. UNWIND remains the most complex instruction.

The heap register: The heap register contains the local heap pointer which is equal in all G-machines. During graph construction it may be necessary to write locally a remote pointer (to a node constructed in parallel in a different cluster). It is computed locally from the heap pointer and the operand of the MKGRAPH instruction, in this case containing the number of the remote cluster.

The pointerstack: Each machine contains a copy of the pointerstack. The pointerstacks are kept identical during execution of the program. The pointerstack has only input from the global right bus and only locally delivers output

ALU and valuestack: The ALU only operates on data present on the valuestack. The valuestack does not communicate with the outside world.

The buses: The distributed G-processor has three global buses. Each G-machine has access to the tag the left and the right bus. Each time only a single machine will do write operations to the buses. It is analogous to the way data in a memory consisting of different banks are put on the bus. An address on the address bus will cause only a single memory bank to become active. All three buses are data buses. Global addressing is only used in case of memory indirections. The address then originates from data, in casu a left field. So global addressing is only done by the left bus. Locally only the heap pointer addresses the heap.

In the G-line only the contents of heap words are transported over the global buses. The state of the machine uniquely determines from which global heap address this word is fetched and so which local machine will access the buses. No contention is possible and no arbitration is required. The code produced by the compiler and the synchronous execution of the processes guarantee this. The tag bus is often used for testing as a part of the program control.

The clock: Besides the buses the clock is the only global hardware. The clock synchronizes the G-machines.

5. Programming the G-line

To present the working of the machine we show the append program as it is compiled for the the conventional G-machine, using the compilation scheme of Johnsson [1].

Instructions 7 - 17 construct the subgraph of fig 3.1. As each G-machine in the G-line has to write only a single field of a specific subgraph, this sequence will be replaced in each G-machine by code to write only that field.

```
0    FAPND    PUSH     0     || copy pntr to x to top of pointerstack
1             EVAL           || evaluate x
2             NULL           || test if x = null
3             JFALSE   1
```

```
4                PUSH      1        || copy pntr to y to top of pointerstack
5                EVAL               || evaluate it
6                JMP       2

7       LABEL 1  PUSHFUN   FHD      || else; construction of suspension
8                PUSH      1        || copy pntr to y to top of pointerstack
9                MKAP               || make apply node
10               PUSHFUN   FAPND    || push pointer to append node
11               PUSHFUN   FTL      || push pointer to tail node
12               PUSH      3        || copy pntr to y to top of pointerstack
13               MKAP               || make apply node
14               MKAP               || make apply node
15               PUSH      3        || copy pntr to y to top of pointerstack
16               MKAP               || make apply node
17               CONS               || make cons node

18      LABEL 2  UPDATE    3        || update root of redex with result
19               RET       2        || pop arguments and return to caller
```

Fig 5.1 Sequential G-code of the function append x y = if (null x) then y else (cons (hd x) append (tl y)).

This varying code will be the instruction 'MKGRAPH_Action Field'. Four different MKGRAPH_ opcodes exist, according to the the kind of action needed to write a field.

MKGRAPH_heap n: construct global heap address by prefixing local heap pointer by and write this into new local heap field
MKGRAPH_stack n: copy n-th element relative to top of pointerstack in new heap field
MKGRAPH_copy n move n into new heap field
MKGRAPH_nop machine performs nop on heap

The nodes of a new subgraph are written in parallel as a vector on the same local addresses in the different clusters of the G-line. An offset between two nodes in a subgraph as in fig 3.1 is translated into an distance between clusters in the G-line. This distance, or the number of the remote processor, is stored in the operand of the MKGRAPH_addr instruction. As the local heappointers are equal a global pointer to the remote node can be constructed at runtime locally by concatenating the contents of the heap pointer register with the operand of the MKGRAPH_addr instruction. If a operand has to be stored in the graph, a 'MKGRAPH_copy n' instruction is used. This is the case for values in the left and the right fields. Tag fields are always constructed by a MKGRAPH-copy instruction. Four MKGRAPH_ instructions are defined.

The transformed program is shown in fig 5.2.

Fig 5.3 shows all local MKGRAPH instructions derived from the sequential program. The first column denotes the machine (number of the machine cluster and machine in the cluster). In the tag machines always the operand of the MKGRAPH instruction is written into the heap. One may consult fig 3.1 to see how each MKGRAPH instruction constructs a field of this subgraph. The only difference is that the arity of the function is added in the operand.

```
0      FAPND      PUSH                0
1                 EVAL
2                 NULL
3                 JFALSE              1

4                 PUSH                1
5                 EVAL
6                 JMP                 2

7      LABEL 1    MKGRAPH_Action      Number    || construct field subgraph
                                                   operand differs per machine
8      LABEL 2    UPDATE              3
9                 RET                 2
```

Fig 5.2 Code of append in G-machines of the G-line.

```
   MACHINE          OPCODE              OPERAND        COMMENTS

0  tag machine      MKGRAPH_copy        cons
1  tag machine      MKGRAPH_copy        app
2  tag machine      MKGRAPH_copy        fun    1
3  tag machine      MKGRAPH_copy        app
4  tag machine      MKGRAPH_copy        app
5  tag machine      MKGRAPH_copy        fun    2
6  tag machine      MKGRAPH_copy        app
7  tag machine      MKGRAPH_copy        fun    1


0  left machine     MKGRAPH_heap        1      || addr(1,hp)
1  left machine     MKGRAPH_heap        2      || addr(2,hp)
2  left machine     MKGRAPH_copy        17     || code addr FHD
3  left machine     MKGRAPH_heap        4      || addr(4,hp)
4  left machine     MKGRAPH_heap        5      || addr(5,hp)
5  left machine     MKGRAPH_copy        0      || code addr FAPND
6  left machine     MKGRAPH_heap        7      || addr(7,hp)
7  left machine     MKGRAPH_copy        10     || code addr FTL


0  right machine    MKGRAPH_heap        3      || address(3,hp)
1  right machine    MKGRAPH_stack       0      || copy pntr to x from pstack
2  right machine    MKGRAPH_nop                || unused field
3  right machine    MKGRAPH_stack       1      || copy pntr to y from pstack
4  right machine    MKGRAPH_heap        6      || address(6,hp)
5  right machine    MKGRAPH_nop                || unused field
6  right machine    MKGRAPH_stack       0      || copy pntr to x from pstack
7  right machine    MKGRAPH_nop                || unused field
```

Fig 5.3 Instructions to construct the subgraph of FAPND in a distributed way.

In the comment is denoted which field of the subgraph of fig 3.1 is written to the local memory. By address(3,hp) is meant the address that is constructed by adding 3 as most significant bits to the local heap pointer. The common code address of a function is the address where the code of this function starts. In this example, though not shown, the code of FTL starts at 10 and of FHD at 17.

5.1. Running the program

All G-machines are started to run together. All execute, synchronized by the global clock, the same instruction. The way the G-instruction is executed in the different G-

machines depends on the type of the machine (tag left or right) and on the kind of instruction it concerns. The G-instructions can be split into three groups, according to their implementation in the G-machines of the G-line.

no_heaps: ADD (and the other arithmetic and logical operations), JMP, JFALSE, LABEL, SLIDE n, PUSH n. These instructions do not access the heap, they only use pointerstack, valuestack and ALU and are executed identical in all G-machines.
old_heaps: PRINT, GET, UPDATE, EVAL, UNWIND, NULL, HEAD, TAIL. Instructions access *existing* graph in the heap that is globally addressed. Depending from the fact if the addressed heap cell is in their own local memory the execution is partly different over the G-machines.
new_heaps: MKGRAPH_ Field. It creates new graph in the heap and accesses only the local memory. Is executed differently in different G-machines

Only the old_heaps communicate with the other G-machines in the G-line. Some examples will show the way the execution of the code is implemented.

ADD: All machines arrive at the same time at the instruction ADD and then their local ALU adds the top two elements of their valuestacks, which were the same in all machines, and replace these by the result of the addition. After execution of ADD all valuestacks are identical again and the shadow condition is fulfilled.

GET. The instruction GET in the conventional G-machine reads the value (integer or boolean) from the heap node pointed at by the top of the pointerstack. This value is pushed on the valuestack. In the distributed machine each G-machine now looks if the address on top of the pointerstack points in its own local memory. If this is actually the case it moves the value from this node to the global left bus else it waits. Next, the rest of the GET instruction again is executed identically by all G-machines. Each one pushes the contents of the global left bus on its own local valuestack. Again: the shadow condition remains fulfilled.

EVAL AND UNWIND EVAL starts like GET, resulting in the root of the redex being on the buses. EVAL proceeds in all G-machines with a test on the tag bus. If the tag is a "cons" , a "number" etc, the contents of the node is available on left and right bus for processing by the next instruction. If it is an apply node each G-machine starts execution of UNWIND:

The contents of the root of the redex is still on the buses. Two actions are undertaken by each G-machine:
- The contents of the right bus is pushed on the pointerstack (it is the pointer to an argument)
- The contents of the left bus is inspected. It contains a pointer to a node one step deeper down in the spine. If it points in the local memory the contents of that memory location is moved to the appropriate bus. All G-machines repeat the UNWIND algorithm until a function-node appears on the buses. Then the contents of the left bus - the code address of this function - is moved to the PC in each G-machine.

All pointerstacks always remain identical (shadow condition).

MKGRAPH Field We have already described to which micro operations the variable part of the instruction is decoded. Furthermore, the address of the root, computed from

the local heap pointer, is pushed on the pointerstack in all machines. The local heap pointers are incremented all over the G-line.

Graph construction instructions from the conventional G-machine (MKAP, MKCONS, PUSHINT i, PUSHNIL, PUSHFUN f) are superseded by MKGRAPH in the G-line.

Special attention has to be paid to the UNWIND instruction in case the depth of the spine (d) is larger than the arity k of the function F at the bottom of the spine. In the conventional G-machine unwinding is done by first pushing pointers to the spine nodes on the pointerstack. In the next step those spine pointers belonging to the current redex are replaced by pointers to arguments. This is called restructuring of the pointerstack. In case d > k the top part of the spine will not be restructured. Restructuring is a costly operation. Our strategy is inverse restructuring or rewinding. Always pointers to the arguments are pushed on the pointerstack. In case d > k the arguments have to be replaced by the spine node pointers. Starting at the top node an inverse restructuring is performed. The argument pointer is replaced by the spine node pointer. This operation is repeated (d - k) times. An inverse restructuring operation certainly is not more costly than restructuring in the G-line. Moreover restructuring always is required in the conventional G-machine. As in our benchmark in less than 3 % of all UNWIND operations d > k, inverse restructuring is required in only few cases.

6. Generating the code

The G-code for the G-line is derived from the sequential G-code by a backend. This backend simulates the execution on the stacks and the heap. Those parts of the G-code which construct a subgraph (instruction 7 - 17 in the example) will do this in the simulated heap resulting in a piece of memory like the append subgraph in fig 3.1. Next we assign each field of this graph to a G-machine and MKGRAPH instructions are generated to construct this field.

7. Efficiency

We have compared the number of memory accesses in the conventional architecture with those in the G-line. To count the number of memory accesses in the conventional reducer is, with some minor differences, similar to the way Hartel has counted [14]. To each G-instruction a number of accesses to the different stores (heap and stacks) is assigned. The representation of the a subgraph in fig 3.1 also contains function nodes (for Hd, Tl and Apnd). This is not necessary as these nodes are preloaded by the machine and the subgraph only contains a pointer to those function nodes. We have corrected our measurements to account for this optimization. The result of this way of counting in the sequential G-machine is in the first column of fig 7.1. We have used peep hole optimization concerning the accesses on the valuestack. If a boolean is stored on the valuestack (p.e. after a tag test) and in the next instruction is popped again (in executing JFALSE) a short cut is used. The second column contains the results of the G-line. To count the number of accesses in the G-line we proceeded as follows. If all G-machines perform an identical memory operation (e.g read on a stack) in parallel, it is counted as a single memory access. If a specific G-machine reads the memory and the other machines are idle counting is continued in an active machine. This is the case when

executing the old-heaps. The maximum number of accesses by MKGRAPH in a single G-machine is one heap access, one access to the pointerstack if a data from this stack has to be moved to the heap and one access to the pointerstack to push a pointer to the root of the subgraph. To make sure the maximum work is measured, counting during execution of MKGRAPH is done in the the cluster that constructs the root of the subgraph.

We have used a set of benchmark programs that are compiled from SASL, a run time typed functional language, to Johnsson's intermediate language and then to G-code. 'Schedule' calculates an optimum schedule of 7 parallel jobs with a branch and bound algorithm [15]. 'Hamming' prints in ascending order the first 50 natural numbers whose primes are 2,3 and 5 only. 'Paraffine' enumerates in order of increasing size the first six paraffine molecules.

	sequential	*micro parallel*	*optimized*
		paraffine	
dump (context switch)	52 K	52 K	52 K
valuestack	77	77	77
pointerstack	429	209	143
heap	377	132	78
total	936	471	351
		hamming	
dump	98 K	98 K	98 K
valuestack	78	78	78
pointerstack	590	299	198
heap	519	192	109
total	1284	668	484
		schedule	
dump	58 K	58 K	58 K
valuestack	47	47	47
pointerstack	539	301	189
heap	498	188	97

Fig 7.1 Memory accesses (in thousands) in a conventional G-machine and in the G-line

In fig 7.1 we show the number of register saves, labeled "dump" due to a context switch (two register saves when a function is called and two more when returning from the function). The dump is not treated in the architecture because it is often implemented on the valuestack.

7.1. Optimizations

Some optimizations fit nicely in the architecture of the G-line because they are implemented very simply in the hardware or because they make further use of the micro parallelism. Two examples show this.

1) Care is taken that after returning from a function call the resulting value is on the buses. This is guarantied by the G-machine that does the update. As the result data is moved via a local bus to the local heap, the only thing to do is to enable the appropriate bus register. This means that after returning from EVAL no instruction needs to access the heap. This concerns instructions like GET, HEAD, TAIL which now become more simple.

2) In most reducers of supercombinators updating is optimized. E.g the function f x y = x + y is compiled to the G-code sequence PUSH 0, EVAL, GET, PUSH 2, EVAL, GET, ADD, MKINT, UPDATE 3, RET 2. MKINT reads the result of the addition from the valuestack, creates a new node with this value in the heap. This node is immediately used to update the root of the redex and made to garbage. The instructions MKINT, UPDATE 3 will be combined into something like UPDINT 3 [2]. This instruction directly does an update with the value at the top of the valuestack. In the G-line this can easily be implemented. The (left)machine containing the node to be updated fetches the value from its local value stack and moves it to its local heap. The tag-machine belonging to the same cluster moves the tag of type 'int' to the tag heap.

The same can be done with MKGRAPH Field, UPDATE n. Then creating a new node for the root of the subgraph can be avoided. As the value with which to update is available in each G-machine it can immediately be written in the root of the redex which now has been reduced. The cost of the combined execution of these instructions varies between 2 heap access + 2 stack access and 1 heap access + 1 stack access. The results of these optimizations are recorded in the third column of fig 7.1. These optimizations do not affect the compiler that generates the sequential G-code. In the experiment described here we have not at all done any static compiler analysis. It would be fair to compare the optimized parallel simulation results with simulation results from an optimized sequential simulation. This depends however from the machine architecture one would choose.

7.2. Arguments

We have counted the number of arguments that are accessed in the heap when running the program. These include arguments that first have to be evaluated before they can be used for some ALU or list constructing operation.

	heap arguments used	*heap accesses (optimized)*
paraffine	33954	78.10^3
hamming	51418	109.10^3
schedule	41326	97.10^3

Fig 7.2 Heap arguments used in relation to heap accesses.

8. Conclusions about the G-line and future work

The efficiency gain seems to be rather independent of the program. The number of accesses to the pointerstack have been reduced to about one third and to the heap to about one fifth. In average the cost of using an argument (including those that consist of 2 fields as floats and cons nodes) in the heap is 2 - 3 accesses. Only slightly more than once

storing a value in the heap and once loading it. The cost of laziness thus has disappeared. We conclude that our architecture performs well.

The use of the graph memory has become an advantage because it offers the possibility of a horizontally coded reducer.

We used infinite hardware in this simulation of the G-line. When hardware is reduced to a feasible size some of the micro parallelism is lost. The following restrictions on hardware can be applied to obtain a realistic machine.

- The number of G-machines is limited. If the size of the subgraph is larger than the number of G-machines, the subgraph is wrapped around the G-line. Some G-machines have to write more than one memory field. If the number of G-machines is N and the size of a subgraph S it now requires (S DIV N) memory cycles to construct a subgraph. We have measured the size of the subgraphs constructed during execution of the programs.

size	*schedule*	*paraffine*	*hamming*
0 - 4	16760	2384	12272
5 - 8	2281	4508	4980
9 - 16	1488	741	0
> 16	0	3	0

Fig 8.1 Size of constructed subgraphs.

Looking at these programs a size of 8 clusters will not cause much loss of parallelism.

- A cluster of tag- left- and right machine can share a number of resources. Only a single valuestack and ALU and a single connection with each bus is needed per cluster. No speed will be lost in this way. Sharing of the pointerstack might cause some loss in parallelism. To construct a cons node or an apply node two different elements of the pointerstack need to be accessed. It costs one more memory cycle in the construction of some of the subgraphs.

8.1. No more replicas

A more radical change of this architecture is realized by splitting the G-line in a global machine and a distributed machine. Making of replicas is then avoided. ALU, valuestack and pointerstack only are present in the global machine. The shared part of the program code resides in the global machine and the MKGRAPH instruction is split in a distributed and a global part. The no-heap instructions only are executed in the global machine, UNWIND and UPDATE in cooperation between global and distributed machine. MKGRAPH is executed in the distributed machine with some support of the global machine. To execute UNWIND, UPDATE and MKGRAPH the local machines contain micro code.

This architecture is subject of current research in which trade offs are made between the amount of micro parallelism and the amount of hardware used. We expect, however, that some loss in parallelism will be outweighed by typical hardware optimizations which generally are not yet included in the design of the G-line.

8.2. To realistic software

We used code generated from an untyped language. The compilation to G-code was straight forward using Johnsson's compilation rules [1]. No compiler optimizations were applied. In our current low level simulations, we use G-code derived from LML, a typed functional language [16]. The G-code generated from typed languages is more efficient than the G-code we used in the experiments described above and better suited to derive parallel microcode.

9. Acknowledgements

We wish to thank Pieter Hartel, Henk Muller and Rutger Hofman for the comments on the draft(s) of this paper.

10. References

[1] T.Johnsson; Efficient compilation of lazy evaluation; Sigplan Notices 19(6):58,69, June 1984.

[2] Simon L.Peyton Jones; Implementation of graph reduction; Prentice Hall; London, 1987.

[3] D.A. Turner, "A new implementation technique for applicative languages"; Software Practice and Experience 9(1) pp 31-49 (A jan. 1979).

[4] J.Fairbairn, S.Wray; TIM: A simple lazy abstract machine to execute supercombinators; Third conference on functional programming and computer architecture; Portland, Oregon, USA; Proceedings 34-35, September 1987.

[5] Guy Argo; Improving the three instruction machine; Proceedings of the FPLCA, pp 100-115, London 1989.

[6] Simon L.Peyton Jones, Jon Salkild; The spineless tagless G-machine; Proceedings of the FPLCA, pp 184-201, London 1989.

[7] L.Augustsson, T.Johnsson; Parallel graph reduction with the <nu,G>-machine; Proceedings of the FPLCA,pp 202-213, London 1989.

[8] W.R.Stoye, T.J.W.Clarke, A.C.Norman; Some practical methods for rapid combinator reduction; Proc. of 1984 ACM Conf on Lisp and Functional Prog, pp 159-166, aug 1984.

[9] M.Scheevel, NORMA, a normal-order combinator reduction machine, colloquium presented at Oregon Graduate Center, july 1984.

[10] R.B.Kieburtz; The G-machine, a fast graph-reduction evaluator; Second Conference on functional Languages and Computer Architecture, LNCS 201, Nancy, 1985; Springer Verlag.

[11] R.B.Kieburtz; A Risc architecture for symbolic computation; Sigplan Notices 22(10):146-155, October 1987.

[12] Willem Vree, Design considerations for a parallel reduction machine; Ph. D. thesis, pp 27-50, Amsterdam, 1989

[13] R.B.Kieburtz and B. Agapiev; Optimizing the evaluation of suspensions, pp 267 - 282 in Proceedings of the Workshop of functional languages, ed. T.Johnsson,

S.L.Peyton Jones, K.Karlsson, Dept of Computer Science, Chalmers Univ. of Technology, Göteborg, Aspensas, 1988.

[14] P.H.Hartel; Performance of lazy combinator graph reduction; To be published in "Software practice and experience".

[15] W.G Vree, P.H.Hartel; Parallel graph reduction for divide-and conquer applications; Part II -program performance; Internal Report D-20, PRM-Project, December 1988.

[16] T.Johnsson; Compiling Lazy Functional Languages, Ph.D. thesis, Göteborg, 1987.

Appendix

SPECIFICATION OF THE G-LINE

M: local machine, o: output, c: codestack, s: pointerstack, v: valuestack, G: graph node, E: environment, D: dump, H: local heappointer.

$\langle M_m,o,EVAL.c,n.s,v,G_a[n=int,i,-],Bus,E,D,H\rangle$ ->
$\langle M_a,o,EVAL.c,n.s,v,G,Bus[int,i,-],E,D,H\rangle$ ->
$\langle M_m,o,c,s,v,G,Bus[int,i,-],E,D,H\rangle$
(similarly for booleans)

$\langle M_m,o,EVAL.c,n.s,v,G_a[n=app,n_l,n_r],E,D,Bus,\rangle$ ->
$\langle M_a,o,EVAL.c,n.s,v,G,Bus[app,n_l,n_r],E,D,H\rangle$ ->
$\langle M_m,o,UNWIND.(),n_r.n.(),v,G,Bus[app,n_l,n_r],E,(c,s).D,H\rangle$

$\langle M_m,o,UNWIND.c,n_j.n_{j-1}...n_0.(),v,G,Bus[app,n_l,n_r],E,(c,s).D,H\rangle$ ->
$\langle M_m,o,UNWIND.(),n_r.n_j.n_{j-1}...n_0.(),v,G,Bus[app,n_l,n_r],E,(c,s).D,H\rangle$ ->
$\langle M_a,o,UNWIND.(),n_r.n_j.n_{j-1}...n_0.(),v,G_a[n_l=tag,left,right],E,D,Bus[tag,left,right],H\rangle$

$\langle M_m,o,UNWIND.(),n_j.n_{j-1}...n_0.(),v,G,Bus[fun\ k,\ c'],E,(c,s).D,H\rangle$ ->
$\langle M_m,o,c',s',v,G,Bus[fun\ k,\ c'],E,D,H\rangle$ *k=j+1*

$\langle M_m,o,UNWIND.(),n_j.n_{j-1}...n_0.s,v,G,Bus[fun\ k,\ c'],E,(c,s).D,H\rangle$ ->
$\langle M_m,o,EVAL.c,s,v,G,Bus[fun,c',k],E,D,H\rangle$ *k>j+1*

$\langle M_m,o,UNWIND.(),n_j.n_{j-1}...n_0.(),v,G,Bus[fun\ k,\ c'],E[fun\ c',k],(c,s).D,H\rangle$ ->
$\langle M_m,o,REWIND.c',n_j.n_{j-1}..n_0.(),v,G_a[n_0=app,p_l,p_r],Bus[app,p_l,p_r],E[fun\ c',k],(c,s).D,H\rangle$
k<j+1

$\langle M_m,o,REWIND.c,n_j...n_{j-k}..n_1.n_0.s,v,G,Bus[app,p_{le},p_{ri}],E[fun\ c',k],D\rangle$ ->
$\langle M_m,o,REWIND.c',n_j..n_{j-k}..n_{le}.n_0.s,v,G,Bus,E[fun\ c',k],D\rangle$
$\langle M_a,o,REWIND.c',n_j.n_{j-1}...n_{le}.n_0.s,v,G_a[n_{le}=app,p_l,p_r],Bus[app,p_l,p_r],E[fun\ c',k],D,H\rangle$ ->
k>j+1

$\langle M_m,o,REWIND.c',n_j...n_{j-k}..n_0.s,v,G,Bus[app,p_l,p_r],E[fun\ c',k],D\rangle$ ->
$\langle M_m,o,c',n_j.n_{j-1}...n_0.s,v,G,Bus[app,p_l,p_r],E[fun\ c',k],D,H\rangle$
k=j+1

$\langle M_m,o,PRINT.c,n.s,v,G_a[n=int,i,-],Bus,E,D,H\rangle$ ->
$\langle M_a,o.i,c.PRINT,n.s,v,G,Bus[int,i,-],E,D,H\rangle$ ->
$\langle M_m,o.i,c,s,v,G,Bus[int,i,-],E,D,H\rangle$
(similarly for booleans)

<M_m,o,PRINT.c,n.s,v,G_a[n=cons,p_l,p_r],Bus,E,D,H> ->
<M_a,o.i,c.PRINT,n.s,v,G,Bus[cons,p_l,p_r],E,D,H> ->
<M_m,o,PRINT.EVAL.PRINT.EVAL.c,p_l.p_r.s,v,G,Bus[cons,p_l,p_r],E,D,H>

<M_m,o,UPDATE k.c, n_0...n_k.s,G_a[n_0=tag,left,right],Bus,E,D,H> ->
<M_a,o,UPDATE k.c, n_0...n_k.s,G,Bus[tag,left,right],E,D,H> ->
<M_m,o,c, n_1...n_k.s,G,Bus[tag,left,right],E,D,H> ->
<M_a',o,c, n_1...n_k.s,G_a'[n_k=tag,left,right],Bus[tag,left,right],E,D,H>

<M_m,o,RET n.c,n_1...n_n.n_0.s,G,Bus[tag,left,right],E,D,H> ->
<M_m,o,RET n.c,n_0.s,G_a[n_0=tag,left,right],Bus[tag,left,right],E,D,H> ->
<M_m,o,c,n_0.s,G,Bus[tag,left,right],E,D,H> *tag ≠ app,fun*

<M_m,o,RET n.c,n_1...n_n.n_0.s,G,Bus[tag,left,right],E,D,H> ->
<M_k,o,RET n.c,n_0.s,G_0[n_0=tag,left,right],Bus[tag,left,right],E,D,H> ->
<M_m,o,EVAL.c,n_0.s,G,Bus,E,D,H> tag = app,fun

<M_m,o,PUSH k.c,n_0....n_k.s,v,G,Bus,E,D,H> ->
<M_m,o,c,n_k.n_0..n_k.d.s,v,G,Bus,E,D,H>

<M_m,o,MKINT.c,s,i.v,G,E,D,Bus,H> ->
<M_0,o,c,n.s,i.v,G_0[n=int,i,-],Bus[int,i,-],E,D,H[lhp]> ->
<M_m,o,c,lhp.s,v,G,Bus,E,D,H[lhp+1]>
(similarly for booleans)

<M_a,o,MKGRAPH_copy n.c,s,v,G,Bus,E,D,H[lhp]> ->
<M_a,o,c,lhp.s,v,G[..,n,..],Bus,E,D,H[lhp+1]>

<M_a,o,MKGRAPH_heap n.c,s,v,G,Bus,E,D,H[lhp]> ->
<M_a,o,c,lhp.s,v,G[..,n.lhp,..],Bus,E,D,H[lhp+1]>

<M_a,o,MKGRAPH_stack n.c,p_1..,p_n.s,v,G,Bus,E,D,H[lhp]> ->
<M_a,o,c,lhp.p_1..,p_n.s,v,G[..,p_n,..],Bus,E,D,H[lhp+1]>

<M_a,o,MKGRAPH_nop .c,s,v,G,Bus,E,D,H[lhp]> ->
<M_a,o,c,lhp.s,v,G,Bus,E,D,[lhp+1]>

<M_m,o,ALLOC.c,s,v,G,Bus,,E,D,H[lhp]> ->
<M_0,o,c,lhp.s,G_0[hole,_,_],Bus,E,D,H[lhp+1]>
<M_m,o,c,n_k..n_1.s,G,Bus,E,D,H> *m≠ 0*

<M_m,o,PUSH k.c,n_0....n_k.s,v,G,Bus,E,D,H> ->
<M_m,o,c,n_k.n_0..n_k.d.s,v,G,Bus,E,D,H>

<M_m,o,c.GET.c,n.s,v,G_a[n=int,i,-],E,D,Bus,E,D,H> ->
<M_a,o,GET.c,n_a.s,v,G,Bus[int,i,-],E,D,H> ->
<M_m,o,c,s,i.v,G,Bus[int,i,-],E,D,H>
(similarly for booleans)

<M_m,o,ADD.c,s,i_2.i_1.v,G,Bus,E,D,H> ->
<M_m,o,c,s,i_2+i_1.v,G,Bus,E,D,H>
(similarly for all arithmetic and logical operations)

<M_m,o,JFALSE .c,s,true.v,G,Bus,E,D,H> ->
<M_m,o,JMP.c,s,v,G,Bus,>

<M_m,o,JMP l,....LABEL l.c,s,v,G,Bus,E,D,H> ->
<M_m,o,LABEL.c,s,v,G,Bus,E,D,H>

<M_m,o,LABEL l.c,s,v,G,Bus,E,D,H> ->
<M_m,o,c,s,v,G,Bus,E,D,H>

<M_m,o,HD.c,n.s,v,G_a[n=cons,p_1,p_2],E,D,Bus,E,D,H> ->
<M_a,o,HD.c,n,s,v,G,Bus[cons,p_1,p_2],E,D,H> ->
<M_m,o,c,p_1.s,v,G,Bus[cons,p_1,p_2],E,D,H>

<M_m,o,TAIL.c,n.s,v,G_a[n=cons,p_1,p_2],E,D,Bus,E,D,H> ->
<M_a,o,TAIL.c,n.s,v,G,Bus[cons,p_1,p_2],E,D,H> ->
<M_m,o,c,p_2.s,v,G,Bus[cons,p_1,p_2],E,D,H>

<M_m,o,c.NULL,n.s,v,G_a[n=cons,p_1,p_2],Bus,E,D,H> ->
<M_a,o,NULL.c,n.s,v,G,Bus[cons,p_1,p_2],E,D,H> ->
<M_m,o,c,s,false.v,G,Bus[cons,p_1,p_2],E,D,H>

<M_m,o,c.NULL,n.s,v,G_a[n=nil,-,-],Bus,E,D,H> ->
<M_a,o,NULL.c,n_a.s,v,G,Bus[nil,-,-],E,D,H> ->
<M_m,o,c,s,true.v,G,Bus[nil,-,-],E,D,H>

Remarks

1)'Bus' denotes the contents of all three buses. Mind that the contents of the buses on entrance or on exit of the rule sometimes is denoted. E.g on exit of EVAL this is the case, so after executing this instruction the next one can be sure the result of EVAL can be read from the buses. UNWIND makes use of this.

2) In G_a[n=_,_,_] the subscript of G_a specifies the machine (and local graph) in which n is pointing.

3) The transition rules contain no optimizations like contraction of MKINT and UPDATE

4) We have separately defined MKINT etc and ALLOC. They could be special cases of the MKGRAPH instruction.

5) An special instruction REWIND is introduced in this diagram to clarify the action in case of unwinding a spine with number of arguments smaller than the depth of the spine.

The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes

(Extended Abstract)

*Gerard Tel**
Department of Computer Science, Utrecht University,
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands.
(Email: gerard@cs.ruu.nl)

Friedemann Mattern
Department of Computer Science, Kaiserslautern University,
P.O. Box 3049, D 6750 Kaiserslautern, Fed. Rep. Germany.
(Email: mattern@informatik.uni-kl.de)

Abstract. It is shown that the termination detection problem for distributed computations can be modeled as an instance of the garbage collection problem. Consequently, algorithms for the termination detection problem are obtained by applying transformations to garbage collection algorithms. The transformation can be applied to collectors of the "mark–and–sweep" type as well as to reference counting garbage collectors. As an example, the scheme is used to transform the weighted reference counting protocol.

1 Introduction

A substantial amount of the research efforts in distributed algorithms design has been devoted to the problem of detecting when a distributed computation has terminated. There are several reasons for the impressive number of publications on this subject. First, as the problem has shown up under varying model assumptions and there are several solutions for each model, a really large number of different algorithms has emerged. All these algorithms were published separately, because unifying approaches, treating a number of algorithms as a class, have been rare. Second, the problem of termination detection, being sufficiently easy to define and yet non–trivial to solve, has been seen as a good candidate

*The work of this author was supported by the ESPRIT II Basic Research Actions Program of the EC under contract no. 3075 (project ALCOM).

to illustrate the merits of design or proof methods for distributed algorithms. Third, it has been observed that the fundamental difficulties of the termination detection problem are the same as those of other problems in distributed computing. Termination detection algorithms are related to algorithms for computing distributed snapshots [CL85], and detecting deadlocks [CMH83]. Thus the problem is seen to be important both from a practical, algorithmical, and from a theoretical, methodological point of view.

From both points of view we consider it useful to recognize general design paradigms for distributed termination detection algorithms. One such paradigm was described in [Te90]. A new paradigm is presented in this paper: it is shown that termination detection algorithms are obtained as suitable instantiations of garbage collection algorithms. A connection between the two problems was pointed out before. Tel, Tan, and Van Leeuwen [TTL88] have shown that garbage collection algorithms (of the so–called *mark–and–sweep* type, see section 1.2) can be derived from termination detection algorithms. Using a different transformation, garbage collection algorithms of the *reference counting* type can also be derived from termination detection algorithms, see section 4.1. The results in this paper further strengthen this connection by presenting a transformation in the reverse direction.

Subsections 1.1 and 1.2 introduce the termination detection problem and the distributed garbage collection problem. Section 2 describes how the termination detection problem can be formulated as garbage collecting one hypothetical object and derives the algorithmical transformation. Section 3 provides an example of the transformation. Section 4 contains some additional remarks and comments.

1.1 The Termination Detection Problem

The problem of termination detection is described formally as follows. A collection $\mathbf{P}$ of *processes* is considered, communicating by message passing. For the sake of simplicity it is assumed that $\mathbf{P}$ is a fixed collection, but the results in this paper are easily generalized to handle process creation and deletion as well. A process is either *passive* or *active*. *Active* processes can send messages, but *passive* processes cannot. An *active* process can spontaneously become *passive*, but a *passive* process can become *active* only on receipt of a message. Formally, the allowed actions of the processes are described as follows. (In all programs to follow, actions are *atomic* and braces ("{" and "}") enclose a guard for an action.)

$\mathbf{S}_p$: { $state_p = active$ }
send a message $\langle M \rangle$

$\mathbf{R}_p$: { A message has arrived }
receive message $\langle M \rangle$; $state_p := active$

$\mathbf{I}_p$: { $state_p = active$ }
$state_p := passive$

Define the *termination condition* as:

No process is *active* and no messages are in transit.

When processes behave as described, this condition is stable: once true, it remains so. The problem of termination detection now is to superimpose on the described *basic* computation a *control* computation which enables one or more of the processes to detect when the termination condition holds. To this end a new special state *terminated* is introduced for each process. The following two criteria specify the correctness of the control algorithm.

T1 *Safety.* If some process is in the *terminated* state then the termination condition holds.

T2 *Liveness.* If the termination condition holds, then eventually some process will be in the *terminated* state.

A *passive* process may take part in this control computation, and receiving control messages does not make a *passive* process *active*.

Solutions to the termination detection problem are non-trivial, mainly due to the possibility that a process becomes *active* after being observed as *passive* by the control algorithm. Several classes of solutions are known. The most important ones are those based on *probes* and those based on *acknowledgements*. The best known example of the former class is [DFG83], and a general treatment is given in [Te90]. Examples of the latter class are [DS80, SF86]. Solutions based on counting sent and received messages are proposed in [Ma87].

1.2 The Distributed Garbage Collection Problem

As our approach for deriving termination detection algorithms is based on solutions to the garbage collection problem, we shall now describe this problem in a model which is close to the model of Lermen and Maurer [LM86]. The advantage of this model is that it abstracts from aspects which are not relevant to our purposes, such as processors, memory cells, and the difference between "local" and "remote" references.

An (object–oriented) distributed system consists of a collection **O** of cooperating *objects*. A subset of **O** is designated as *root objects*. Objects are able to hold *references* to other objects. These references can be transmitted in messages, see below. A reference to an object r will be called an r–reference. An object r is a *descendant* of q if q holds an r–reference or a message containing an r–reference is in transit to q. An object is *reachable* if it is a root object or a descendant of a reachable object. An object p holding an r–reference may *delete* it, after which p no longer holds this reference. Also, a reachable object p holding an r–reference may *copy* the reference to another object q, by sending an r–reference in a message to q. Object q will hold an r–reference after receipt of this message. An object can have multiple references to the same target object.

An object is called *garbage* if it is not reachable. As only reachable objects copy references, only references to reachable objects are copied, and thus a garbage object remains garbage forever. For reasons of memory management it is required that garbage objects are identified and collected. This task is taken care of by a garbage collecting algorithm. The following two criteria define the correctness of a garbage collecting algorithm.

G1 *Safety.* If an object is collected, it is garbage.

G2 *Liveness.* If an object is garbage, it will eventually be collected.

Many solutions have been proposed to the distributed garbage collection problem, most of which fall into one of two categories: collectors of the *reference counting* type and collectors of the *mark–and–sweep* type. Both types of solutions have been known for over 30 years for classical, non–distributed systems [Co60, McC60].

Collectors of the first type [LM86, WW87, Be89] maintain for each non–root object a count of the number of references in existence to that object. References in other objects as well as references in messages are taken into account. The reference count is incremented when a corresponding reference is copied, and decremented when such a reference is deleted. When the count for an object drops to zero, it can be concluded that the object is garbage and consequently the object can be collected. Reference counting garbage collectors are unable to collect *cyclic garbage* (a collection of garbage objects pointing to each other). As will be seen at the end of section 2, this does not render our transformation invalid for reference counting garbage collectors.

Collectors of the second type [Dij78] mark all reachable objects as such, starting from the roots and recursively marking all descendants of marked objects. In this way all reachable objects become marked eventually. The design of the marking algorithm is complicated by the possibility that references are copied and deleted during its operation. The objects must cooperate with the marking algorithm, e.g., by also marking objects

when references are copied, cf. [Dij78]. When the marking phase is terminated a sweep through all objects is made, in which all unmarked objects are collected. These two phases are repeated as long as necessary.

2 Termination Detection Using Garbage Collection

In this section we describe how the termination detection problem in general can be modeled as an instance of the garbage collection problem. As a result, solutions to the termination detection problem can be derived from garbage collection algorithms, of which an example will be shown in section 3. First the collection **O** of objects used for this purpose is described, as well as the behavior of these objects. Next it is shown that the termination condition is equivalent to one particular object becoming garbage, so termination can be detected by a garbage collection algorithm.

Recall that **P** is the set of processes whose termination is to be detected. The collection **O** of objects consists of one root object A_p for every process p in **P**, and a single *indicator object* Z. Object A_p mimics the behavior of process p as far as the basic computation is concerned (it sends and receives p's basic messages, and has all the variables p has). Messages may contain a reference, in which case the message is a copy message for that reference. Object A_p is called *passive* (*active*) when the mimicked process p is *passive* (*active*). As A_p is a root object, it is always reachable.

The indicator object Z is not a root object. Its only purpose is to indicate the termination condition with its reachability status by the following equivalence, which will be maintained during execution.

(IND) Z is garbage $\Leftrightarrow$ the termination condition holds.

Theorem 2.1 *IND holds when the following two rules are observed:*

R1 *An object holds a Z–reference if and only if it is active.*

R2 *Each message of the basic computation contains a Z–reference.*

Proof. Z is garbage is equivalent to: Z is not a descendant of any of the A_p. By definition, this means that no A_p holds a Z–reference, and to no A_p a message is in transit containing a Z–reference. By R1 and R2 this is equivalent to: no A_p is *active* and to no A_p a message (of the basic computation) is in transit. This is the definition of the termination condition. □

It remains to be shown how R1 and R2 can be maintained. It is possible to ensure through proper initialization that R1 and R2 hold initially. To this end, assume that *active* objects are initialized with the necessary Z–reference, and *passive* objects without it, and that messages in transit initially contain the reference also. To maintain R1 and R2 during the distributed computation, each transmission of a message copies the Z–reference, and processes delete their Z–reference when they become *passive.* More explicitly, the actions to be carried out by A_p are modified as follows:

$\mathbf{S}_p$: { $state_p = active$ }
send a message $\langle M, Z \rangle$

$\mathbf{R}_p$: { A basic message has arrived }
receive message $\langle M, Z \rangle$; $state_p := active$;
insert Z in the references of A_p

$\mathbf{I}_p$: { $state_p = active$ }
$state_p := passive$;
delete Z from the references of A_p

With these modifications R1 and R2 are maintained indeed. R1 is maintained because Z–references are deleted in action $\mathbf{I}_p$, and inserted in action $\mathbf{R}_p$. The latter is possible because the message contains a Z–reference by R2. R2 is maintained because in action $\mathbf{S}_p$ a Z-reference is included in every message. This is possible because only *active* objects send messages, and these objects contain a Z–reference by R1. Thus R1 and R2 are maintained during computation, and by theorem 2.1 IND holds. To arrive at a termination detection algorithm, superimpose upon the objects as described a garbage collection algorithm to detect that Z is garbage. The garbage collection algorithm is then modified so as to inform the objects A_p when Z is identified as garbage. (On receiving this notice, the root objects enter the *terminated* state. We omit this (trivial) operation from the description of the algorithms that will follow.)

Theorem 2.2 *The algorithm as constructed satisfies conditions T1 and T2.*

Proof. Assume any process enters the *terminated* state. This happens upon notice that Z is collected. By the correctness of the garbage collection algorithm (condition G1) this implies that Z is garbage. By IND the termination condition holds.

Assume the termination condition holds. By IND, Z is garbage, hence, by the liveness of the garbage collector (condition G2) Z will eventually be collected. Notice of this will be sent to the processes, and these will enter the *terminated* state in finite time. □

Garbage collectors of the reference counting type are not able to collect cyclic structures of garbage, which may possibly harm the liveness of the termination detection algorithm. It is, however, easily seen that Z is not part of such a cyclic structure, and in fact the following, stronger equivalence holds.

There are no references to $Z \Leftrightarrow$ the termination condition holds.

Summary of the transformation. The construction of a termination detection algorithm is summarized in the following four steps.

1. Form the set $\mathbf{O}$ of objects, consisting of the root objects A_p and one indicator object Z.
2. Superimpose upon the actions of the basic computation the handling of the Z–reference.
3. Superimpose upon this combined algorithm a garbage collection algorithm.
4. Replace the collection of Z (or its identification as garbage) by a notification of termination.

3 An Example of the Transformation

The transformation described in section 2 can in principle be applied to any garbage collection scheme, of the reference counting as well as the mark–and–sweep type, or working according to other principles. In this section we consider the transformation of a garbage collection algorithm based on weighted reference counting. The resulting termination detection algorithm turns out to be an already known algorithm: it was proposed in [Ma89]. More derivations, yielding new and non–trivial termination detection algorithms, are found in the full paper [TM90].

In a weighted reference counting scheme, each reference has an associated positive weight. Each object o maintains a reference count, which equals (barring certain update delays) the total weight of existing o–references. (The term "reference weight accumulator" might be more appropriate for this variable, but in accordance with the existing literature we shall continue to use the word "count".) When a reference is copied, its weight is *split* among the existing and the new reference. Thus, although the *number* of references increases, the *weight* remains the same, and the reference count need not be incremented and no message need be sent to the referenced object. When an object deletes an r–reference, a decrement message is sent to r, reporting the weight of the deleted

reference. Upon receipt of this message, r subtracts this weight from its reference count (and is collected if the count drops to zero).

3.1 Description of the Scheme

Distributed weighted reference counting schemes have been given by Watson and Watson [WW87], Bevan [Be89], and others. In the description below the mechanism to create new objects is omitted, because in the transformation no new objects are ever created. An o–reference is a tuple (o, w), where w denotes the weight of the reference. Initially for each non–root object o, the reference count RC_o equals the sum of the weights of all existing o–references. The following (atomic) actions can take place. ($\mathbf{CR}_p$ represents the sending of a copy message, $\mathbf{RR}_p$ the receipt of such a message, $\mathbf{DR}_p$ the deletion of a reference and the associated sending of a decrement–weight message, and $\mathbf{RD}_o$ the receipt of such a message.)

$\mathbf{CR}_p$: { p holds reference (o, w) }
send $\mathbf{cop}(o, w/2)$ to q ; $w := w/2$

$\mathbf{RR}_p$: { A message $\mathbf{cop}(o, w)$ has arrived at p }
receive $\mathbf{cop}(o, w)$;
if p has an o–reference
then add w to its weight
else insert the o–reference with weight w

$\mathbf{DR}_p$: { p holds reference (o, w) }
send $\mathbf{dec}(o, w)$ to o ; delete the o–reference

$\mathbf{RD}_o$: { A $\mathbf{dec}(o, w)$ message has arrived at o }
receive $\mathbf{dec}(o, w)$; $RC_o := RC_o - w$;
if $RC_o = 0$ **then** collect o

Action $\mathbf{RR}_p$ guarantees that in this scheme an object has at most one reference to each other object. A correctness proof and analysis of the scheme is found in [WW87] or [Be89] and is based on invariance of the following two assertions:

1. Each reference has a positive weight; each delete message contains a positive weight.
2. $RC_o = \sum_{R=(o,w)} w + \sum_{D=\mathbf{dec}(o,w)} w$, where R ranges over all o–references in existence (including **cop** messages) and D ranges over all delete messages in transit.

3.2 Transformation into a Termination Detection Algorithm

To transform the garbage collection scheme into a termination detection algorithm we apply the four–step construction of section 2.

1. The set **O** of objects consists of the objects A_p and the indicator object Z.

2. Superimpose upon the actions of the basic computation the handling of the Z–reference. This yields the following program text.

   S$_p$: { $state_p = active$ }
   send a message $\langle M, Z \rangle$

   R$_p$: { A basic message has arrived }
   receive message $\langle M, Z \rangle$; $state_p := active$;
   insert Z in the references of A_p

   I$_p$: { $state_p = active$ }
   $state_p := passive$;
   delete Z from the references of A_p

3. Superimpose the reference counting scheme upon these actions. To this end, action **CR**$_p$ is included in action **S**$_p$, action **RR**$_p$ is included in action **R**$_p$, and action **DR**$_p$ is included in action **I**$_p$. For o the object Z is substituted. This results in the following program text.

   S$_p$: { $state_p = active$ and p holds reference (Z, w) }
   send a message $\langle M, \mathbf{cop}(Z, w/2) \rangle$; $w := w/2$

   R$_p$: { A basic message has arrived }
   receive message $\langle M, \mathbf{cop}(Z, w) \rangle$; $state_p := active$;
   if p has a Z–reference
   then add w to its weight
   else insert the Z–reference with weight w

   I$_p$: { $state_p = active$ }
   $state_p := passive$;
   send $\mathbf{dec}(Z, w)$ to Z ; delete the Z–reference

RD$_Z$: { A **dec**(Z, w) message has arrived at Z }
receive **dec**(Z, w) ; $RC_Z := RC_Z - w$;
if $RC_Z = 0$ **then** collect Z

4. Replace the collection of Z by a notification of termination. Some more simplifications can be made in addition: the actual handling of the Z reference can be removed; instead we equip every process p with a variable W_p, representing the weight of p's (virtual) Z–reference (0 if p has no such reference). The subscript Z is dropped. This finally results in the following algorithm.

S$_p$: { $state_p = active$ }
send a message $\langle M, W_p/2 \rangle$; $W_p := W_p/2$

R$_p$: { A basic message has arrived }
receive message $\langle M, W \rangle$; $state_p := active$;
$W_p := W_p + W$

I$_p$: { $state_p = active$ }
$state_p := passive$;
send **dec**(W_p) to Z ; $W_p := 0$

RD: { A **dec**(W) message has arrived at Z }
receive **dec**(W) ; $RC := RC - W$;
if $RC = 0$ **then** send **term** to all A_p

The initial conditions for this algorithm are: $W_p = 0$ if p is *passive*; $W_p > 0$ if p is *active*; $RC = \sum_p W_p$; and no messages are in transit. (Or, if there are messages, RC correctly reflects their weight.)

The termination detection algorithm that has just been derived is known as the *Credit Recovery* algorithm [Ma89]. The algorithms discussed in this section face the problem of so–called *weight underflow*. When weights are represented in a finite number of bits, there exists a smallest positive value a weight can take, and it is not possible to split this weight in two positive parts. Furthermore, the accumulation of small fragments may cause problems. Solutions to these problems and variants of the scheme may be found in [Be89, Ma89].

4 Conclusions

In this paper we have presented a transformation of garbage collection schemes into termination detection algorithms. Applying the transformation to the weighted reference counting scheme, we have derived the Credit Recovery algorithm for termination detection. Virtually all garbage collection schemes can be transformed into sensible termination detection algorithms. The full paper [TM90] contains derivations of more termination detection algorithms, including three new ones: the Activity Counting algorithm, the Generational termination detection algorithm, and a "dual–tour" token algorithm for a ring of processes. It also contains a discussion of several related aspects, of which we only sketch two here.

4.1 Reverse Transformation

It is also possible to transform a termination detection algorithm into a reference counting garbage collection scheme. The aim of a reference counting algorithm is to collect an object o when all o–references (in objects) have been deleted and no more o–references are in transit (in copy messages).

An object is defined to be *o–active* if it holds an o–reference and *o–passive* otherwise, and a message is called an o–activation message if it carries an o–reference. Under these definitions, an *o–passive* object becomes *o–active* only upon receipt of an o–activation message, and only *o–active* objects send o–activation messages. Now the *o–termination condition*, defined as:

No process is *o–active* and no o–activation messages are in transit

is stable and can be detected by a termination detection algorithm. Furthermore

(RT) There are no o–references $\Leftrightarrow$ the o–termination condition holds.

To arrive at a reference counting garbage collection algorithm, a termination detection algorithm is superimposed on the o–reference handling. When o–termination is detected, o is collected. For each object a separate instance of the termination detection algorithm is executed concurrently.

Although this transformation could be applied to any termination detection algorithm, the resulting reference counting garbage collection scheme would not be feasible in all cases. A complete algorithm along these lines, based on the algorithm in [DS80], was proposed by Rudalics [Ru90].

4.2 Deadlock Detection

The termination detection problem is an instance of a class of detection problems in distributed systems. Communication deadlock detection is a generalization where also a part of the network can be terminated. In this problem, for each *passive* process a subset of the processes is determined at the moment it becomes *passive*. The process can become *active* only by receiving a message from a process in this subset. The termination detection problem is obtained, when each process always chooses the full set of processes. We are currently investigating how the approach in this paper can be generalized to derive (mark–and–sweep) deadlock detection algorithms from garbage collection algorithms.

Acknowledgements: We want to thank Martin Rudalics and Jörg Richter for their discussions of the paper and numerous suggestions and comments. We want to thank the Eindhoven Tuesday Afternoon Club and Reinhard Schwarz for their careful revision of the text.

References

[Be89] Bevan, D.I., *An Efficient Reference Counting Solution to the Distributed Garbage Collection Problem,* Parallel Computing 9 (1989) 179–192.

[CL85] Chandy, K.M., L. Lamport, *Distributed Snapshots: Determining Global States of Distributed Systems*, ACM Trans. on Computer Systems 3 (1985) 45–56.

[CMH83] Chandy, K.M., J. Misra, L.M. Haas, *Distributed Deadlock Detection*, ACM Trans. on Computer Systems 1 (1983) 144–156.

[Co60] Collins, G.E., *A Method for Overlapping and Erasure of Lists*, Comm. ACM 3 (1960) 655–657.

[DFG83] Dijkstra, E.W., W.H.J. Feijen, A.J.M. van Gasteren, *Derivation of a Termination Detection Algorithm for Distributed Computations,* Inf. Proc. Lett. 16 (1983) 217–219.

[Dij78] Dijkstra, E.W., L. Lamport, A.J. Martin, C.S. Scholten, E.F.M. Steffens, *On–the–fly Garbage Collection: An Exercise in Cooperation*, Comm. ACM 21 (1978) 966–975.

[DS80] Dijkstra, E.W., C.S. Scholten, *Termination Detection for Diffusing Computations*, Inf. Proc. Lett. 11 (1980) 1–4.

[LM86] Lermen, C.–W., D. Maurer, *A Protocol for Distributed Reference Counting*, ACM Conference on Lisp and Functional Programming, Cambridge, 1986, pp. 343–354.

[Ma87] Mattern, F., *Algorithms for Distributed Termination Detection*, Distributed Computing 2 (1987) 161–175.

[Ma89] Mattern, F., *Global Quiescence Detection Based on Credit Distribution and Recovery,* Inf. Proc. Lett. 30 (1989) 195–200.

[McC60] McCarthy, J., *Recursive Functions of Symbolic Expressions and Their Computation by Machine*, Comm. ACM 3 (1960) 184–195.

[Ru90] Rudalics, M., *Implementation of Distributed Reference Counts*, Technical Report (forthcoming), Research Institute for Symbolic Computation, J. Kepler University, Linz, 1990.

[SF86] Shavit, N., N. Francez, *A New Approach to Detection of Locally Indicative Stability*, in: L. Kott (ed.), *Proceedings ICALP 1986*, Lecture Notes in Computer Science 226, Springer–Verlag, 1986, pp. 344–358.

[Te90] Tel, G., *Total Algorithms*, Technical Report RUU–CS–88–16, Dept. of Computer Science, Utrecht University, 1988. Also in: Algorithms Review 1 (1990) 13–42.

[TM90] Tel, G., F. Mattern, *The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes*, Technical Report RUU–CS–90–24, Dept. of Computer Science, Utrecht University, 1990.

[TTL88] Tel, G., R.B. Tan, J. van Leeuwen, *The Derivation of Graph Marking Algorithms from Distributed Termination Detection Protocols*, Science of Computer Programming 10 (1988) 107–137.

[WW87] Watson, P., I. Watson, *An Efficient Garbage Collection Scheme for Parallel Computer Architectures*, in: J.W. de Bakker, A.J. Nijman, P.C. Treleaven (eds.), Proceedings Parallel Architectures and Languages Europe, vol. II, Lecture Notes in Computer Science 259, Springer–Verlag, 1987, pp. 432–443.

Indirect Reference Counting: A Distributed Garbage Collection Algorithm *

José M. Piquer †
INRIA - École Polytechnique
Domaine de Voluceau, B. P. 105
78153 LE CHESNAY CEDEX (France).
email: piquer@inria.inria.fr

Abstract

This paper exposes a Garbage Collection (GC) algorithm for loosely-coupled multi-processors. The algorithm is based on reference counting and is designed to reclaim distant-pointed objects. It behaves like *weighted reference counting*, using only *decrement* messages. The main advantages are that it never creates indirect cells (so accesses to distant pointed objects are always done in constant time) and it does not need synchronization with the proprietary site of the object. Object migration is also supported with only one *decrement* message, involving only the source and destination sites. On the other hand, two extra fields are needed in every remote pointer and *decrement* messages can be generated in cascades of arbitrary size, making it less predictable.

A first version adapted to a distributed Lisp running on a group of Transputer[1] processors will be presented along with some measures. Each processor has its own local Garbage Collector (of the Mark-Scan type) and the indirect reference counting for remote pointers has been implemented without having to change the local GC.

In this paper the algorithm is presented, it is compared with the *weighted reference counting* garbage collector and some experimental results are shown along with some implementation issues.

Keywords: Garbage Collection, Reference Counting, Distributed Systems.

1 Introduction

Many garbage collecting algorithms have been proposed for distributed systems. They can be classified as Mark-Scan or as Reference Count (excepting some work existing on distributed copying [Ruda 86]). The first ones are rather complex, as they must

*This work was partially supported by the University of Chile, INRIA and the french government.

†Author's current address: Departamento de Computación, Universidad de Chile, Casilla 2777, Santiago, Chile.

[1]Transputer is a trademark of INMOS Ltd.

be "on-the-fly" [Dijk 78] and distributed [Huda 82, Augu 87, Derb 90], adding the termination problem [Dijk 80] and the detection of global states [Chan 85] to the parallelism problem. The reference counting algorithms have the problem of being unable to reclaim cyclic structures (even though they can be extended to do it), but are much simpler than the Mark-Scan ones and do not require synchronization between the collectors and the mutators. We have chosen the reference count scheme mainly for its simplicity and because it could be implemented without modifying the local garbage collectors. In this way we have two completely independent Garbage Collection algorithms for local objects (a Mark-Scan) and for remote pointers (Reference Counting), an idea already advocated in [Fost 89]. Their only interaction is at the end of each local garbage collection, to find which remote pointers have been deleted.

The environment in which the algorithm is designed to run is a loosely-coupled multi-processor system with independent memories, and a reliable point to point message passing system. The message passing is very expensive and remote pointers already have many fields of information concerning the remote site, address, etc. In this kind of machine, we can accept to lose memory if extra messages can be avoided when requiring remote access, or when doing garbage collection. The proposed algorithm requires two extra fields per remote pointer, and one reference count per remote-pointed object, but it guarantees that the remote accesses are always done without indirections and it never requires synchronization with another site. This is important in a language with mutable data such as Lisp, because the maximum number of indirect cells created by algorithms like *weighted reference counting* cannot generally be estimated, as is done for pure functional languages in [Beva 87]. The algorithm is easy to extend to support object migration, which is used by the distributed Lisp to mutate distant objects.

2 The Model

The basic model is composed of the set of processors P and the set of all objects O. Given an object $o \in O$, it resides at a processor $p \in P$: p is called the *owner* of o. The set of all the remote pointers to o is denoted RP(o). It is assumed that each object has one and only one *owner* at a given time (although it may change).

A remote pointer to an object o is called an o-reference[2]. For every object $o \in O$, RP(o) includes all of the existing o-references, including the remote pointers to o contained in messages already sent but not yet received. Only remote pointers are considered in this model, so the local pointers to o are not included in RP(o) and they are not called o-references.

The model requires also that, given an object o residing at site p, every other site can hold at most one o-reference. Obviously, there are no o-references at p. Remote pointers are usually handled this way: if there are multiple pointers at a site to the same remote object o, they pass by a local indirection. This indirection represents for us the only o-reference at the site.

Any garbage collection algorithm must detect the objects which are not remote-pointed from any other site. For this, every remote pointer operation must be con-

[2]The o-reference terminology and the basis of this model were proposed in [Lerm 86].

sidered, and the model includes only four operations:

1. Creation of an o-reference

   A site p, where an object o resides, transmits an o-reference to another site q. This operation does not imply the creation of the object, and happens each time that the *owner* of an object o sends a message with an o-reference to another site.

2. Duplication of an o-reference

   A site q, which already has an o-reference to an object on another site p, transmits the o-reference to a third site r. This operation differs from creation because the *owner* of the object (p) is not involved. So, it does not know that a new o-reference was created.

3. Deletion of an o-reference

   A site q, holding an o-reference to an object located on a distant site p, discards it.

4. Migration of an object o

   A site q, holding an o-reference to an object located at a distant site p, wants to become the new *owner* of the object (with the permission of p). The object o is going to change its *owner*, and every o-reference is going to change along with it, while the *old owner* transforms its local pointer to o into an o-reference to the *new owner*. This is done with a distributed protocol using, for example, multicast or broadcast messages. This migration protocol is independent of the GC algorithm. However, the GC protocol must support asynchronous owner changes, with messages in transit, without losing consistency.

   Existing distributed garbage collection algorithms do not consider this operation. It will be shown how these algorithms can be extended to handle it.

The model abstracts from the remote pointers implementation and the way to decide when to delete an o-reference. On the other hand it distinguishes the o-reference creation from its duplication which pose completely different problems to the reference counting algorithms as is explained later.

The distributed garbage collection algorithm should specify how to decide when an object o can be deleted locally (i.e. how to detect that RP(o) is empty) and what actions are to be performed in the four cases above.

As the model only considers remote pointers, it is presumed that there are local garbage collectors keeping track of the local references and objects. Thus, when an object o can be deleted locally, it is the responsibility of the local garbage collector to do so. The only restrictions on interactions between the distributed and local garbage collectors are that the local collector must never reclaim remote-pointed objects (it is enough to keep them in a local list), and that at the end of each local garbage collection, every o-reference locally reclaimed could be found (to detect the deletion operation).

3 The Existing Reference Counting Algorithms

Existing reference counting algorithms (initially proposed for Lisp systems in [Coll 60]) are based on maintaining, for each object o, the total number of existing pointers to it, called the reference count of o. For this, every object o has an extra field, Ref_cnt(o), which, in the classic algorithm, keeps the reference count of o. In our model, it is the number of elements of RP(o), denoted card(RP(o)).

In the shared memory version, the mutator adjusts every object's Ref_cnt(o) each time a pointer is created, duplicated or deleted. When the reference count reaches zero, the object can be reclaimed, which means that all of the pointers contained in it can be deleted. This scheme does not detect self-referential structures which are not accessible from the outside, but extensions to handle cycles have been designed for purely functional languages[Bobr 80], combinator graph reduction machines[Brow 85] and distributed systems[Beck 86]. However, in this paper only the basic algorithm is considered.

When trying to map the simple reference counting algorithm onto our distributed model, there are new problems and new algorithms. In this section we present two known distributed reference counting algorithms and their problems.

3.1 The Naive Algorithm

The first algorithm obtained from direct extension of the shared memory version is to keep the reference count of each object o equal to the total number of o-references in the system. Note that only remote pointers are considered.

Thus, for every object o we have:

$$\text{card}(\text{RP}(o)) = \text{Ref\_cnt}(o).$$

The actions associated with each operation are:

1. Creation of an o-reference

   When p sends an o-reference to q, it increments locally the Ref_cnt(o).

2. Duplication of an o-reference

   When q duplicates an o-reference, it sends an *increment* message to the owner site p to make it increment Ref_cnt(o).

3. Deletion of an o-reference

   When an o-reference is discarded at site q, it sends a *decrement* message to the owner site p to make it decrement Ref_cnt(o).

4. Migration of an object o

   Migrations are not handled in the original algorithm. However, it is important to see what kind of problems may be encountered if objects can change their *owner*. A simple solution is to perform one distant pointer creation (from the *old owner* to the *new owner*) and every time an o-reference is changed (at the other sites), send a *decrement* message to the *old owner* and an *increment* message to the *new owner*. If there are n o-references in the system, this algorithm uses $2(n-1)$

messages for one migration. This is extremely expensive in messages, but the algorithm can be improved, noting that the reference count of the object remains the same (since no o-reference is created or deleted). Therefore, the object can migrate along with its reference count, without spending any messages.

To consider messages in transit on the network, the invariant must be extended. Denote by DM(o) the set of all the *decrement* messages concerning o not yet received, and by IM(o) the set of the *increment* messages. The extended invariant becomes:

For every object o we have:

$$\text{card}(\text{RP}(o)) = \text{Ref\_cnt}(o) - \text{card}(\text{DM}(o)) + \text{card}(\text{IM}(o)).$$

The garbage collection rule is: "an object o can be deleted when Ref_cnt(o) reaches zero".

The problem of this algorithm is that it does not work correctly if the messages in the network are not guaranteed to be totally time-ordered (which is expensive). When there exists asynchronous communication between sites, messages are only partially ordered as was noted by the well-known paper [Lamp 78].

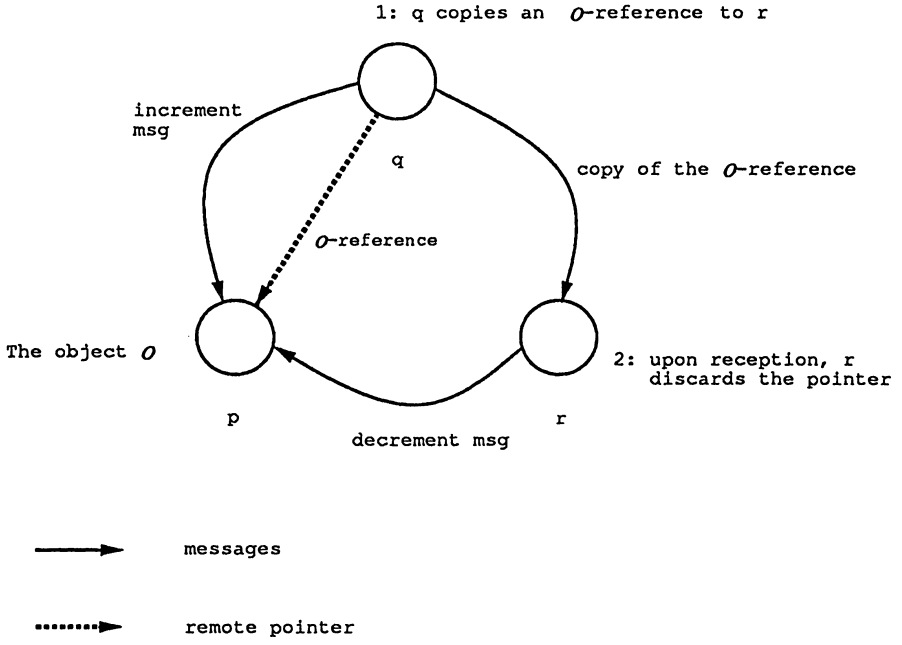


Figure 1: Two contradictory messages on their way to p

The co-habitation of two kinds of messages (*increment* and *decrement*) introduces a problem which invalidates the algorithm. Since it cannot be guaranteed that two

messages sent at different times from different sites will arrive to a third site time-ordered, a *decrement* message can arrive first even if an *increment* message is on its way to the site. This situation can be seen in Figure 1.

The problem exists even if the messages can be guaranteed to be ordered between two given sites, because q can send a copy of an o-reference (for an object residing at p) to r, with r discarding it immediately after reception. In this case there are two messages in transit to p from two different sites and it cannot be known which will arrive first. Obviously, the worst case is the *decrement* message arriving first, which can cause a premature zero of the object's reference count at p. Some other solutions have been proposed ([Lerm 86, Gold 89]) but they use more messages or consume more memory.

Furthermore, the improved migration protocol (migrating the object along with its reference count) is incorrect in an asynchronous environment. Messages affecting the reference count of an object can be in transit while the object is migrating. If using the improved migration algorithm, a site can receive *increment* or *decrement* messages for an object already migrated to another site. This can be handled keeping track of migrated objects and forwarding messages to the *new owner* site. However, this indirection must be maintained until every message has been received, which is a global condition and thus, difficult to detect.

3.2 Weighted Reference Count

A better solution to the distributed reference counting problem is the *weighted reference count* [Beva 87, Wats 87]. This algorithm avoids the problem of the premature zero by simply eliminating the *increment* messages. The algorithm is elegant and simple: each o-reference rp has a *weight* associated with it (noted Weight(rp)) and each object has, as always, a reference count. The difference is that now the Ref_cnt(o) does not keep the number of pointers to o but the sum of their weights. So, the invariant is:

For every object o:

$$\sum_{rp \in \mathrm{RP}(o)} \mathrm{Weight}(rp) = \mathrm{Ref\_cnt}(o).$$

To preserve this invariant, the actions associated with the remote pointer operations are:

1. Creation

   The new o-reference takes an initial weight, say w (with $w > 0$), and the reference count of o is incremented by w. (It is supposed that, at object creation, its reference count is zero.)

2. Duplication

   If o resides at p, and q is sending a copy of the o-reference to r, the original o-reference at q takes half of its old weight and the newly created copy takes the other half (this is why usually the initial value w is a power of two) [3]. In this

[3] In fact, this is an optimization. Theoretically, if w is the old weight of the pointer, it is enough to find two new weights u and v such that $u + v = w$.

way, the invariant is preserved without having to send a message to the *owner*, p.

3. Deletion

   When an o-reference is deleted, a *decrement* message is sent to the *owner* along with the weight of the deleted pointer. Upon reception of a *decrement* message, the reference count is decremented by this amount.

4. Migration

   This operation was not considered in the original papers. When an object o migrates from p to q, we can migrate the object along with its reference count and the new distant pointer (from p to q) can take the weight associated with the old pointer (from q to p). If every pointer is changed at the same time, the invariant is preserved.

In order to consider *decrement* messages in transit, the original invariant must be extended to consider the weights of the *decrement* messages not yet received. Denoting as DM(o) the set of existing *decrement* messages related to remote pointers to o, the extended invariant is:

$$\sum_{rp \in \mathrm{RP}(o)} \mathrm{Weight}(rp) = \mathrm{Ref\_cnt}(o) - \sum_{m \in \mathrm{DM}(o)} \mathrm{Weight}(m).$$

The big problem with this algorithm is the copying of an o-reference with weight 1. One solution is to send an *increment* message to the proprietary site [Beck 86] and wait for an acknowledge, which means a synchronization with this site before proceeding with the local computation. Another solution is to create an *indirection cell* [Wats 87, Beva 87] with a new reference count of w, but the accesses are now done to this cell and must be forwarded to the original object. In a distributed environment, this can create remote indirections which are very expensive, and also affects accesses to the data. Even worse, references can be copied many times from one processor to another, having to create indirection cells many times.

The migration protocol also poses a problem if considering messages in transit: *decrement* messages can arrive at the wrong site (the *old owner*). The *old owner* site can keep track of the migration to forward these messages, but, again, the global condition "every o-reference has been changed" is not easy to detect.

4 Indirect Reference Counting

In this section the proposed algorithm is presented, which is based on avoiding the *increment* messages by maintaining a *distributed reference count*[Piqu 90a].

The *weighted reference* algorithm eliminates the *increment* messages using a new definition of the field Ref_cnt(o) which only decreases. The *indirect reference* algorithm eliminates the *increment* messages by always maintaining enough information at each node to do the increments locally, without any communication. This is true even for duplications.

The basis of the algorithm is to maintain a tree structure representing the diffusion tree of the pointer throughout the system. In fact, this structure is equivalent to

the tree used by [Dijk 80] to detect termination, and we are using it to detect the end of the pointer's diffusion. This structure contains every o-reference and the object itself (which is always the root of the tree). When an o-reference is created or duplicated, a node is added as a child of the creator. When an o-reference is deleted, the corresponding node is deleted from the tree only if it was a leaf. If not, the descriptor is kept in the structure until it becomes a leaf (to avoid a distributed deletion, the leaf case being trivial). Obviously, when the root is the only node in the tree, there are no more o-references in the system.

The implementation is very simple: an inverted tree is used, where each node keeps one pointer to its parent and a counter with the number of children[4]. The structure can be seen in Figure 2.

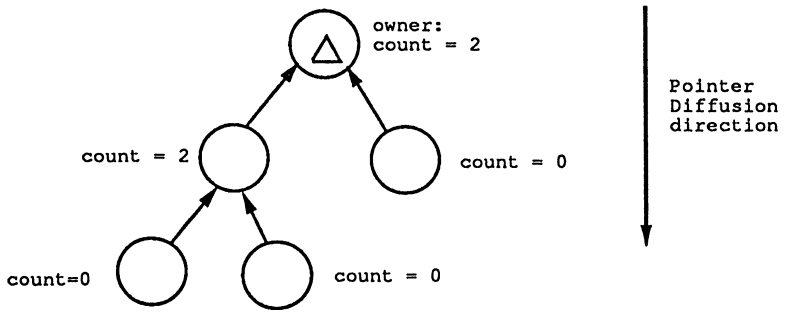


Figure 2: The inverted tree

The nodes of the tree are the o-references themselves, which are extended with two new fields: `copy_cnt` (the number of children) and `parent` (the pointer to the parent in the tree). We suppose that an o-reference always contains the `owner` field (which in our case is a pointer to the root of the tree).
Each remote pointer is a record:

```
remote_pointer = record
        integer owner;          /* owner, root of the tree */
        .                       /* necessary information for accesses */
        .
        integer copy_cnt;       /* number of duplications */
        integer parent;         /* from which we received it */
end;
```

The object o (the root of the tree) has a similar descriptor, where `owner` shows that the local site is the owner, `parent` is `NIL`, and the `copy_cnt` keeps the number of creations done locally. Upon record creation, the `parent` initial value is `NONE`, to distinguish it from the root descriptor (with `parent` value `NIL`).

The invariant of the algorithm is to preserve the tree structure consistently, with only one root, and with the correct number of children. It is easy to see that if the

[4] This is the Indirect Reference Count because, when it reaches zero (the node becomes a leaf), the node can be deleted from the structure if the local o-reference is deleted.

tree is correct, when the copy_cnt of the root is zero, the object can be deleted locally. For every o-reference or object rp at site p, we denote Children(rp, p) the set of all o-references with parent p. At each node p, the invariant is:

$$\text{card}(\text{Children}(rp, p)) = \text{copy\_cnt}(rp).$$

The total number of o-references on the system is equal to the total number of nodes on the tree:

$$\text{card}(\text{RP}(o)) = \sum_{rp \in \text{RP}(o) \bigcup \{o\}} \text{copy\_cnt}(rp).$$

To preserve this structure, we must avoid cycles, so every received o-reference already known is refused, replying with a *decrement* message. The actions performed upon operations on remote pointers are:

1. Creation

   - at p, when a message containing an o-reference is sent:

     ```
     o.copy_cnt = o.copy_cnt + 1;
     ```

   - at q, upon reception of a message with an o-reference rp:

     ```
     if ( rp.parent == NONE )
     {
        rp.parent = p;
        rp.copy_cnt = 0;
     }
     else
        /* the reference already belonged to the tree */
        ''send decrement message to p'';
     ```

2. Duplication (it is as a Creation)

   - at q, when a message with a copy of an o-reference rp is sent:

     ```
     rp.copy_cnt = rp.copy_cnt + 1;
     ```

   - at r, upon reception of a message with a copy of an o-reference rp:

     ```
     if ( rp.parent == NONE )
     {
        rp.parent = q;
        rp.copy_cnt = 0;
     }
     else
        /* the reference already belonged to the tree */
        ''send decrement message to q'';
     ```

3. Deletion

When an o-reference rp is deleted at p, and its `copy_cnt` is zero, it can be deleted from the tree. Leaf deletion is trivial: a *decrement* message is sent to the parent. If the count is not zero, the descriptor must be maintained waiting for other *decrement* messages, but it is marked as *dcleted*:

```
if( rp.copy_cnt == 0)
  {
    ''send decrement message to rp.parent'';
    Delete(rp);
  }
else
    Mark_deleted(rp);
```

Upon reception of the *decrement* message for an object o, the number of children must be decremented. If the count reaches zero, and if the local o-reference was deleted, a *decrement* message is sent to the parent:

```
/* reception of a decrement message for a remote pointer rp */

rp.copy_cnt = rp.copy_cnt - 1;
if (rp.copy_cnt == 0 && Test_deleted(rp))
  {
    ''send decrement message to rp.parent'';
    Delete(rp);
  }
```

An o-reference marked as *deleted* can be unmarked if the same o-reference is received again, before its `copy_cnt` reaches zero. However, a *decrement* message must be sent to the o-reference's sender, because the descriptor already has a valid `parent`. Thus, an `Unmark_deleted` directive must be added to the code handling the reception of any o-reference.

4. Migration

The migration of an object o from p to q means a change of the root in the diffusion tree. This operation is trivial on an inverted tree if the old root is known: the new root (at q) is extracted from the tree (along with its sub-tree) and the old root (at p) is added as a son of the new root. The extraction costs one *decrement* message, and the addition is done locally at the respective nodes (the new and old roots):

- at p, when it changes the owner of the object `o`:

  `o.parent = q;`

- at q, upon reception of a message authorizing it to be the new owner of an object pointed by rp:

```
''send decrement message to rp.parent'';
rp.parent = NIL;
rp.copy_cnt = rp.copy_cnt + 1;
```

The condition to locally delete an object o is simply that the copy_cnt of o equals zero.

This system sends only *decrement* messages, as the increments are always done locally. The sites are informed of the deletion of locally sent o-references, only when the reference and all of its remote duplicated copies have been deleted. Also, accesses to distant objects are always done directly [5].

Migrations are handled neatly, with a simple modification of the tree structure. The migration operation complicates the other GC algorithms because it asynchronously modifies the owner pointers which are used by the *decrement* messages. If the objects were to migrate with their reference counts, messages could arrive at invalid destinations from which they would have to be forwarded. If not, many messages should be used to assure the invariant at any time.

The *indirect reference counting* does not use the owner pointers, but rather the parent pointers, to send its *decrement* messages. However, parent pointers are also modified during migration (the other operations are only allowed to initialize the parent field, not to change it) but the reference counts never move: objects migrate alone and they get the local reference counts. Therefore, any *decrement* message in transit will always arrive at the correct destination (which is the site holding the reference count to be decremented).

A shortcoming of the GC algorithm is that the deletion of an o-reference could generate many messages in cascades of arbitrary size, if the pointer was forwarded from one site to another many times. However, this only happens if all of the copy_cnt's are reaching zero at once.

5 Implementation Issues

The *indirect reference counting* algorithm has been implemented on a distributed Lisp system called TransPive[Piqu 90b], based on Le-Lisp version 15.2 [Chai 84], extended to support remote pointers. In this system, remote pointers can be accessed for writing (for functions such as rplaca) which causes data migration. The current implementation uses the following organization:

- remote pointers

  Remote pointers are implemented adding an extra field on every Lisp object. This field, called descriptor, points to an object descriptor which contains all the information about the object: obj_id is a unique object identifier, owner is the object's owner, and copy_cnt and parent are the extra fields for the garbage collector. This descriptor also exists if the object is local.

[5] The parent is a new field and it is not used upon accesses to the values. The remote pointers always point directly to the site where o resides (as there are no indirection cells) using the owner field.

- object identifier

  Each site has a local hash table with the descriptor of every locally known object. The keys are the unique object identifiers. We have a function `(search_desc obj_id)` which returns a pointer to the descriptor.

- local garbage collectors

  The local GC always invokes a function `gcalarm` [Chai 84] after each execution. This permits us to search in the hash table all the deleted objects. We have a function called `garbagep` to detect descriptors of deleted objects. This descriptors are kept in the hash table until their `copy_cnt` field is zero.

  Also a `GC_list` is constructed to retain every object with a `copy_cnt` greater than zero, to prevent them from being garbage collected (the hash table does not keep a pointer to the object itself, only to the descriptor). Every time an object gets a `copy_cnt` equal to zero, it is deleted from this list.

The Lisp codification of the algorithm is (ignoring semaphores and some details):

```
; every Lisp object has a new field:
; (descriptor obj)

; (descriptor obj) returns a structure with fields:
;       (obj_id desc)       ;    unique object identifier
;       (owner  desc)       ;    owner of this object
;       (copy_cnt desc)     ;    indirect reference count
;       (parent desc)       ;    from which it came to us
;

; Each time a pointer is sent in a message the copy_ptr routine is called.
; This is done both for creation and duplication of a remote pointer

; The actions performed are:
;    add the object to the GC_list (the first time)
;        to prevent the local GC from collecting it
;    increment its copy_cnt

(de copy_ptr (obj)
    (let ((desc (descriptor obj)))
        (when (= (copy_cnt desc) 0)
            (setq GC_list (cons obj GC_list)) )

        (copy_cnt desc (+ (copy_cnt desc) 1)) ))

; After a local GC
(de gcalarm ()
    ; update the GC_list
    (setq GC_list (del_ptrs GC_list))
```

```
    ; update the hash_tab
    (foreach (desc hash_tab)
        (when (and (garbagep desc) (= (copy_cnt desc) 0))
            ; this is a leaf, we can just delete it
            (send_decr_msg (obj_id desc) (parent desc))
            (delete_from_htab hash_tab desc) )))

; delete from the list every object with ref_cnt = 0
(de del_ptrs (list)
    (if (null list)
        ()
        (if (= (copy_cnt (descriptor (car list))) 0)
            (del_ptrs (cdr list))
            (cons (car list) (del_ptrs (cdr list))) )))

; The GC process:

; upon reception of a decrement message
(de decr_msg (obj_id)
    (let ((desc (search_desc obj_id)))
        (copy_cnt desc (- (copy_cnt desc) 1)) ))


; Migrations:

; We are the new owner of the object
(de new_owner (desc)
    (send_decr_msg (obj_id desc) (parent desc))
    (parent desc ())
    (copy_cnt desc (+ (copy_cnt desc) 1)) )

; We are the old owner
(de migrate_to (desc new_owner)
    (parent desc new_owner) )
```

We can see in this implementation that, upon the reception of a *decrement* message, we just perform the decrement. The test to see if the object was locally deleted and its copy_cnt is already zero is done only after local garbage collection (in gcalarm).

6 Measures and Results

We have done some measures on three distributed applications: a merge-sort (using physical modification routines, thus causing migrations and being very inefficient), a search on a graph and a *tic-tac-toe* playing program. The applications were run on four Transputers, excepting the *tic-tac-toe* which was run on five Transputers.

Applic.	Execution Time (s)		
	with GC	no GC	overhead (%)
sort(1000)	4.1/3.1	3.8/2.8	7/9
sort(2000)	7.1/6.0	6.6/5.2	7/14
search	18.9/10.0	18.5/9.7	2/3
tic-tac-toe	11.3	11.2	1

Table 1
The GC overhead

We have measured the total execution time of the same application with or without the GC. The results are shown in Table 1, and we can see that the total time overhead of the GC is, on average, 10%. There are two numbers for each application, the first time and the second time it runs, because the second time the TransPive cache system has kept the data copied locally and so it runs faster. The *tic-tac-toe* application runs at the same speed both times.

The memory available for each Lisp was 1 Megabyte, and each execution generated, on average, one local GC at each node.

7 Conclusions

A new reference counting algorithm has been designed and implemented for distributed systems, based on a distributed tree structure of the pointers to each object, keeping *indirect reference counts* at each node. It spends two extra fields in each pointer (`parent` and `copy_cnt`) and can generate more than one message when a reference is deleted, but in total there is always at most one message per deleted reference. It never creates indirect cells nor needs to synchronize with any other site, and migrations are supported neatly with only one overhead message for the GC.

The algorithm is very simple and has a time overhead of approximatively 10% in the cases studied. Obviously, it is not suited to do garbage collection of local references, as two fields per pointer is unrealistic. However, when only remote pointers are concerned it becomes feasible and very simple to add to any local garbage collection system.

Generating messages in cascades of arbitrary size can be a serious problem if used on real-time systems (which was not our case).

Sharing a typical disadvantage of reference counting algorithms, the algorithm, as presented, is unable to collect distributed cyclic structures.

8 Acknowledgements

I am greatly indebted to Dr. Christian Queinnec who always suggested interesting ideas. This work was made possible thanks to the constant support of INRIA, the University of Chile and the French-Chilean cooperation program.

References

[Augu 87] Lex Augusteijn, *Garbage Collection in a Distributed Environment*, LNCS 259, PARLE Proceedings Vol. I, Eindhoven, Springer Verlag, June 1987.

[Beck 86] M. J. Beckerle, K. Ekanadham, *Distributed Garbage Collection with no Global Synchronization*, IBM Research Report, RC 11667 (#52377), January 1986.

[Beva 87] D. I. Bevan, *Distributed Garbage Collection Using Reference Counting*, LNCS 259, PARLE Proceedings Vol. II, Eindhoven, Springer Verlag, June 1987.

[Bobr 80] D. G. Bobrow, *Managing Reentrant Structures Using Reference Counts*, ACM Trans. on Programming Languages and Systems, Vol. 2, No. 3, pp. 269-273, July 1980.

[Brow 85] D. R. Brownbridge, *Cyclic Reference Counting for Combinator Machines*, Functional Programming Languages and Computer Architecture, LNCS 201, pp. 273-288, Springer Verlag, September 1985.

[Chai 84] J. Chailloux, M. Devin, J. M. Hullot, *Le-Lisp: A Portable and Efficient Lisp System*, Proc. 1984 ACM Symposium on Lisp and Functional Programming, August 1984.

[Chan 85] K. M. Chandy, L. Lamport, *Distributed snapshots: Determining global states of distributed systems*, ACM Trans. on Computer Systems, Vol. 3, No. 1, February 1985.

[Coll 60] G. E. Collins, *A Method for Overlapping and Erasure of Lists*, Comm. of the ACM, Vol. 3, No. 12, pp. 655-657, December 1960.

[Derb 90] M. H. Derbyshire, *Mark Scan Garbage Collection On A Distributed Architecture*, Lisp and Symbolic Computation, Vol. 3, No. 2, pp. 135-170, April 1990.

[Dijk 78] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens, *On-the-fly Garbage Collection: an exercise in cooperation*, Comm. of the ACM Vol. 21, No. 11, November 1978.

[Dijk 80] E. W. Dijkstra, C. S. Scholten, *Termination Detection for Diffusing Computations*, Information Processing Letters, Vol. 11, No. 1, August 1980.

[Fost 89] I. Foster, *A Multicomputer Garbage Collector for a Single-Assignment Language*, Int. Journal of Parallel Programming, Vol. 18, No. 3, 1989.

[Gold 89] B. Goldberg, *Generational Reference Counting: A Reduced-Communication Distributed Storage Reclamation Scheme*, SIGPLAN Conference on Programming Languages Design and Implementation, Portland, Oregon, June 1989.

[Huda 82] P. Hudak, R. M. Keller, *Garbage Collection and Task Deletion in a Distributed Applicative Processing System*, 1982 ACM Symposium on Lisp and Functional Programming, 1982.

[Lamp 78] L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, Comm. ACM, Vol. 21, No. 7, pp. 558-565, July 1978.

[Lerm 86] C. W. Lermen, D. Maurer, *A Protocol for Distributed Reference Counting*, Proc. 1986 ACM Conference on Lisp and Functional Programming, Cambridge, Massachussets, August 1986.

[Piqu 90a] J. M. Piquer, *Un GC parallèle pour un Lisp distribué*, Journées francophones des langages applicatifs, La Rochelle, January 1990. BIGRE 69, July 1990 (french).

[Piqu 90b] J. M. Piquer, *Sharing Data Structures in a Distributed Lisp*, Proc. High Performance and Parallel Computing in Lisp Workshop, Twickenham, London, UK, November 1990.

[Ruda 86] M. Rudalics, *Distributed Copying Garbage Collection*, Proceedings of the ACM Conference on LISP and functional prog., Cambridge, Massachussets, August 1986.

[Wats 87] P. Watson, I. Watson, *An Efficient Garbage Collection Scheme for Parallel Computer Architectures*, LNCS 259, PARLE Proceedings Vol. II, Eindhoven, Springer Verlag, June 1987.

Periodic Multiprocessor Scheduling

Jan Korst[1], Emile Aarts[1,2], Jan Karel Lenstra[2,3] and Jaap Wessels[2]

1. Philips Research Laboratories, P.O. Box 80.000, 5600 JA Eindhoven, the Netherlands
2. Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, the Netherlands
3. CWI, P.O. Box 4079, 1009 AB Amsterdam, the Netherlands

Abstract

A number of scheduling and assignment problems are presented involving the execution of periodic operations in a multiprocessor environment. We consider the computational complexity of these problems and propose approximation algorithms for operations with identical periods as well as for operations with arbitrary integer periods.

Keywords: periodic scheduling, periodic assignment, cyclic scheduling, nonpreemptive scheduling

1 Introduction

This paper deals with the problem of scheduling periodic operations, i.e., operations that have to be repeated at a constant rate over an infinite time horizon. Periodic scheduling problems naturally arise in such diverse areas as real-time processing, process control, vehicle scheduling, personnel scheduling and preventive maintenance scheduling; see Section 2 for references. Our interests in periodic scheduling originate from the field of real-time video signal processing, where the samples of a video signal have to be processed at a constant high frequency (10 - 100 MHz) on a network of processors. Due to the high frequencies, the processing of successive samples necessarily overlaps in time. The intrinsic periodic nature of video signal processing gives rise to a periodic scheduling formulation. This application area poses some specific constraints, resulting in a class of optimization problems that so far have received little attention in the literature. In this paper we discuss this class of problems by examining their computational complexity, introducing approximation algorithms, and indicating relevant results presented in the literature.

We aim to keep the discussion as general as possible by proposing solution strategies that are also applicable in other application areas. Many papers on periodic scheduling are concerned with specific applications, proposing solution strategies that are often strongly tailored to the application at hand, a notable exception being the paper by Serafini & Ukovich [1989], which presents a general mathematical model for periodic scheduling problems. However, their emphasis is on periodic scheduling subject to precedence constraints. In our paper, the emphasis is on periodic scheduling subject to resource constraints. In that respect, our work is complementary to theirs.

The organization of the paper is as follows. Section 2 briefly surveys the literature on periodic scheduling. Section 3 gives a mathematical model of periodic scheduling, from which a number of interrelated optimization problems are derived. The computational complexity of these problems is examined in Section 4. Section 5 gives approximation algorithms and bounds on their worst-case performance, if available. Section 6 contains some concluding remarks.

2 Survey of the Literature

In the literature, the notion 'scheduling' refers to planning in time as well as planning in time and space. In this paper, we take the latter interpretation. We divide the literature on scheduling periodic operations into two main areas of interest, namely

(i) *Periodic Scheduling*: assigning start times and processors to periodic operations so as to minimize the number of processors, possibly subject to precedence constraints, and

(ii) *Periodic Assignment*: assigning processors to periodic operations so as to minimize the number of processors for periodic operations with fixed start times.

Clearly, periodic assignment is a subproblem in periodic scheduling. Next, we briefly describe some results obtained in both areas. We do not aim to give a complete overview.

2.1 Periodic Scheduling

Most of the literature on scheduling periodic operations in time is restricted to preemptive scheduling. Preemptive scheduling allows interruption of an execution on a given processor at some time and its resumption at the same time on a different processor or at a later time on any processor.

2.1.1 Preemptive Periodic Scheduling

Preemptive periodic scheduling problems are usually modelled as follows. Given a set of operations $O = \{o_1, \ldots, o_n\}$, any operation $o_i \in O$ is periodically requested to be executed with a given period $p(o_i)$ between two successive requests of operation o_i. Once requested at time t an execution of o_i is required to be completed at time $t + d(o_i)$, called its deadline. The objective is then to find a feasible schedule that requires a minimal number of processors, where a schedule is called feasible if all deadlines are met. Leung & Merrill [1980] prove that the problem of deciding whether a feasible schedule exists on m processors is NP-complete, even for $m = 1$. However, this problem can be solved in polynomial time if the deadline of each execution coincides with the next request for the operation. For $m = 1$, Liu & Layland [1973] and Labetoulle [1974] prove that, if a feasible schedule exists, then it is obtained by the so-called *deadline driven algorithm*, which is a dynamic-priority algorithm that schedules executions with earliest deadlines as soon as possible. Liu & Layland also give a fixed-priority scheduling algorithm for $m = 1$, known as *rate-monotonic priority assignment*, which is optimal in the sense that the algorithm finds a feasible schedule whenever a feasible fixed-priority schedule exists. Dhall & Liu [1978] present two fixed-priority scheduling algorithms for $m \geq 1$, and discuss their worst-case performance. Leung & Whitehead [1982] study the complexity of preemptive fixed-priority scheduling. Lawler & Martel [1981] show that a feasible preemptive schedule exists if and only if a feasible periodic schedule exists with a period equal to the least common multiple of the periods of the individual operations. Bertossi & Bonuccelli [1983, 1985] consider preemptive scheduling on multiprocessor systems consisting of 'processors of different speeds'. Scheduling periodic operations together with 'sporadic time-critical operations' is examined by Chetto & Chetto [1989].

2.1.2 Nonpreemptive Periodic Scheduling

So far, nonpreemptive periodic scheduling has received little attention in the literature. To schedule periodic operations nonpreemptively, it is usually assumed that the operations have to

be executed with a fixed time between successive executions of the same operation. Gonzalez & Soh propose an optimization algorithm for nonpreemptively scheduling periodic operations for the rather special case that the period of the ith operation is half the period of the $(i+1)$th one. Serafini & Ukovich [1989] discuss nonpreemptive periodic scheduling subject to precedence constraints and show that this problem is NP-complete. Park & Yun [1985] give an ILP formulation of a nonpreemptive scheduling problem. They consider a set of independent periodic operations, where each execution requires a given number of resources during one unit of time, and aim to minimize the maximum required amount of resources. They show how this problem can be partitioned into a set of independent subproblems, which can be optimized independently. The partitioning divides the operations into subsets such that the periods of operations in different subsets are relatively prime. A problem related to nonpreemptive periodic scheduling is the problem of inscribing regular polygons in a circle so as to maximize the minimum distance between two vertices on the circle. Burkard [1986] solves this problem for a set of regular polygons that includes only two different types of polygons. Vince [1989] presents a more general approach to this problem.

2.2 Periodic Assignment

Periodic assignment deals with the problem of assigning the executions of periodic operations to a minimal number of processors, assuming that the executions are fixed in time. As we show in the next sections, this problem is closely related to that of colouring circular arcs. Circular-arc colouring has been studied by several authors. Garey, Johnson, Miller & Papadimitriou [1980] prove that circular-arc graph colouring is NP-hard. Tucker [1975] gives upper bounds on the number of colours needed to colour various types of circular-arc graphs. Orlin, Bonuccelli & Bovet [1981] and Shih & Hsu [1989] give efficient algorithms for the polynomially solvable subproblem of colouring proper circular-arc graphs.

Bartholdi, Orlin & Ratliff [1980] consider the periodic assignment problem under the assumption that the availability of resources is also periodic. This problem naturally arises in the area of personnel scheduling, where periodic jobs have to be assigned to persons having periodic working hours. Bartholdi [1981] proposes a linear programming round-off algorithm and gives its worst-case deviation from optimum. Orlin [1982] discusses the periodic assignment problem under the assumption that processors require a setup time s_{ij} to switch from execution i to execution j. This problem naturally arises in the area of vehicle scheduling, where a vehicle has to be transported from the end point of route i to the starting point of route j before it can start traversing route j.

3 Problem Description

In this section we give a formal description of a number of interrelated periodic scheduling and assignment problems. We restrict ourselves to nonpreemptive scheduling and do not consider precedence constraints.

Let $O = \{o_1, \ldots, o_n\}$ be a set of n periodic operations. For each $o \in O$ an execution time $e(o) \in \mathbb{N}$ and a period $p(o) \in \mathbb{N}$ are given. We assume that $p(o) \geq 1$ and $e(o) \leq p(o)$ for each $o \in O$. Once an execution of an operation o is started at a time unit $t \in \mathbb{Z}$, it is completed without interruption on the same processor. Note that in this paper time is measured in time units, i.e., time periods of equal length. If an operation o with execution time $e(o)$ is said to

start at time unit t, it starts at the beginning of time unit t and completes at the end of time unit $t + e(o) - 1$. Similarly, a time interval $[t_1, t_2]$ denotes a set of consecutive time units, given by $\{t_1, t_1+1, \ldots, t_2\}$. The kth execution of operation o is denoted by $o[k]$. If execution $o[k]$ is started at time unit t, then execution $o[k+1]$ is started at time unit $t+p(o)$. The set of all executions is given by

$$E = \{o[k] \mid o \in O,\ k \in \mathbb{Z}\}.$$

So, each operation $o \in O$ is started exactly every $p(o)$ time units. Consequently, if for an operation o the start time of an arbitrary execution is fixed, then all executions of o are fixed in time. Without loss of generality, the executions of operation o are uniquely specified by a start time $s(o)$, with $0 \leq s(o) < p(o)$. Hence, a schedule $\mathcal{S}$ of the operations in O is uniquely determined by an n-tuple $(s(o_1), s(o_2), \ldots, s(o_n))$, with $0 \leq s(o_i) < p(o_i)$ for all $o_i \in O$. Furthermore, the operations are considered independent, i.e., there are no precedence constraints between executions of different operations.

Scheduling periodic operations naturally leads to periodic schedules. A schedule $\mathcal{S}$ is called periodic with period P if for each time unit $t \in \mathbb{Z}$ and each $o \in O$ the following holds:

operation o is executed at time unit t if and only if it is executed at time unit $t + P$.

Clearly, in order for a schedule to be periodic with period P, it is required that $p(o) \mid P$, for each $o \in O$. Consequently, the minimal period P of a schedule is given by $\text{lcm}(p(o_1), \ldots, p(o_n))$, i.e., the least common multiple of the periods of the individual operations.

Let M denote the set of processors. The processors are supposed to be identical, i.e., each operation $o \in O$ can be executed on any processor $m \in M$ and the time to execute operation o does not depend on the processor. Furthermore, a processor can only execute one operation at a time. We aim to minimize the number of processors necessary for the execution of the operations in O. Given a schedule $\mathcal{S}$, we can define the *thickness function* $T_{\mathcal{S}} : \mathbb{Z} \to \mathbb{N}$ which assigns to each $t \in \mathbb{Z}$ the number of operations that are being executed at that time unit. Since a processor can only execute one operation at a time, $\max_t T_{\mathcal{S}}(t)$ gives, for a given schedule $\mathcal{S}$, a lower bound on the number of processors that is required to carry out schedule $\mathcal{S}$. If schedule $\mathcal{S}$ is periodic with period P, then the thickness function $T_{\mathcal{S}}$ is also periodic with a period P', for which $P' \mid P$. Hence, to determine $\max_t T_{\mathcal{S}}(t)$, it suffices to consider time units $t \in \{1, \ldots, P\}$.

With respect to the assignment of executions to processors we consider two different cases, namely

(i) the *constrained* case, where all executions of an operation o have to be assigned to the same processor, for all $o \in O$, i.e., an assignment from O to M is required, and

(ii) the *unconstrained* case, where each execution $o[k]$ can be assigned to a different processor, i.e., an assignment from E to M is required.

An assignment of each execution in E to a processor in M may be difficult to specify, since E is a (countably) infinite set. We therefore restrict ourselves to periodic assignments. An assignment is called periodic with period $P \in \mathbb{N}$ if for each time unit $t \in \mathbb{Z}$, each $o \in O$, and for each $m \in M$ the following holds:

m executes o at time unit t if and only if m executes o at time unit $t + P$.

If for a periodic schedule $\mathcal{S}$ with period P the corresponding assignment is periodic with period P', then necessarily $P \mid P'$. In the constrained case, i.e., if all executions of an operation are assigned to the same processor, the assignment is necessarily periodic with period $P' = P$. For the unconstrained case, restricting oneself to periodic assignments does not lead to the use of extra processors as long as the length of period P' is not restricted. This is shown in the following theorem.

Theorem 1 *For each periodic schedule $\mathcal{S}$ a periodic assignment exists requiring only* $\max_t T_{\mathcal{S}}(t)$ *processors.*

Proof We have seen that $\max_t T_{\mathcal{S}}(t)$ gives a lower bound on the required number of processors. Now a finite set of executions can be optimally assigned to $\max_t T_{\mathcal{S}}(t)$ processors, using an $O(n \log n)$ algorithm [Hashimoto & Stevens, 1971; Gupta, Lee & Leung, 1979], where n denotes the number of executions. The algorithm assigns the executions in order of increasing start times to the first available processor, i.e., to the available processor with the smallest index number. Let us consider the assignment of a finite set of executions, namely the set of all executions in the time interval $[0, mP-1]$, with $m \in \mathbb{N}$ and $P = \text{lcm}(p(o_1), \ldots, p(o_n))$. We show that, if m is chosen sufficiently large, the assignment necessarily becomes periodic with some period $m'P$, $m' < m$. Let us examine the assignment in intervals $[lP, (l+1)P-1]$, with $0 \leq l < m$. The assignment can attain only a finite set of different solutions in such an interval $[lP, (l+1)P-1]$, since a finite set of executions can be assigned to a finite set of processors. Consequently, if m is chosen sufficiently large, then in two intervals $[lP, (l+1)P-1]$ and $[l'P, (l'+1)P-1]$, with $0 \leq l < l' < m$, the assignment must necessarily be identical. Hence, the assignment necessarily becomes periodic with period $(l'-l)P$, using only $\max_t T_{\mathcal{S}}(t)$ processors, which completes the proof of the theorem. ∎

The minimum period for which a periodic assignment uses $\max_t T_{\mathcal{S}}(t)$ processors may generally be very large. For reasons of simplicity, we restrict ourselves in this paper to periodic assignments with periods of minimal length, i.e., with a period $P = \text{lcm}(p(o_1), \ldots, p(o_n))$. In this way, an operation o is executed on at most $P/p(o)$ different processors. For the unconstrained case, an assignment is thus completely specified if the processor is given for $P/p(o)$ successive executions of each operation $o \in O$, denoted by $o[1], o[2], \ldots, o[P/p(o)]$, where $o[1]$ is defined to be the first execution starting at a time unit $t \geq 0$.

Given the definitions and assumptions described above, we can define the following periodic assignment problems. We formulate these problems as decision problems.

Unconstrained Periodic Assignment (UPA)
Given a schedule $\mathcal{S}$ for a set O of periodic operations with an execution time $e(o) \in \mathbb{N}$ and a period $p(o) \in \mathbb{N}$ for each $o \in O$, and an integer k, does an unconstrained periodic assignment with period $P = \text{lcm}(p(o_1), \ldots, p(o_n))$ exist that uses at most k processors?

Constrained Periodic Assignment (CPA)
Given a schedule $\mathcal{S}$ for a set O of periodic operations with an execution time $e(o) \in \mathbb{N}$ and a period $p(o) \in \mathbb{N}$ for each $o \in O$, and an integer k, does a constrained periodic assignment exist that uses at most k processors?

Likewise, we define the following periodic scheduling problems.

Unconstrained Periodic Scheduling (UPS)
Given a set O of periodic operations with an execution time $e(o) \in \mathbb{N}$ and a period $p(o) \in \mathbb{N}$ for each $o \in O$, and an integer k, does a schedule exist for which an unconstrained periodic assignment with period $P = \text{lcm}(p(o_1), \ldots, p(o_n))$ uses at most k *processors?*

Constrained Periodic Scheduling (CPS)
Given a set O of periodic operations with an execution time $e(o) \in \mathbb{N}$ and a period $p(o) \in \mathbb{N}$ for each $o \in O$, and an integer k, does a schedule exist for which a constrained periodic assignment uses at most k processors?

With respect to CPS the following theorem gives a necessary and sufficient condition for scheduling the executions of two operations on the same processor.

Theorem 2 *The executions of two periodic operations o_i and o_j can be scheduled on the same processor if and only if*

$$\gcd(p(o_i), p(o_j)) \geq e(o_i) + e(o_j). \tag{1}$$

Proof Let $g = \gcd(p(o_i), p(o_j))$. We first prove that (1) is a sufficient condition. This is shown as follows. Choosing the start times $s(o_i) = 0$ and $s(o_j) = e(o_i)$, operation o_i is executed in a subset of the set I_i of intervals, defined by $[lg, lg + e(o_i) - 1]$, $l \in \mathbb{Z}$, and operation o_j is executed in a subset of the set I_j of intervals, defined by $[lg + e(o_i), lg + e(o_i) + e(o_j) - 1]$, $l \in \mathbb{Z}$. Hence, if $g \geq e(o_i) + e(o_j)$, then no intervals of I_i and I_j overlap, which proves the sufficiency of (1).

We prove the necessity of (1) by showing that, if $g < e(o_i) + e(o_j)$, operation o_i and o_j cannot be scheduled on the same processor. So, assume that $g < e(o_i) + e(o_j)$. Without loss of generality we may assume that $s(o_i) = 0$. We now have to prove that integers x, y exist for which

$$[xp(o_i), xp(o_i) + e(o_i) - 1] \cap [s(o_j) + yp(o_j), s(o_j) + yp(o_j) + e(o_j) - 1] \neq \emptyset$$

or, equivalently,

$$[xp(o_i) - yp(o_j), xp(o_i) - yp(o_j) + e(o_i) - 1] \cap [s(o_j), s(o_j) + e(o_j) - 1] \neq \emptyset.$$

From elementary number theory it is known that integers w, z exist for which $wp(o_i) + zp(o_j) = g$. If we choose $x = lw$ and $y = -lz$, with $l \in \mathbb{Z}$, it suffices to show that for some integer l

$$[lg, lg + e(o_i) - 1] \cap [s(o_j), s(o_j) + e(o_j) - 1] \neq \emptyset.$$

Clearly, this must be the case since the free intervals between the intervals $[lg, lg + e(o_i) - 1]$, $l = 0, 1, \ldots$, are of length $g - e(o_i)$, while the intervals $[s(o_j), s(o_j) + e(o_j) - 1]$ are of length $e(o_j)$. Hence, the assumption that $g < e(o_i) + e(o_j)$ implies that some integer l necessarily exists for which $[lg, lg + e(o_i) - 1]$ and $[s(o_j), s(o_j) + e(o_j) - 1]$ overlap. This completes the proof of the theorem. ∎

A similar condition can be derived for CPA, as is shown in the following theorem.

Theorem 3 *For CPA, two periodic operations o_i and o_j with given start times $s(o_i)$ and $s(o_j)$, can be executed on the same processor if and only if*

$$e(o_i) \leq (s(o_j) - s(o_i)) \bmod g \leq g - e(o_j), \tag{2}$$

where $g = \gcd(p(o_i), p(o_j))$.

Proof Without loss of generality we may assume that $s(o_i) = 0$. This is true since, if $s(o_i) \neq 0$, then the start times of o_i and o_j can be shifted such that $s(o_i)$ becomes zero, without affecting possible overlap. The sufficiency of (2) is shown as follows. Let us consider time intervals $[0 + kg, g - 1 + kg]$, with $k \in \mathbb{Z}$. The first $e(o_i)$ time units of each of these intervals can be allocated for executions of o_i, and the remaining $g - e(o_i)$ time units for executions of o_j. Now, if (2) holds, then the allocated time units surely suffices to execute o_i and o_j. The first $e(o_i)$ time units of the intervals are only used to execute o_i once every $p(o_i)/g$ intervals. The remaining $g - e(o_i)$ time units are only (partly) used to execute o_j once every $p(o_j)/g$ intervals.

The necessity of (2) is shown as follows. Let us again consider the time intervals $[0+kg, g-1+kg]$, with $k \in \mathbb{Z}$. If (2) does not hold then the execution of o_j overlaps the first $e(o_i)$ time units once every $p(o_j)/g$ time intervals. We have already seen that the first $e(o_i)$ time units of the intervals are used for the execution of o_i once every $p(o_i)/g$ time units. Now, by definition, $\gcd(p(o_i)/g, p(o_j)/g) = 1$. Hence, if (2) does not hold, then operations o_i and o_j cannot be executed on the same processor. This completes the proof of the theorem. ∎

Note that Theorem 2 can be considered a corollary of Theorem 3, since (1) directly follows from (2). In the next section we examine the computational complexity of the problems defined above.

4 Computational Complexity

To examine the complexity of the periodic assignment problems CPA and UPA, we focus our attention on the subset of problem instances for which $p(o) = p$ for all $o \in O$. Note that under this restriction CPA and UPA are identical. If we prove that this subset of instances is NP-complete, then both CPA and UPA have been proved to be NP-complete.

Theorem 4 *CPA and UPA are NP-complete.*

Proof It is easily verified that CPA and UPA are in $\mathcal{NP}$. Now the NP-completeness is proved by a reduction from circular-arc colouring, which has been shown to be NP-complete by Garey, Johnson, Miller & Papadimitriou [1980]. We first define circular-arc colouring. Let a set of circular arcs $A = \{a_1, \ldots, a_n\}$ be given, where each arc a_i, specified by an ordered pair (l_i, r_i), with $l_i, r_i \in \{0, 1, \ldots, 2n-1\}$, is an arc on a circle with circumference $2n$ that stretches clockwise from point l_i to point r_i, containing both endpoints, and let an integer k be given. The problem is now: is A k-colourable, i.e., does a function $f : A \to \{1, \ldots, k\}$ exist such that $f(a_i) \neq f(a_j)$ whenever a_i and a_j overlap? Any instance of circular-arc colouring can be transformed to a periodic assignment instance as follows. For each arc a_i we define a periodic operation with period $p(o_i) = 2n$, start time $s(o_i) = l_i$, and execution time $e(o_i) = r_i - l_i + 1$ if $r_i \geq l_i$ and $e(o_i) = r_i - l_i + 2n + 1$ if $r_i < l_i$. Now two periodic operations can be assigned to the same processor if and only if the corresponding circular arcs can be coloured with the

same colour. Consequently, the circular arcs can be coloured using k colours if and only if the periodic operations can be assigned to k processors. Evidently, this is a polynomial-time transformation, which completes the proof of the theorem. ■

Note that the transformation from circular-arc colouring defines an equivalence between circular-arc colouring and the problem of assigning operations with identical periods, which we will use in Section 5.1.

To consider the complexity of CPS and UPS we again focus our attention on the subset of problem instances for which $p(o) = p$ for all $o \in O$. Again notice that this subset is in the intersection of the CPS and UPS problem instances.

Theorem 5 *CPS and UPS are NP-complete in the strong sense.*

Proof It is easily verified that CPS and UPS belong to $\mathcal{NP}$. We now prove the NP-completeness by a reduction from bin packing, which is NP-complete in the strong sense [Garey & Johnson, 1979]. An instance of bin packing is specified as follows. Let a finite set $A = \{a_1, \ldots, a_n\}$ of items be given, with for each item $a_i \in A$ a positive integer size $s(a_i)$, a positive bin capacity B and a positive integer k. Can A be partitioned into k disjoint subsets $A_1, \ldots, A_k$, such that the sum of the sizes in each subset A_i does not exceed the bin capacity B? Any instance of bin packing can be directly transformed into an instance of CPS or UPS as follows. For each item a_i we define a periodic operation o_i with execution time $e(o_i) = s(a_i)$ and period $p(o_i) = B$. Clearly, a number of periodic operations can be executed on the same processor if the corresponding items can be packed in one bin, and vice versa. Hence, the items $a_1, \ldots, a_n$ can be packed into k bins if and only if the operations $o_1, \ldots, o_n$ can be scheduled on k processors. Since the above transformation is polynomial, CPS and UPS are both NP-complete in the strong sense. ■

An alternative reduction from 3-partition can be constructed, showing that the problems remain NP-complete in the strong sense for the case that only one processor is available. Hence, this gives a stronger result. We have chosen, however, to give the reduction from bin packing since this reduction defines an equivalence between bin packing and the problem of scheduling periodic operations with identical periods, which we will use in Section 5.1.

5 Approximation Algorithms

All problems presented in Section 3 are NP-complete. This means that, unless $\mathcal{P} = \mathcal{NP}$, efficient optimization algorithms do not exist for these problems. We therefore focus our attention on approximation algorithms, i.e., algorithms which do not guarantee to find an optimal solution for every instance but attempt to find near-optimal solutions. In the remainder of this paper we present approximation algorithms for the periodic scheduling and assignment problems presented in Section 3 and, to some extent, analyse their performance. An interesting subclass of problems arises if we assume that the operations all have identical periods. We first consider approximation algorithms for this subclass of problems.

5.1 Periodic Operations with Identical Periods

In Section 4 we already indicated the equivalence between bin packing and the problem

of scheduling periodic operations with identical periods. Hence, approximation algorithms for bin packing can be directly applied to this problem. A large number of approximation algorithms exist for bin packing, ranging from simple approximation algorithms called *first fit* and *first fit decreasing*, which have asymptotic performance ratios of $\frac{17}{10}$ and $\frac{11}{9}$, respectively, to approximation schemes. An extensive survey of the literature on approximation algorithms for bin packing is given by Coffmann, Garey & Johnson [1984]. A bin packing algorithm gives a partitioning of the operations into subsets such that the operations in the same subset can be assigned to the same processor. A feasible schedule can then easily be obtained by scheduling the operations in each subset one after the other, in some arbitrary order. The wealth of approximation algorithms for bin packing provided by the literature surely suffices to effectively handle this subclass of periodic scheduling problems.

To present approximation algorithms for the assignment of periodic operations with identical periods we refer to its equivalence with the problem of colouring circular arcs, as indicated in Section 4. To the best of our knowledge, Tucker [1975] is the only author who considers the subject of approximation algorithms for colouring circular arcs, in order to give an upper bound on the number of colours necessary for colouring circular arcs. Elaborating on this result, we present the following 2-step approximation algorithm for colouring circular arcs, called *sort&match*.

1. Partition the set of arcs into two subsets A and B, where A contains all arcs that cover one specific point $t \in \{0, 1, \ldots, 2n-1\}$ for which the thickness function attains a minimum value, and B contains all remaining arcs. Consequently, $|A| = \min_t T_S(t)$. Now the arcs in B can be optimally assigned using the assignment algorithms of Hashimoto & Stevens [1971] or Gupta, Lee & Leung [1979] using $\max_t T_S(t)$ colours: the arcs a_i in B are sorted in order of their starting point l_i and they are assigned in this order to the first available colour, i.e., the available colour with the smallest index number.

2. Determine a maximum subset A' of arcs in A which can be coloured with a colour that is already used in step 1 to colour arcs in B. This problem can be formulated as a maximum-cardinality matching problem in a bipartite graph, which can be solved efficiently using an augmenting path algorithm [Edmonds, 1965; Hopcroft & Karp, 1973]. Finally, each remaining arc in $A - A'$ is given a different free colour.

Tucker [1975] only considers the first step of the algorithm presented above. Clearly the algorithm requires at most $\max_t T_S(t) + \min_t T_S(t)$ colours. Since $\max_t T_S(t)$ is a lower bound on the number of required colours, *sort&match* has a worst-case performance ratio of 2. This worst-case performance ratio already holds for the first step of the algorithm (assuming that all arcs in A are given a different free colour), which Tucker already showed. The worst-case performance bound can be shown to be tight [Korst, Aarts, Lenstra & Wessels, 1991]. The average-case performance of *sort&match* is much better. Experimental results indicate that the algorithm almost always finds solutions that are within 10% of the optimum for randomly generated instances [Korst, Aarts, Lenstra & Wessels, 1991].

5.2 Periodic Operations with Arbitrary Periods

In this subsection we discuss possible approximation algorithms for the UPA, CPA, UPS and CPS problems, for the case that operations have arbitrary integer periods.

Approximation Algorithm for UPA

Sort&match, presented in Section 5.1, can also be used as an approximation algorithm for UPA by associating an arc with each execution that is contained in a time window of length $P = \text{lcm}(p(o_1), \ldots, p(o_n))$. Note, however, that here the number of arcs is not polynomially bounded by the number of operations. The performance bound of *sort&match* clearly remains unaffected. Circular arcs can be efficiently coloured if they are proper, i.e., if no arc is completely contained in another arc [Orlin, Bonuccelli & Bovet, 1981;Shih & Hsu, 1989]. Hence, if periodic operations all have identical execution times, they can be optimally assigned to processors in a time that is polynomial in the number of executions.

Approximation Algorithms for CPA

Using Theorem 3 we can easily determine for each pair of periodic operations whether they can be assigned to the same processor. Consequently, we can define a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where each $v_i \in \mathcal{V}$ is associated with a periodic operation o_i. Two vertices v_i and v_j are adjacent if the associated operations o_i and o_j cannot be assigned to the same processor. The resulting graph $\mathcal{G}$ is called a *periodic-interval graph*. Now it is easy to see that solving a CPA instance is identical to colouring the vertices of the corresponding periodic-interval graph with a minimum number of colours. A periodic-interval graph can be considered to be a generalization of a circular-arc graph in the case that all periods are identical. To the best of our knowledge no graph colouring algorithms are presented in the literature that are tailored to colouring periodic-interval graphs. However, approximation algorithms for colouring arbitrary graphs might give satisfactory results in practice.

Approximation Algorithms for UPS

Experimental results indicate that *sort&match* is able to find solutions for UPA that are often close to $\max_t T_S(t)$. It therefore seems tempting to handle UPS using the following two-step approach:

1. first determine start times for the operations such that $\max_t T_S(t)$ is minimized, and
2. next use *sort&match* to find a feasible assignment.

Now the problem of finding a schedule such that $\max_t T_S(t)$ is minimized can be shown to be NP-complete. This immediately follows from the fact that UPS remains NP-complete for the single processor case. Consequently, we can restrict ourselves to constructing an approximation algorithm for the problem of finding start times that minimize $\max_t T_S(t)$. Note that for a set O' of periodic operations with $\gcd(p(o_i), p(o_j)) = 1$ for all $o_i, o_j \in O'$, we have $\max_t T_S(t) = |O'|$ for any possible choice of start times. This is a corollary of Theorem 2; see also [Park & Yun, 1985]. Consequently, the set of periodic operations O can be partitioned into a number of disjoint subsets $O_1, O_2, \ldots, O_l$ such that $\gcd(p(o_i), p(o_j)) = 1$ for each pair of operations o_i, o_j that have been assigned to different subsets, and $\max_t T_{S(O_i)}(t)$ can be minimized independently for each subset O_i. The total thickness $\max_t T_S(t)$ is then given by $\sum_{O_i} \max_t T_{S(O_i)}$. This partitioning approach will reduce the size of the problem.

We now restrict ourselves to minimizing $\max_t T_S(t)$ for a given subset O_i. This can be done as follows. First select a subset $O_{i'}$ of O_i, for which $\gcd(p(o_i), p(o_j)) = 1$ for all $o_i, o_j \in O_{i'}$, such that $O_{i'}$ is as large as possible. This is done by using some independent set heuristic. The operations in $O_{i'}$ are given arbitrary start times. Next, the remaining operations must be given start times subject to the start times of the operations in $O_{i'}$. If the number of operations in $O_i - O_{i'}$ is small, an enumeration is most appropriate. Otherwise, some constructive or

local search approach can be used.

Approximation Algorithm for CPS

In the case of CPS we observe the following. If one or more periodic operations are assigned to a processor, then the time that the processor remains idle can be expressed as one or more periodic intervals, each with a period and a duration. For example, if a periodic operation o_i with period $p(o_i)$ and execution time $e(o_i)$ is assigned to an idle processor, then the remaining idle time can be expressed as a periodic interval with period $p(o_i)$ and a duration $p(o_i) - e(o_i)$. We can thus consider the problem of assigning periodic operations to processors as the problem of assigning periodic operations to periodic intervals. For reasons of simplicity we denote a periodic operation o_i with period $p(o_i)$ and execution time $e(o_i)$ by the ordered pair (p_i, e_i) and a periodic interval with period p_j and duration d_j by the ordered pair $[\![p_j, d_j]\!]$. From Theorem 2 we derive that a periodic operation (p_i, e_i) can be assigned to a periodic interval $[\![p_j, d_j]\!]$ if and only if $\gcd(p_i, p_j) \geq e_i + (p_j - d_j)$. Let $g = \gcd(p_i, p_j)$ and $e_j = p_j - d_j$; then by assigning periodic operation (p_i, e_i) to periodic interval $[\![p_j, d_j]\!]$, the remaining idle time can be expressed as a set of periodic intervals in a number of alternative ways. We assume that a periodic operation is always started at the begin of the periodic interval to which it is assigned. Consequently, the remaining idle time can be expressed as one of the following three alternatives.

1. $p_i/g - 1$ periodic intervals $[\![p_i, g - e_j]\!]$,
   $p_j/g - 1$ periodic intervals $[\![p_j, e_j]\!]$, and
   1 periodic interval $[\![p_i, g - e_i - e_j]\!]$
2. $p_i/g - 1$ periodic intervals $[\![p_i, e_i]\!]$,
   $p_j/g - 1$ periodic intervals $[\![p_j, e_j]\!]$, and
   1 periodic interval $[\![g, g - e_i - e_j]\!]$
3. $p_i/g - 1$ periodic intervals $[\![p_i, e_i]\!]$,
   $p_j/g - 1$ periodic intervals $[\![p_j, g - e_i]\!]$, and
   1 periodic interval $[\![p_j, g - e_i - e_j]\!]$

In all three cases the number of periodic intervals is given by

$$\frac{p_i + p_j}{\gcd(p_i, p_j)} - 1.$$

Note that, if $p_i = p_j$, the three alternatives are identical, leading to only one periodic interval. Otherwise, if $p_i | p_j$ or $p_j | p_i$, then the three alternatives reduce to two essentially different ones.

Based on this observation, we propose the following iterative approximation algorithm. In each iteration all possible assignments of periodic operations to periodic intervals are considered and the one that is considered best is selected to be scheduled. The 'goodness' of an operation-to-interval assignment is defined by the amount of idle time that remains after assigning the periodic operation to the periodic interval. In each iteration the assignment of (p_i, e_i) to $[\![p_j, d_j]\!]$ is selected for which $d_j/p_j - e_i/p_i$ is minimal, provided that the assignment is feasible. Clearly, the amount of idle time that remains after assigning an operation (p_i, e_i) to an idle processor is given by $1 - e_i/p_i$. Consequently, the algorithm will not assign a periodic operation to an idle processor as long as the periodic operation can be assigned to a periodic interval of a processor that is already in use. After each iteration, the remaining idle time is expressed as one or more periodic intervals using one of the three alternatives

mentioned above. Which alternative is selected is determined by considering how well the unassigned operations fit in the periodic intervals. This can be considered as a maximum-weight matching problem on a bipartite graph, which can be handled efficiently.

A detailed analysis of the algorithm is beyond the scope of the paper. We mention that, in the case of periodic operations with identical periods, solutions are found that are identical to the ones obtained by *first fit decreasing* for bin packing.

6 Conclusions

A number of closely interrelated optimization problems have been discussed from the field of nonpreemptive periodic scheduling. The complexity of these problems has been examined. We have derived Necessary and sufficient conditions for executing two periodic operations on a single processor. Finally, approximation algorithms have been proposed for periodic scheduling and periodic assignment problems, for the constrained case as well as the unconstrained case.
The material presented in this paper leaves the following open problems:

- Which constraints do we have to impose on the problems discussed in this paper to allow for efficient optimization algorithms?
- Do approximation algorithms exist for colouring periodic-interval graphs that have a constant worst-case performance ratio?
- Do approximation algorithms exist for colouring circular-arc graphs with a worst-case performance ratio smaller than two?
- Is it possible to give a constant worst-case performance ratio for the approximation algorithms for CPS and UPS?

Bibliography

Bartholdi, J.J. [1981], A guaranteed-accuracy round-off algorithm for cyclic scheduling and set covering, *Operations Research* **29**, 501-510.

Bartholdi, J.J., J.B. Orlin, and H.D. Ratliff [1980], Cyclic scheduling via integer programs with circular ones, *Operations Research* **28**, 1074-1085.

Bertossi, A.A. and M.A. Bonuccelli [1983], Preemptive scheduling of periodic jobs in uniform multiprocessor systems, *Information Processing Letters* **16**, 3-6.

Bertossi, A.A. and M.A. Bonuccelli [1985], A polynomial feasibility test for preemptive periodic scheduling of unrelated processors, *Discrete Applied Mathematics* **12**, 195-201.

Burkard, R.E. [1986], Optimal schedules for periodically recurring events, *Discrete Applied Mathematics* **15**, 167-180.

Chetto, H. and M. Chetto [1989], Scheduling periodic and sporadic tasks in a real-time system, *Information Processing Letters* **30**, 177-184.

Coffmann, E.G., Jr., M.R. Garey, and D.S. Johnson [1984], Approximation algorithms for bin packing - an updated survey, in: G. Ausiello, M. Lucertini, and P. Serafini (Eds.), *Algorithms Design and Computer System Design*, CISM Courses and Lectures 284, Springer, Vienna, 49-106.

Dhall, S.K. and C.L. Liu [1978], On a real-time scheduling problem, *Operations Research* **26**, 127-140.

Edmonds, J. [1965], Paths, trees and flowers, *Canadian Journal of Mathematics* **17**, 449-467.

Garey, M.R. and D.S. Johnson [1979], *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., San Francisco.

Garey, M.R., D.S. Johnson, G.L. Miller, and C.H. Papadimitriou [1980], The complexity of coloring circular arcs and chords, *SIAM Journal on Algebraic and Discrete Methods* **1**, 216-227.

Gonzalez, M.J. and J.W. Soh [1975], Periodic job scheduling in a distributed processor system, *IEEE Transactions on Aerospace and Electronic Systems* **12**, 530-536.

Gupta, U.I., D.T. Lee, and J.Y.-T. Leung [1979], An optimal solution for the channel-assignment problem, *IEEE Transaction on Computers* **28**, 807-810.

Hashimoto, A. and J. Stevens [1971], Wire routing by optimizing channel assignment with large apertures, *Proceedings of the 8th Design Automation Conference*, 155-169.

Hopcroft, J.E. and R.M. Karp [1973], An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, *SIAM Journal on Computing* **2**, 225-231.

Korst, J.H.M., E.H.L. Aarts, J.K. Lenstra, and J. Wessels [1991], Periodic Assignment and Graph Colouring, *Philips Research Manuscript.*

Labetoulle, J. [1974], Some theorems on real time scheduling, in: E. Gelenbe and R. Mahl (Eds.), *Computer Architecture and Networks*, North-Holland, Amsterdam, 285-293.

Lawler, E.L. and C.U. Martel [1981], Scheduling periodically occurring tasks on multiple processors, *Information Processing Letters* **12**, 9-12.

Leung, J.Y.-T. and M.L. Merrill [1980], A note on preemptive scheduling of periodic, real-time tasks, *Information Processing Letters* **11**, 115-118.

Leung, J.Y.-T. and J. Whitehead [1982], On the complexity of fixed-priority scheduling of periodic, real-time tasks, *Performance Evaluation* **2**, 237-250.

Liu, C.L. and J.W. Layland [1973], Scheduling algorithms for multiprogramming in a hard real-time environment, *Journal of the Association for Computing Machinery* **20**, 46-61.

Orlin, J.B. [1982], Minimizing the number of vehicles to meet a fixed periodic schedule: an application of periodic posets, *Operations Research* **30**, 760-776.

Orlin, J.B., M.A. Bonuccelli, and D.P. Bovet [1981], An $O(n^2)$ algorithm for coloring proper circular arc graphs, *SIAM Journal on Algebraic and Discrete Methods* **2**, 88-93.

Park, K.S. and D.K. Yun [1985], Optimal scheduling of periodic activities, *Operations Research* **33**, 690-695.

Serafini, P. and W. Ukovich [1989], A mathematical model for periodic scheduling problems, *SIAM Journal on Discrete Mathematics* **2**, 550-581.

Shih, W.-K. and W.-L. Hsu [1989], An $O(n^{1.5})$ algorithm to color proper circular arcs, *Discrete Applied Mathematics* **25**, 321-323.

Tucker, A [1975], Coloring a family of circular arcs, *SIAM Journal on Applied Mathematics* **29**, 493-552.

Vince, J. [1989], Scheduling periodic events, *Discrete Applied Mathematics* **25**, 299-310.

Embeddings of shuffle-like graphs in hypercubes *

M. Baumslag§, M.C. Heydemann†, J. Opatrny‡, D. Sotteau†
§Comp. and Inf. Science Dept, Univ. of Massachusetts, USA
†LRI, UA 410 CNRS, bât 490, Univ. Paris-Sud, 91405 Orsay France
‡Dept of Computer Sciences, Concordia Univ., Montréal, Canada

Abstract

Let G and H be two simple undirected graphs. An *embedding* of the graph G in the graph H is an injective mapping f from the vertices of G into the vertices of H together with a mapping P_f of edges of G into paths in H. The *dilation* of the embedding is the maximum taken over all the lengths of the paths $P_f(x,y)$ associated with the edges (x,y) of G.
One challenge pointed out in [9] is to find embeddings of the de Bruijn graph in the hypercube of the same order which have a low dilation. For a de Bruijn graph of diameter D we give an embedding in a hypercube of the same diameter of dilation $2\lceil D/5 \rceil$, and determine the edge-congestion and vertex-congestion of this embedding. Similar results are given for the shuffle-exchange graphs.

1 Introduction

A parallel algorithm can be represented by a graph, say G, in which the nodes represent the processes, and the edges represent the communications among the processes. Similarly, a parallel computer can be represented by a graph, say H, in which the nodes represent the processors and the edges represent the communication links among the processors. An important problem in parallel computations is how to map G into H such that H can carry out efficiently all communications specified by G. This problem has two components. First, how to map the processes of G into the processors of H, and second, how to assign a physical communication path in H to each edge of G so that the parallel algorithm can be efficiently executed. This problem is known in graph theoric terms as graph embedding problem. We can define it more precisely as follows.

Let G and H be two simple undirected graphs. An *embedding* of the graph G in the graph H is an injective mapping f from the vertices of G into the vertices of H together with a mapping P_f of edges of G into paths in H. For any edge (x,y) of G, $P_f(x,y)$ denotes the path between the vertices $f(x)$ and $f(y)$ in H assigned by P_f.

*The work was supported partially by NSERC of Canada and by PRC C3 of France and was partially done while the third author was visiting the University of Paris-Sud.

From among the parameters that have been used in measuring the efficiency of embeddings (see for example [10]), we will restrict our attention to the following ones.

The *dilation* of a given embedding f, denoted by $dil(f)$, is the maximum of the lengths of the paths $P_f(x,y)$ in H associated with all edges (x,y) of G. The minimum dilation of an embedding of G in H, denoted $dil(G,H)$, is the minimum of $dil(f)$ taken over all the embeddings f of G in H. Clearly the minimum will be reached in particular if we take for $P_f(x,y)$ shortest paths between x and y in H. In this paper we will always do so.

The *expansion* of the embedding f is the ratio of the number of vertices in H to the number of vertices in G. Here we only consider embeddings with expansion equal to 1.

The *edge-congestion* of f, denoted by $econg(f)$ is the maximum, over all edges e of H, of the number of edges of G mapped to a path of H which includes e.

The *vertex-congestion* of f, denoted by $vcong(f)$ is the maximum, over all vertices v of H, of the number of edges of G mapped to a path of H containing v as internal vertex.

Determining the computational power of hypercubes is a central problem in the theory and practice of parallel networks. In particular, we would like to know what communication patterns the hypercube can simulate efficiently. The following embeddings in hypercubes are known (for other results see the survey [12]):

- Any binary tree can be embedded in the smallest hypercube big enough to contain it with constant dilation and edge-congestion [2].
- Any d-dimensional mesh can be embedded in the smallest hypercube big enough to contain it with dilation $O(d)$ [4].
- Any butterfly-like graph (*i.e.,* the cube-connected cycles, the butterfly or the FFT graphs) can be embedded in the smallest hypercube big enough to contain it with dilation 2 and unit edge-congestion [9].

Also, Winkler [14] has characterized the graphs that can be embedded as an *isometric subgraph* of the hypercube, and Greenberg and Bhatt [8] have recently studied the problem of embedding *multiple copies* of the above classes of graphs into hypercubes.

One challenge pointed out by D. Greenberg, L. Heath and A. Rosenberg [9] is to find low dilation embeddings with expansion equal to 1 of shuffle-like networks, such as the de Bruijn graphs and the shuffle-exchange graphs, in hypercubes. De Bruijn networks have been studied as a possible choice for designing large communication networks , and very efficient, general sorting algorithms have been developed for them [1]. A de Bruijn network is being constructed for the NASA's Galileo space mission to be used as a signal decoder [5].

A different approach to the problem of a simulation of one network by another is through the notion of **work-preserving emulations** (see, for example, [11]). A guest network G is said to have a work-preserving emulation on a host network H if any T computation steps of G can be emulated in $O(T|G|/|H|)$ steps on H. In this model, computations of G can be *replicated* at several nodes of H (this may be visualized as a one-to-many embedding). Schwabe [13] recently proved that (in this model) any T steps of an N-node de Bruijn network can be emulated in $O(T)$ steps on an N-node hypercube,

provided that $T \geq \log N$. However, this result does not give an embedding of a de Bruijn network in a hypercube with expansion 1.

The D-dimensional hypercube, denoted by $H(D)$, has for vertex set the set of all binary words of length D. There is an edge between any two words that differ in exactly one position. The distance between two vertices x and y of $H(D)$ is denoted by $d_D(x,y)$.

A de Bruijn digraph $B(d,D)$ of order d^D, out and indegree d has been defined in [6] as follows. Its vertex set is the set of all words of length D on an alphabet A of size d. There is an arc from any vertex $x_1x_2\cdots x_D$ to the d vertices $x_2x_3\cdots x_D\lambda$, where λ is any letter of A. The undirected de Bruijn graph $UB(d,D)$ is obtained from the de Bruijn digraph by taking the underlying graph and deleting self loops and multiple edges. In this paper we will be concerned with the undirected de Bruijn graph on the alphabet $A = \{0,1\}$. We will denote $UB(2,D)$ by $B(D)$ for short.

The binary shuffle-exchange graph of diameter D, denoted by $S(D)$, is the graph whose vertices are all binary words of length D and whose edges are of two types. A *shuffle edge* connects any vertex $x_1x_2\cdots x_D$ to the vertex $x_2x_3\cdots x_Dx_1$. An *exchange edge* connects any vertex $x_1x_2\cdots x_D$ to the vertex $x_1x_2\cdots\overline{x_D}$ where $\overline{x_D} = 1 - x_D$.

The cartesian product of two graphs G and G', denoted by $G\Box G'$, is the graph whose vertices are all the pairs (u,v) where u is a vertex of G and v is a vertex of G'. We will denote here by uv the pair (u,v). The notation (u,v) will be reserved for the edge between the vertices u and v if it exists. Two vertices uv and $u'v'$ are connected in $G\Box G'$ if and only if $u = u'$ and (v,v') is an edge of G' or $v = v'$ and (u,u') is an edge of G.

Some heuristics have been given in [3] for embeddings of de Bruijn graphs in hypercubes. As far as the present authors can ascertain, there were no other known non-trivial embeddings of the deBruijn network into the hypercube until now.

In this paper we give constructions of embeddings of de Bruijn and shuffle-exchange graphs in hypercubes and deduce upper bounds for the parameters defined above. Notice that the dilation of an embedding of these shuffle-like graphs which contain odd cycles is at least 2 since the hypercube is bipartite (this is the case for $B(D)$, $D \geq 2$, and $S(D)$, $D \geq 3$). But we don't know any better lower bound than 2 for the dilation or 1 for the congestions of these embeddings.

In section 2 we obtain embeddings of the de Bruijn graph $B(D)$ in the hypercube $H(D)$ with dilation less than or equal to $2\lceil D/5\rceil$ and edge-congestion 2. From these embeddings we deduce in section 3 a bound for the dilation of embeddings of shuffle-exchange graphs.

In the appendix we give embeddings of shuffle-like graphs for graphs of diameters 2 to 6.

2 Upper bounds for the parameters of embeddings of de Bruijn graphs

Proposition 2.1 *Let f and f' be embeddings of $B(D)$ and $B(D')$ respectively in $H(D)$ and $H(D')$. Then there exists an embedding g of $B(D+D')$ in $H(D+D')$ with*

$$dil(g) \leq dil(f) + dil(f')$$

$$econg(g) \leq max\ (dil(f),\ dil(f'),\ 2)$$

$$vcon(g) \leq vcong(f) + vcong(f') + 2$$

Thus, for any strictly positive D and D', we have

$$dil(B(D+D'), H(D+D')) \leq dil(B(D), H(D)) + dil(B(D'), H(D')),$$

$$econg(B(D+D'), H(D+D')) \leq max\ (econg(B(D), H(D)), econg(B(D'), H(D')), 2)$$

$$vcong(B(D+D'), H(D+D')) \leq vcong(B(D), H(D)) + vcong(B(D'), H(D')) + 2.$$

Proof: The hypercube $H(D+D')$ is isomorphic to the cartesian product of the hypercubes $H(D)$ and $H(D')$. Any vertex $u = x_1x_2\cdots x_{D+D'}$ of $H(D+D')$ can be written as u_1u_2 where $u_1 = x_1x_2\cdots x_D$ and $u_2 = x_{D+1}\cdots x_{D+D'}$. When u_1, u_2 is fixed, the vertices u_1u, uu_2 of $H(D+D')$ span an induced subgraph isomorphic to $H(D')$, $H(D)$ respectively. If $s = s_1s_2$ and $t = t_1t_2$ are two vertices of $H(D+D')$ then

$$d_{D+D'}(s,t) = d_D(s_1,t_1) + d_{D'}(s_2,t_2). \tag{1}$$

Let f and f' be embeddings of $B(D)$ and $B(D')$ in $H(D)$ and $H(D')$, respectively. We define an embedding g of $B(D+D')$ in $H(D+D')$ as follows. For any vertex $x = x_1x_2\cdots x_{D+D'}$,

$$g(x_1x_2\cdots x_{D+D'}) = f(x_1x_2\cdots x_D)f'(x_{D+1}x_{D+2}\cdots x_{D+D'})$$

The paths of $H(D+D')$ associated with the edges of $B(D+D')$ will be specified later. This embedding could be considered as the composition of two embeddings. First, an embedding of $B(D+D')$ in the cartesian product of $B(D)$ and $B(D')$, where any vertex $u = x_1x_2\cdots x_{D+D'}$ of $B(D+D')$ is mapped on the vertex $x_1x_2\cdots x_Dx_{D+1}\cdots x_{D+D'}$ of $B(D)\Box B(D')$ where $u_1 = x_1x_2\cdots x_D$ is a vertex of $B(D)$ and $u_2 = x_{D+1}\cdots x_{D+D'}$ is a vertex of $B(D')$. Second, an embedding of $B(D)\Box B(D')$ in $H(D)\Box H(D')$ defined by the embeddings f of $B(D)$ in $H(D)$ and f' of $B(D)$ in $H(D')$. Embeddings of cartesian products have been studied in [10], but here we will not take this approach since it is much simpler to study g directly.
Consider an edge (u,v) of $B(D+D')$ with $u = u_1u_2 = x_1x_2\cdots x_Dx_{D+1}\cdots x_{D+D'}$ and $v = v_1v_2 = x_2x_3\cdots x_{D+1}x_{D+2}\cdots x_{D+D'}y$. Note that (u,v) is an edge of $B(D+D')$ if and only if (u_1,v_1) is an edge of $B(D)$ and (u_2,v_2) is an edge of $B(D')$.

By the definition of the embedding g, $g(u) = f(u_1)f'(u_2)$ and $g(v) = f(v_1)f'(v_2)$ in $H(D+D')$. Therefore, by (1) we get

$$d_{D+D'}(g(u),g(v)) = d_D(f(u_1),f(v_1)) + d_{D'}(f'(u_2),f'(v_2)).$$

Thus, for $D \geq D' \geq 1$,

$$dil(g) \leq dil(f) + dil(f').$$

We will now study the congestion of g.
To complete the definition of the embedding g, let us define the path $P_g(u,v)$ in $H(D+D')$ between $g(u)$ and $g(v)$, for every edge (u,v) of $B(D+D')$. Without loss of generality, we can assume that $u = u_1u_2$ with $u_1 = \gamma u'_1$, $u_2 = \alpha u'_2$, and $v = v_1v_2$ with $v_1 = u'_1\alpha$, $v_2 = u'_2\beta$ where $\alpha, \beta, \gamma \in \{0,1\}$.
Let $w = u_1v_2$. The path $P_g(u,v)$ is obtained as the concatenation of two paths.
The first one is a path from $g(u) = f(u_1)f'(u_2)$ to $g(w) = f(u_1)f'(v_2)$ in the subhypercube isomorphic to $H(D')$ obtained by making the first D coordinates equal to $f(u_1)$, a path isomorphic to $P_{f'}(u_2,v_2)$.
The second one is a path from $g(w) = f(u_1)f'(v_2)$ to $g(v) = f(v_1)f'(v_2)$ in the subhypercube isomorphic to $H(D)$ obtained by making the last D' coordinates equal to $f'(v_2)$, a path isomorphic to $P_f(u_1,v_1)$.

Remark: If we know two vertices from $\{g(u), g(v), g(w)\}$, we can determine the third one uniquely except if $D = 1$ or $D' = 1$. Also, given any vertex $g(w)$ (or equivalently w since g is bijective) we can determine $g(u)$ and $g(v)$ up to the parameter α (where $u_2 = \alpha u'_2$ and $v_1 = u'_1\alpha$). Thus, there are only two paths $P_g(u,v)$ going through $g(w)$.

Let us consider the load induced on any edge of $H(D+D')$ by all the paths $P_g(u,v)$. The edges of type (a_1a_2, a_1b_2), in a subhypercube isomorphic to $H(D')$, are only loaded by paths $P_{f'}(u_2,v_2)$ for all pairs of adjacent vertices u_2 and v_2 of $B(D')$, except if $D' = 1$. In this case the load of the unique edge (a_10, a_11) can be two. Similarly the edges of type (a_1a_2, b_1a_2), in a subhypercube isomorphic to $H(D)$, are only loaded by paths $P_f(u_1,v_1)$ for all pairs of adjacent vertices u_1 and v_1 of $B(D)$ except if $D = 1$. Indeed, from the above remark, a given path $P_{f'}(u_2,v_2)$ is connected to a unique path $P_f(u_1,v_1)$ (and conversely) in a unique path $P_g(u,v)$ except if D or D' equals 1. Therefore,

$$econg(g) \leq max(econg(f), econg(f'), 2).$$

Let us now consider any vertex of $H(D+D')$. Since f and f' are bijective, we can write such a vertex as $f(a_1)f'(a_2)$ (where a_1, a_2, is a vertex of $B(D)$, $B(D')$ respectively). Vertex $f(a_1)f'(a_2)$ is loaded by paths $P_{f'}(u_2,v_2)$ for all pairs of adjacent vertices u_2 and v_2 of $B(D')$ in the subhypercube isomorphic to $H(D')$ having the first D coordinates equal to $f(a_1)$. It is also loaded by paths $P_f(u_1,v_1)$ for all pairs of adjacent vertices u_1 and v_1 of $B(D)$ in the subhypercube isomorphic to $H(D)$ having the last D' coordinates equal to $f'(a_2)$. Finally, $f(a_1)f'(a_2)$ is also loaded as the connecting vertex $g(w)$ of two paths $P_g(u,v)$ for $w = a_1a_2$, by the above remark. Therefore

$$vcong(g) \leq vcong(f) + vcong(f') + 2. \quad \square$$

Lemma 2.2 *For $1 \leq D \leq 5$ there exist embeddings of the de Bruijn graph $B(D)$ in the hypercube $H(D)$ with the following parameters.*

D	1	2	3	4	5
dilation	1	2	2	2	2
edge congestion	1	2	2	2	2
vertex congestion	0	1	1	2	2

Proof: Embeddings giving these values are described in the appendix.

Notice that the dilation of embeddings of graphs in the above lemma matches the lower bound, but we don't know, for $D \geq 4$, if the congestions are optimal.

Theorem 2.3 *For $D \geq 2$, there exists an embedding of the de Bruijn graph $B(D)$ in the hypercube $H(D)$ with parameters bounded as follows.*

D	$\equiv 1 \pmod 5$	$\equiv 2$ *or* $3 \pmod 5$	$\equiv 4$ *or* $0 \pmod 5$
dilation	$\leq 2\lceil D/5 \rceil - 1$	$\leq 2\lceil D/5 \rceil$	
edge congestion	$= 2$		
vertex congestion	$\leq 4\lceil D/5 \rceil - 4$	$\leq 4\lceil D/5 \rceil - 3$	$\leq 4\lceil D/5 \rceil - 2$

Proof: For $D \leq 5$ we use the result of lemma 2.2. For $D \geq 6$ we use inductively the result of proposition 2.1. If $D = 5(k-1)+r$, $1 \leq r \leq 5$, we embed $B(D)$ in the cartesian product of $(k-1)$ copies of $B(5)$ and a copy of $B(r)$. The hypercube $H(D)$ is the cartesian product of $(k-1)$ copies of $H(5)$ and a copy of $H(r)$. We embed each $B(5)$ in a distinct copy of $H(5)$ and $B(r)$ in $H(r)$. From proposition 2.1 we get immediately

$$\begin{aligned} dil(B(D),H(D)) &\leq (k-1)dil(B(5),H(5)) + dil(B(r),H(r)) \\ &\leq 2(k-1)+2 \quad \text{if } r \neq 1 \\ &\leq 2(k-1)+1 \quad \text{if } r = 1 \end{aligned}$$

$$\begin{aligned} econg(B(D),H(D)) &\leq max_{\,1\leq r\leq 5}\ econg(B(r),H(r)) \\ &\leq 2 \end{aligned}$$

$$\begin{aligned} vcong(B(D),H(D)) &\leq (k-1)vcong(B(5),H(5)) + vcong(B(r),H(r)) + 2(k-1) \\ &\leq 4(k-1) + vcong(B(r),H(r)) \\ &\leq 4\left\lceil \tfrac{D}{5} \right\rceil - 4 \quad \text{if } r = 1 \\ &\leq 4\left\lceil \tfrac{D}{5} \right\rceil - 3 \quad \text{if } r = 2,3 \\ &\leq 4\left\lceil \tfrac{D}{5} \right\rceil - 2 \quad \text{if } r = 4,5. \qquad \square \end{aligned}$$

3 Dilation of embeddings of shuffle-exchange graphs

The following result was proved recently by Feldmann and Unger [7].

Theorem 3.1 *For any D the shuffle-exchange graph $S(D)$ is isomorphic to a subgraph of the binary de Bruijn graph $B(D)$.*

So we immediately get

Corollary 3.2

$$dil(S(D), H(D)) \leq dil(B(D), H(D))$$

Therefore, using theorem 2.3 we have the following.

Corollary 3.3 *For $D \geq 2$, $dil(S(D), H(D)) \leq 2\lceil D/5 \rceil$. If, furthermore, $D \equiv 1 \pmod 5$, then $dil(S(D), H(D)) \leq 2\lceil D/5 \rceil - 1$.*

Obviously the bounds of theorem 2.3 on the edge and vertex congestion are also valid for the embedding of the shuffle-exchange graph.

However for small values of D we get better values for the parameters as showed in the following array.

Lemma 3.4 *For small values of D there exist embeddings of the shuffle-exchange graph $S(D)$ in the hypercube $H(D)$ with the following parameters.*

D	1	2	3	4	5	6
dilation	1	1	2	2	2	2
edge congestion	1	1	1	1	1	2
vertex congestion	0	0	1	1	1	2

Proof: The embeddings are given in the appendix.
Notice that for these small values of D, the parameters of the embeddings are optimal except possibly the edge-congestion for $D = 6$.

Remark 3.5 : Using the results of the previous section we can give a direct proof of corollary 3.3. Indeed, let $D \geq 7$, $D = 5(k-1)+r$, $k \geq 2$, $1 \leq r \leq 5$. Consider the embedding g of $B(D)$ in $H(D)$ described in the proof of theorem 2.3,

$$g(x_1x_2\cdots x_D) = f_5(x_1x_2\cdots x_5)f_5(x_6x_7\cdots x_{10})\cdots \\ f_5(x_{5(k-2)+1}x_{5(k-2)+2}\cdots x_{5(k-1)})f_r(x_{5(k-1)+1}\cdots x_D)$$

where, for $1 \leq i \leq 5$, f_i is an embedding of $B(i)$ in $H(i)$ from lemma 2.2.

Let g' be the embedding of $S(D)$ in $H(D)$ such that $g' = g$. We will prove that $dil(g') \leq max\ (4, dil(g))$, which will give us an upper bound on $dil(S(D), H(D))$ identical to the upper bound on $dil(g)$ from theorem 2.3 for $D \geq 7$.

Let us consider the two types of edges (u, v) of S(D). If (u, v) is a shuffle edge of $S(D)$ then (u, v) is also an edge of $B(D)$, and by the proof of theorem 2.3, we get

$$d_D(g'(u), g'(v)) \leq d_D(g(u), g(v)) \leq dil(g).$$

If (u, v) is an exchange edge, that is $u = x_1x_2 \cdots x_D$ and $v = x_1x_2 \cdots x_{D-1}\overline{x_D}$ with $\overline{x_D} = 1 - x_D$, then u and v are mapped into the same subhypercube $H(r)$ of $H(D)$, and we have

$$d_D(g'(u), g'(v)) \leq d_r(f_r(x_{5(k-1)+1} \cdots x_D), f_r(x_{5(k-1)+1} \cdots \overline{x_D})).$$

For $1 \leq r \leq 4$, since we have $d_r(u', v') \leq 4$ for any vertices u', v' of $H(r)$, we obtain

$$d_D(g'(u), g'(v)) \leq 4.$$

For $r = 5$, by calculating the distances $d_5(f_5(u'), f_5(v'))$ for any exchange edge (u', v') of $S(5)$, we get

$$d_5(f_5(u'), f_5(v')) \leq 4.$$

Therefore, $d_D(g'(u), g'(v)) \leq \max(4, dil(g))$. □

Remark 3.6 We believe that a more direct proof could give better bounds for the parameters of the embedding of a shuffle-exchange graph in the hypercube than for those of the embedding of a de Bruijn graph.

4 Conclusion

The dilation of embeddings of binary shuffle-like graphs in hypercubes presented in this paper is better for $D \geq 5$ than those obtained by heuristics in [3]. However, the dilation of our embeddings is increasing linearly with the diameter. It is still an open problem whether the shuffle-like graphs can be embedded in the hypercubes with expansion 1, dilation $O(1)$ and edge-congestion $O(1)$.

As mentioned in [12], a de Bruijn graph can be embedded with dilation 2 into the shuffle-exchange graph of the same diameter, and from theorem 3.1 the shuffle-exchange graph is a subgraph of the de Bruijn graph. This gives

$$dil(S(D), H(D)) \leq dil(B(D), H(D)), \text{ and } dil(B(D), H(D)) \leq 2\, dil(S(D), H(D)).$$

Therefore it would be sufficient to find embeddings of one of these two types of graphs with dilation $O(1)$ in hypercubes, in order to find good embeddings for the other type.

Let us notice that if we could find an embedding of $S(D)$ in $H(D)$ with dilation $dil(S(D), H(D))$ where the exchange edges are embedded on edges of $H(D)$, then we would have

$$dil(B(D), H(D)) \leq dil(S(D), H(D)) + 1.$$

On the other hand we think that it might be possible to construct an embedding f'_{D+1} of $S(D+1)$ in $H(D+1)$ by splitting each vertex $x_1x_2\cdots x_D$ of $B(D)$ in two vertices $x_1x_2\cdots x_D0$ and $x_1x_2\cdots x_D1$. If f_D is an embedding of $B(D)$ in $H(D)$ of minimum dilation, the set of any two vertices $x_1x_2\cdots x_D0$ and $x_1x_2\cdots x_D1$ is embedded by f'_{D+1} on the set of the two vertices $f_D(x_1x_2\cdots x_D)0$ and $f_D(x_1x_2\cdots x_D)1$. If that is the case (as obtained for $D \leq 5$ in the appendix 5.2), we would get

$$dil(S(D+1), H(D+1)) \leq dil(B(D), H(D)).$$

Clearly, whenever we find an embedding of a de Bruijn graph G of fixed diameter $D \geq 6$ in the hypercube $H(D)$, whose dilation is smaller than the one from theorem 2.3, we can use G in the proof of the theorem, and improve the results of the theorem 2.3 and corollary 3.3 appropriately. However, the bound on the dilation would remain linearly proportional to the diameter of the graph. Thus, any attempt to show that the dilation of the embeddings of shuffle-like graphs is $O(1)$ must use a method different from ours.

5 Appendix: embeddings for small values of the diameter

We give here embeddings for $B(D)$ and $S(D)$ for $D \leq 6$.
Any vertex of each of the graphs considered is specified by the integer whose binary representation is the corresponding word of $B(D)$, $S(D)$, or $H(D)$. In the right column we give a vertex u of $B(D)$, $S(D)$, respectively, and in the left column we give the image of u in $H(D)$.

5.1 Embeddings of B(D)

For $D = 1$, $B(1)$ and $H(1)$ are isomorphic and we take the identity function as the embedding.

We give in the following table embeddings f_D of $B(D)$ in $H(D)$.

H(2)	B(2)
0	0
1	1
2	2
3	3

H(3)	B(3)
0	0
1	1
2	4
3	2
4	7
5	3
6	6
7	5

H(4)	B(4)
0	3
1	1
2	9
3	8
4	2
5	0
6	4
7	10
8	6
9	7
10	12
11	14
12	11
13	5
14	13
15	15

H(5)	B(5)
0	17
1	3
2	12
3	6
4	2
5	1
6	5
7	0
8	24
9	20
10	18
11	9
12	8
13	16
14	4
15	10
16	14
17	7
18	25
19	19
20	23
21	15
22	11
23	31
24	28
25	26
26	22
27	13
28	29
29	30
30	27
31	21

Table 1: Embeddings f_D of $B(D)$ in $H(D)$ with dilation 2.

Although it does not show in the tables, the given embeddings have "some" symetry, but we have not found a general rule for it.
We have explored three directions to find these embeddings, hoping to find a general pattern. The first one is by computer, but it gives embedding without any regularity. The second one is by trying to find symetries based on the binary representation of the vertices; this is the case for f_D, $D \leq 4$. In the third one, we try to first optimize the dilation of the embeddings of the cycles of $B(D)$ of the form $(x_1x_2\cdots x_D,\ x_2x_3\cdots x_Dx_1,\ \cdots,x_Dx_1\cdots x_{D-1})$ in $H(D)$. We obtained f_5 in that way (the first embedding of $B(5)$ in $H(5)$ with dilation 2 was found by computer).

5.2 Embeddings of S(D)

For $D = 1$, $S(1)$ and $H(1)$ are isomorphic and we take the identity function as the embedding.

We give in the following table embeddings f'_D of $S(D)$ in $H(D)$ for $2 \leq D \leq 6$.

H(2)	S(2)
0	0
1	3
2	1
3	2

H(3)	S(3)
0	0
1	1
2	5
3	4
4	3
5	2
6	6
7	7

H(4)	S(4)
0	1
1	2
2	0
3	3
4	14
5	7
6	15
7	6
8	8
9	4
10	9
11	5
12	13
13	11
14	12
15	10

H(5)	S(5)
0	6
1	7
2	18
3	19
4	3
5	2
6	17
7	16
8	12
9	13
10	24
11	25
12	15
13	14
14	29
15	28
16	5
17	4
18	9
19	8
20	0
21	1
22	20
23	21
24	23
25	22
26	27
27	26
28	10
29	11
30	30
31	31

H(6)	S(6)	H(6)	S(6)
0	34	32	48
1	6	33	40
2	35	34	49
3	7	35	41
4	5	36	17
5	3	37	33
6	4	38	16
7	2	39	32
8	24	40	36
9	12	41	18
10	25	42	37
11	13	43	19
12	10	44	9
13	0	45	20
14	11	46	8
15	1	47	21
16	29	48	57
17	15	49	53
18	28	50	56
19	14	51	52
20	46	52	58
21	30	53	60
22	47	54	59
23	31	55	61
24	51	56	45
25	39	57	27
26	50	58	44
27	38	59	26
28	23	60	54
29	63	61	43
30	22	62	55
31	62	63	42

Table 2: Embeddings f'_D of $S(D)$ in $H(D)$ with dilation 2 for $3 \leq D \leq 6$.

The embeddings f'_{D+1}, for $1 \leq D \leq 5$, are obtained from the embeddings f_D of $B(D)$ in $H(D)$ by splitting every vertex $x_1x_2\cdots x_D$ of $B(D)$ in two vertices $x_1x_2\cdots x_D0$ and $x_1x_2\cdots x_D1$ of $S(D+1)$ which are mapped on adjacent vertices of $H(D+1)$.

Furthermore for each $D \leq 5$ we have $dil(f'_{D+1}) \leq dil(f_D)$. It would be interesting to know if this construction is always possible, for it would give $dil(S(D+1), H(D+1)) \leq dil(B(D), H(D))$.

References

[1] J.-C. Bermond and C. Peyrat. de Bruijn and Kautz networks: a competitor for the hypercube? *Proceedings of the 1st European Workshop on Hypercubes and Distributed Computers, Rennes, North Holland, F.Andre and J.P.Verjus ed.*, pages 279–293, 1989.

[2] S. Bhatt, F. Chung, T. Leighton, and A. Rosenberg. Optimal simulations of tree machines. *IEEE*, pages 274–282, 1986.

[3] M. Bouabdallah and J.-C. Konig. Embedding de Bruijn networks in the hypercube. *Preprint*, 1990.

[4] M. Chan. Embedding of d-dimensional grids into optimal hypercubes. *1st ACM Symposium on Parallel Algorithms and Architectures*, pages 52–57, 1989.

[5] O. Collins, S. Dolinar, R. McEliece, and F. Pollara. A VLSI decomposition of the de Bruijn graph. *Preprint*, 1989.

[6] N. de Bruijn. A combinatorical problem. *Koninklijke Nederlandsche Akademie van Wetenschappen Proc.*, A 49:758–764, 1946.

[7] R. Feldmann and W. Unger. The cube connected cycle network is a subgraph of the butterfly network. Technical report, University of Paderborn, 1991.

[8] D. Greenberg and S. Bhatt. Routing multiple paths in hypercubes. *Proceedings of SPAA*, pages 45–54, 1990.

[9] D. S. Greenberg, L. S. Heath, and A. Rosenberg. Optimal embeddings of butterfly-like graphs in the hypercube. *Mathematical Systems Theory*, 23:61–77, 1990.

[10] C.-T. Ho and S. L. Johnson. Embedding meshes in boolean cubes by graph decomposition. *Journal of Parallel and Distributed Computing*, 8:325–339, 1990.

[11] R. Koch, T. Leighton, B. Maggs, S. Rao, and A. Rosenberg. Work-preserving emulations of fixed-connection networks. *Preprint.*

[12] B. Monien and H. Sudborough. Comparing interconnection networks. *Preprint*, 1988.

[13] E. J. Schwabe. On the computational equivalence of hypercube-derived networks. *2nd Symposium on Parallel Algorithms and Architectures*, pages 388–397, 1990.

[14] P. M. Winkler. The metric structure of graphs. *Surveys in Combinatorics, (C. Whitehead, ed.), London Math. Soc. Lecture Notes Series*, 123:197–221, 1987.

MAPPING UNIFORM RECURRENCES ONTO SMALL SIZE ARRAYS

Vincent Van Dongen
Philips Research Laboratory
4 Av. Albert Einstein, B-1348 Louvain-la-Neuve, Belgium
e-mail: vvd@prlb.philips.be

Abstract

Given a regular application described by a system of uniform recurrence equations, systolic arrays are commonly derived by means of an affine transformation; an affine schedule determines when the computations are performed and an affine processor allocation where they are performed. Circuit transformations are then applied on the resulting circuit when the application needs to be mapped onto a smaller size array. This method is in two steps and thus can hardly be optimized globally.

We hereafter present a different method for designing small size arrays. We derive them in one step by means of an affine schedule and a near-affine processor allocation. By doing so, we can generalize the optimization technique for affine mapping to be applicable here. The method is illustrated on the band-matrix multiplication and on the convolution algorithms.

1 Introduction

Systolic arrays are particular circuits made of identical processing elements connected in a local and regular manner; a high throughput is achieved by making extensive use of parallelism and pipelining. The regularity and locality of their connections make them ideally suited for a VLSI implementation. Such arrays are applicable to problems in signal processing, numerical computing, graph theory and other areas [KL78].

Given an application described by a system of uniform recurrence equations, systolic arrays are derived when using affine space-time transformations; an affine schedule determines when the computations are performed and an affine processor allocation where they are performed [Qui84,Rao85]. This space-time mapping technique is generalized in this paper to derive small size arrays, i.e. arrays containing fewer cells than the ones obtained when using affine mappings.

The problem of realizing automatically small size arrays is important; its applications can be classified as follows:

1. *Software compilation for general-purpose arrays.*
   The problem is encountered in the mapping of applications, here systems of URE's, on fixed size general-purpose arrays, e.g. a number of Transputers [The89] or the Warp [AAG*87].

2. *Hardware compilation.*
   Small size circuits may be required due to area constraints.

The common approach for deriving small systolic arrays consists in two steps [Bu90,Cla90,ND88,DI88,GN89]. An array is first derived by means of an affine space-time mapping. This array usually contains too many processors. The schedule is then slowed down so that less parallelism is achieved, and the processor allocation is modified correspondingly. This second step is a particular circuit transformation.

Our approach is different. we derive small size arrays directly from the behavioral description by means of a unique space-time mapping , as shown in figure 1. The advantage of this direct approach is that it can be optimized. Its drawback is that it only works with a particular partitioning strategy known as the *Locally Sequential Globally Parallel* (LSGP) partitioning scheme [Bu90,ND88,Kun87,GN89]. But this scheme is ideal when using a general-purpose processor array as an array of Transputers [The89,Bra90].

In the next section, systems of uniform recurrences are introduced. Then, in section 3, near-affine mappings are defined. In particular, it is shown that these mappings can be viewed as a set of affine transformations defined on lattices of the index space. In section 4, constraints for dealing with these mappings are derived. In section 5, we recall how an affine schedule can be optimized. In section 6, we generalize the optimization method to deal with near-affine mappings. We summarize the complete methodology in section 7, and we conclude in section 8.

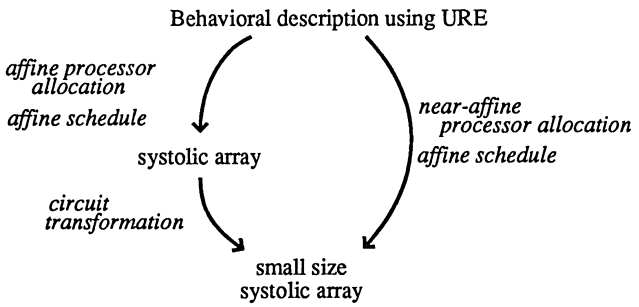


Figure 1: The common approach for deriving small systolic arrays consists in two steps. In this paper, small size arrays are derived directly from the behavioral description by means of a unique space-time mapping.

2 Uniform recurrence equations

We follow the suggestion given by Karp, et. al. [KMW67] of describing "regular" algorithms (for which systolic arrays are suited) using uniform recurrence equations (URE). Quinton was first in recognizing that such a description can be used in the synthesis of systolic arrays [Qui84].

Definition 2.1 A system of *uniform recurrences* is a set of m recurrences of the form

$$\begin{array}{ll} \mathbf{z} \in \mathcal{D}_s \rightarrow & O_1(\mathbf{z}) = f_1(O_{1_1}(\mathbf{z} - \vartheta_{1,1}), O_{1_2}(\mathbf{z} - \vartheta_{2,1}), ..., O_{1_{l_1}}(\mathbf{z} - \vartheta_{l_1,1})) \\ \mathbf{z} \in \mathcal{D}_s \rightarrow & O_2(\mathbf{z}) = f_2(O_{2_1}(\mathbf{z} - \vartheta_{1,2}), O_{2_2}(\mathbf{z} - \vartheta_{2,2}), ..., O_{2_{l_2}}(\mathbf{z} - \vartheta_{l_2,2})) \\ ... & \\ \mathbf{z} \in \mathcal{D}_s \rightarrow & O_m(\mathbf{z}) = f_n(O_{m_1}(\mathbf{z} - \vartheta_{1,m}), ..., O_{m_{l_m}}(\mathbf{z} - \vartheta_{l_m,m})) \end{array} \quad (1)$$

where

$$\forall v \in [1, m], \forall u \in [1, l_v] \rightarrow \vartheta_{u,v} \in \mathbf{Z}^e, \quad (2)$$

and $\mathcal{D}_s$ is a convex polyhedron of $\mathbf{Z}^e$ parameterized with s (the number of operands of f_v is noted l_v). o

The vectors $\vartheta_{u,v}$ will be called the *dependence vectors* of (1), and $O_v(\mathbf{z})$ will be called an *instance* of the variable O_v.

Example 2.1 The convolution algorithm (for FIR filtering) can be implemented with the following system of URE:

$$\begin{array}{lll} i \geq 1,\ 1 \leq k \leq N & \rightarrow & y(i,k) = y(i,k-1) + p(i,k) \\ i \geq 1,\ 1 \leq k \leq N & \rightarrow & p(i,k) = w(i,k) \times x(i,k) \\ i \geq 1,\ 1 \leq k \leq N & \rightarrow & w(i,k) = w(i-1,k) \\ i \geq 1,\ 1 \leq k \leq N & \rightarrow & x(i,k) = x(i-1,k-1) \end{array} \tag{3}$$

The values of $x(i,0)$ are initialized with the input signal x_i, and the outputs are $y_i = y(i,N)$.

□

Example 2.2 The band-matrix multiplication can be computed with the following system of URE:

$$\begin{array}{l} C(i,j,k) = C(i,j,k-1) + A(i,j,k) \times B(i,j,k) \\ A(i,j,k) = A(i-1,j,k) \\ B(i,j,k) = B(i,j-1,k) \end{array} \tag{4}$$

The index domain is defined by the following constraints:

$$0 \leq i \leq N-1, 0 \leq j \leq N-1, 0 \leq k \leq N-1, -w_1 \leq i-k \leq w_1, -w_2 \leq j-k \leq w_2. \tag{5}$$

The domain is a polyhedron having 16 vertices. The size parameters w_1 and w_2 represent half band widths; A is of band width $2.w_1 + 1$ and B is of band width $2.w_2 + 1$.

□

3 Space-time mapping

We suggest to use space-time mappings of the form

$$\mathbf{Z}^e \rightarrow \mathbf{Z}^e : (\mathbf{z}, v) \rightarrow \begin{pmatrix} t(\mathbf{z},v) & = \mathbf{T}.\mathbf{z} + \alpha_v \\ p_1(\mathbf{z},v) & = (\mathbf{P}_1.\mathbf{z} + \beta_{v,1}) \text{ div } d_1 \\ p_2(\mathbf{z},v) & = (\mathbf{P}_2.\mathbf{z} + \beta_{v,2}) \text{ div } d_2 \\ \ldots & \\ p_{e-1}(\mathbf{z},v) & = (\mathbf{P}_{e-1}.\mathbf{z} + \beta_{v,e-1}) \text{ div } d_{e-1} \\ v' & = v \end{pmatrix} \tag{6}$$

where " div " denotes the integer division, and $d_l \in \mathbf{N}_0$, $\forall l \in [1, e-1]$. In matricial form, the mapping is written (we omit $v' = v$):

$$\begin{pmatrix} t(\mathbf{z},v) \\ \mathbf{p}(\mathbf{z},v) \end{pmatrix} = \begin{pmatrix} \mathbf{T}.\mathbf{z} + \alpha_v \\ (\mathbf{P}.\mathbf{z} + \beta_v) \text{ div } \mathbf{d}_p, \end{pmatrix} \tag{7}$$

where $\mathbf{P} \in \mathbf{Z}^{e-1} \times \mathbf{Z}^e$, $\beta_v \in \mathbf{Z}^{e-1}$, $\mathbf{d}_p = (d_1, d_2, ..., d_e - 1) \in \mathbf{N}_0^{e-1}$ and "div" represents the component wise integer division. By definition, the processor allocation $\mathbf{p}(\mathbf{z},v)$ of

(7) is *near-affine*, a sub-class of quasi-affine mappings [Van91]. . The schedule $t(\mathbf{z}, v)$ is affine. Note that when $d_1 = d_2 = ... = d_{e-1} = 1$, the mapping (6) is affine.

With affine mappings, the computations that are processed by the same processor are on a line perpendicular to $\mathbf{P}_1$, $\mathbf{P}_2$, etc. The hyperplane perpendicular to $\mathbf{P}_l$ contains all the computations performed by the processors with the same l^{th} component. With a *near-affine* allocation, the computations performed by the processors with the same l^{th} component are on d_l consecutive hyperplanes perpendicular to $\mathbf{P}_l$. Thus, "bands" of indexed points are mapped onto the same processor. Within each band, the computations are performed in sequence by making use of the local memory of the processor.

Example 2.1 (cont'd) With the mapping

$$\begin{pmatrix} t(i,k,v) \\ p(i,k,v) \end{pmatrix} = \begin{pmatrix} 2.i + k \\ k \text{ div } 2 \end{pmatrix}, \tag{8}$$

one maps the uniform recurrences (3) with $N = 6$ onto the circuit shown in figure 2. It works as follows. Each input to the combinational logic is a two-input switch. Every clock cycle, the switches change of input. This is achieved with a control signal whose value alternates from one to zero. The control signal can either be broadcasted or it can be pipelined through the array as shown in figure 2. The signal values can be generated on line by a simple circuit that initializes its values when the *reset* is on. This part of the circuit is called the *control signal generator* in figure 2. The rate of the circuit is one-half; every two clock cycles, a new input signal enters the array, and an output value is being produced.

This can be easily generalized as follows. With the affine schedule

$$t(i,k,v) = R.i + k,$$

and the processor allocation

$$p(i,k,v) = k \text{ div } R.$$

where $R \geq 2$, one maps the uniform recurrences (3) onto an array of $M = \lceil N/R \rceil$ processors. Every R clock cycles, a new input value enters the array, and a new output is produced; the I/O rate of the array is $1/R$. Each processor contains one latch for y, and $R+1$ latches for x. The number of switches is one for x, and one for y. Note that in the complete array, the total number of latches is $N + M$ for x; it is proportional to N for any value of M. When $R \geq N$, the architecture is sequential.

The processors communicate with their neighbours once every R clock cycles only; they are loosely coupled which is ideal when using a general-purpose fixed size array with expensive inter-processor communication.

□

We now show that the near-affine mapping (7) transforms the system of URE (1) into a finite set of systems of URE defined on lattices of $(t, \mathbf{p}) \in \mathbf{Z}^e$. Let the *rational approximant* of (7) be:

$$\begin{pmatrix} t(\mathbf{z},v) \\ \tilde{\mathbf{p}}(\mathbf{z},v) \end{pmatrix} = \begin{pmatrix} \mathbf{T}.\mathbf{z} + \alpha_v \\ (\mathbf{P}.\mathbf{z} + \beta_v)/\mathrm{d}_p, \end{pmatrix} \tag{9}$$

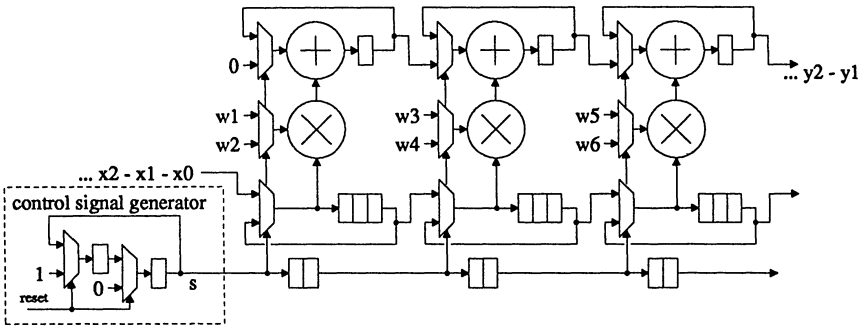


Figure 2: LSGP partitioning: the circuit is a systolic array that uses some internal memory and a simple control mechanism.

The mapping (7) can be written as:

$$\begin{pmatrix} t(\mathbf{z},v) \\ \mathbf{p}(\mathbf{z},v) \end{pmatrix} = \begin{pmatrix} \mathbf{T}.\mathbf{z}+\alpha_v \\ (\mathbf{P}.\mathbf{z}+\beta_v-\mathbf{r}_p)/\mathbf{d}_p \end{pmatrix} = \begin{pmatrix} t \\ \tilde{\mathbf{p}} \end{pmatrix} - \begin{pmatrix} 0 \\ \mathbf{r}_p/\mathbf{d}_p \end{pmatrix}, \tag{10}$$

with

$$\mathbf{r}_p = (\mathbf{P}.\mathbf{z}+\beta_v) \bmod \mathbf{d_p}. \tag{11}$$

The components of $\mathbf{r}_p$ are $(r_1, r_2, ..., r_{e-1})$, and for all $l \in [1, e-1]$, $r_l \in [0, d_l - 1]$. So, $\mathbf{r}_p$ has $d1 \times d_2 \times ... \times d_{e-1}$ distinct values. For each value of $\mathbf{r}_p$, the mapping (10) is affine over the lattice of $\mathbf{z}$ defined by (11). Thus, a near-affine mapping is a finite set of affine mappings defined over lattices of $\mathbf{z}$. More precisely, it is a rational affine mapping followed by a translation that depends on a lattice of $\mathbf{z}$; the rational approximant is followed by a *perturbation function*.

The inverse of a near-affine mapping is a finite set of affine mappings defined over lattices of $(t, \mathbf{p})$. The latter lattices can be computed as follows. The inverse of (10) is:

$$\begin{aligned} \mathbf{z} &= \tfrac{1}{a} \cdot adj \begin{pmatrix} \mathbf{T} \\ \mathbf{P} \end{pmatrix} \cdot \begin{pmatrix} t-\alpha_v \\ \mathbf{d}_p \times \mathbf{p} - \beta_v + \mathbf{r}_p \end{pmatrix} \\ &= \tfrac{1}{a} \cdot adj \begin{pmatrix} \mathbf{T} \\ \mathbf{P} \end{pmatrix} \cdot \begin{pmatrix} t-\alpha_v \\ \mathbf{d}_p \times \mathbf{p} - \beta_v \end{pmatrix} + \tfrac{1}{a} \cdot adj \begin{pmatrix} \mathbf{T} \\ \mathbf{P} \end{pmatrix} \cdot \begin{pmatrix} 0 \\ \mathbf{r}_p \end{pmatrix}, \end{aligned} \tag{12}$$

where a is the determinant of $\begin{pmatrix} \mathbf{T} \\ \mathbf{P} \end{pmatrix}$ and $adj \begin{pmatrix} \mathbf{T} \\ \mathbf{P} \end{pmatrix}$ denotes the adjoint matrix of $\begin{pmatrix} \mathbf{T} \\ \mathbf{P} \end{pmatrix}$. The index $\mathbf{z}$ is in $\mathbf{Z}^e$ if and only if

$$adj \begin{pmatrix} \mathbf{T} \\ \mathbf{P} \end{pmatrix} \cdot \begin{pmatrix} t-\alpha_v \\ \mathbf{d}_p \times \mathbf{p} - \beta_v + \mathbf{r}_p \end{pmatrix} \bmod \begin{pmatrix} a \\ \mathbf{a} \end{pmatrix} = \begin{pmatrix} 0 \\ \mathbf{0} \end{pmatrix}. \tag{13}$$

For a fixed value of $\mathbf{r}_p$, (12) is an affine mapping and (13) defines a lattice of $(t, \mathbf{p})$. More precisely, it is a rational affine transformation followed by a translation that depends on the lattice of $(t, \mathbf{p})$.

Example 2.1 (cont'd) The mapping (8) can be rewritten as:

$$\begin{pmatrix} t(i,k,v) \\ p(i,k,v) \end{pmatrix} = \begin{pmatrix} 2.i + k \\ (k - r)/2 \end{pmatrix},$$

where $r = k \bmod 2$. This mapping consists of two affine mappings defined on lattices of (i, k). Figures 3(a0) and 3(b0) show the first lattice and the result of the associated transformation. Figures 3(a1) and 3(b1) show the second lattice and the result of the associated transformation.

The inverse of (8) is:

$$\begin{pmatrix} i \\ k \end{pmatrix} = \frac{1}{2} \cdot \begin{pmatrix} 1 & -1 \\ 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} t \\ 2.p + r \end{pmatrix}. \tag{14}$$

The two lattices in $(t, \mathbf{p})$ are defined by $(t - 2.p - r) \bmod 2 = 0$.

□

Because of this, each uniform recurrence of (1) is transformed into a set of uniform recurrences, one for each lattice of $(t, \mathbf{p})$. Furthermore, the domain $\mathcal{D}_s$ is a convex polyhedron of $\mathbf{Z}^e$, and the image of $\mathcal{D}_s$ becomes the union of polyhedrons defined on lattices of $(t, \mathbf{p})$.

Finally, let us consider the control that must be added to the circuit to take into account the changes in the connections upon the lattices of $(t, \mathbf{p})$. It can be verified that any hyperplane $z_i = c$, where z_i is the i^{th} component of $\mathbf{z}$ and c is some constant, becomes

$$\mathbf{L}_i \cdot \begin{pmatrix} t - \alpha_v \\ \mathbf{p} - \beta_v + \mathbf{r}_p \end{pmatrix} = c.a,$$

where $\mathbf{L}_i$ is the i^{th} line of $adj \begin{pmatrix} \mathbf{T} \\ \mathbf{P} \end{pmatrix}$. This hyperplane of $(t, \mathbf{p})$ is exactly the one used to define the lattice in (13). Thus, a control signal can run along it to specify this lattice. At most e control signals will be required in the final array, one per axis.

Example 2.1 (cont'd) Consider the recurrence $x(i, k) = x((i, k) - (1, 1))$ of (3). When $k \bmod 2 = 0$, (i, k) is in the first lattice and $((i, k) - (1, 1))$ is in the second one. The application of the corresponding affine transformations gives:

$x(2.i + k, k/2) = x(2.(i - 1) + (k - 1), ((k - 1) - 1)/2)$ which is equivalent to:

$x(t, p) = x(t - 3, p - 1)$. On the other hand, when $k \bmod 2 = 1$, we obtain:

$x(2.i + k, (k - 1)/2) = x(2.(i - 1) + (k - 1), (k - 1)/2)$ which is equivalent to:

$x(t, p) = x(t - 3, p)$. Figure 4 shows these two cases.

Furthermore, the image of $\mathcal{D}_s$ is the union of polyhedrons defined on lattices of $(t, \mathbf{p})$. Consider the inequality $i \geq 0$ defining a boundary of the domain of (3). For the first

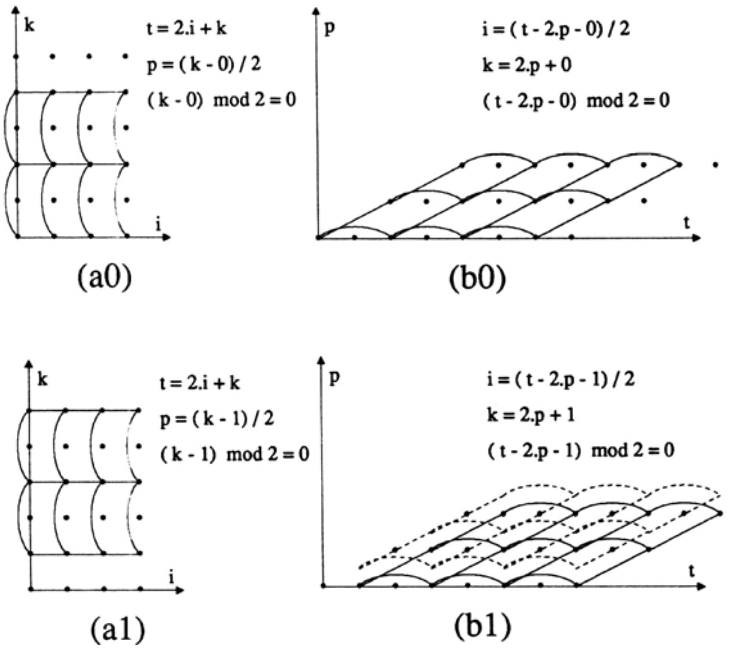


Figure 3: A near-affine mapping is a finite set of affine mappings: on the lattice shown in (a0), the first affine transformation gives the lattice shown in (b0), while on the lattice shown in (a1), the second transformation gives the lattice shown in (b1). It can also be obtained with a rational affine transformation followed by a translation that depends on a lattice of z; the result of the rational transformation is shown in dash in (b1).

lattice of (t, p), $i = (t-2.p-0)/2$, and the condition becomes: $(t-2.p-0)/2 \geq 0$. For the second lattice of (t, p), $i = (t-2.p-1)/2$, and the condition becomes: $(t-2.p-1)/2 \geq 0$. Figure 4 shows the two resulting boundaries in (t, p).

For the control, the hyperplane $i = 0$ becomes $t - 2.p - r = 0$. A control signal can run along it, i.e. along $(t, p) = (2, 1)$. (See figure 2.) The hyperplane $k = 0$ becomes $2.p + r = 0$. No control signal is needed along that plane since it does not define the lattices. (See figure 3.)

□

4 Constraints

The LSGP partitioning scheme is achieved with an affine schedule. The basic constraints on the schedule are [Qui84]:

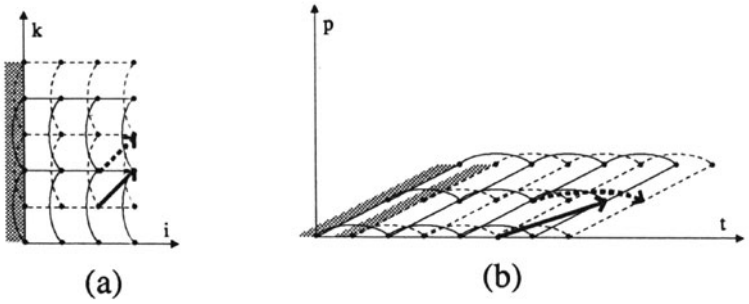


Figure 4: A near-affine mapping transforms a uniform recurrence into a finite set of uniform recurrences defined on lattices of $(t, \mathbf{p})$. It transforms a linear constraint of $\mathcal{D}_e$ into a finite set of linear constraints in $(t, \mathbf{p})$.

$$\forall v \in [1,m], \forall u \in [1,l_v] \rightarrow \mathbf{T} \cdot \vartheta_{u,v} + \alpha_v - \alpha_{v_u} \geq 1, \quad (15)$$

$$\forall v \in [1,m] \rightarrow \alpha_v \geq -min_{\mathbf{V}_k \in \mathcal{D}_e}(\mathbf{T}.\mathbf{V}_k), \quad (16)$$

where the $\mathbf{V}_k$'s are the vertices of $\mathcal{D}_e$. Furthermore, if the domain is semi-infinite along a ray $\mathbf{R}$, the schedule is subject to the additional constraint:

$$\mathbf{T} \cdot \mathbf{R} \geq 1. \quad (17)$$

On the other hand, the constraint on the processor allocation, when the domain is semi-infinite along $\mathbf{R}$, is

$$\forall l \in [1, e-1] \rightarrow \mathbf{P}_l \cdot \mathbf{R} = 0. \quad (18)$$

This must be satisfied with both affine and near-affine processor allocations.

Any two different instances of O_v that are computed in parallel must be performed on distinct processors. In other words, the mapping (6) must be *one-to-one*, We now consider this problem.

Let "w" denote the number of d_l's that are equal to one; $w \in [0, e-1]$. When $w = 0$, all d_l's are different from one, while when $w = e - 1$, the mapping is affine. In the following, let us assume that $d_1 = d_2 = ... = d_w = 1$ and that d_{w+1}, d_{w+2}, ..., d_{e-1} are different from one. (One can always order the $e-1$ axes of the processor allocation so that it is verified.) Furthermore, let g_l denote the greatest common divisor of the components of $\mathbf{P}_l$ (i.e. $g_l = gcd(\mathbf{P}_l)$), and $\mathbf{P}'_l = \mathbf{P}_l / g_l$. Also, we define g_t as $gcd(\mathbf{T})$ and $\mathbf{T}'$ as $\mathbf{T}' = \mathbf{T}/g_t$. Thus, $\mathbf{T}'$ and $\mathbf{P}'_l$ are integral vectors whose components are relatively prime. Let us summarize the results given in [Van91].

- The values g_t, g_1, ..., g_w have no effect on the condition for having a one-to-one mapping.

- For the mapping (6) to be one-to-one, the absolute value of the determinant of $(\mathbf{T}', \mathbf{P}'_1, ..., \mathbf{P}'_w, \mathbf{P}_{w+1}, ..., \mathbf{P}_{e-1})$ must be greater than $\prod_{l=1}^{e-1} d_l$.
- When $e = 2$, the mapping (7) is one-to-one if and only if

$$\left\| \begin{matrix} \mathbf{T}' \\ \mathbf{P}_1 \end{matrix} \right\| \geq d_1. \tag{19}$$

- When $e \geq 3$ and $w = 1$, a systematic method polynomial with e can be used to check that the mapping is one-to-one. check that the mapping is one-to-one when only one value of d_l is different from one.
- In the general case, the mapping (7) is one-to-one if and only if, $\forall v \in [1, m]$, there exists no pair of points in some reduced domains $\mathcal{R}_v$ that have the same image.

Because a near-affine mapping is a set of affine transformations on lattices of $\mathbf{z}$, another method for checking that it is one-to-one consists in verifying that the images of these lattices do not intersect. First, one can compute the non-empty lattices defined by (13), by solving systems of diophantine equations. Second, one can check that any pair of non-empty mappings do not intersect. This is simple to check since two lattices defined by (13) and characterized by $\mathbf{r}_p = \mathbf{r}$ and $\mathbf{r}_p = \mathbf{r}'$ do not intersect if and only if the following system of diophantine equations of vector variable $\mathbf{k}$ is empty:

$$\mathbf{A}.(\mathbf{r} - \mathbf{r}') = \mathbf{k}.\mathbf{a},$$

where $\mathbf{A}$ is the $e \times (e-1)$ matrix formed with the $e-1$ last columns of $adj \begin{pmatrix} \mathbf{T} \\ \mathbf{P} \end{pmatrix}$, and $\mathbf{a}$ is a vector whose e entries are equal to a.

Example 2.1 (cont'd) Consider the near-affine mapping (8) and its inverse (14). Here, $\mathbf{A} = \begin{pmatrix} -1 \\ 2 \end{pmatrix}$. Its two lattices do not intersect if and only if the system

$$\begin{pmatrix} -1 \\ 2 \end{pmatrix}.(0-1) = \begin{pmatrix} k_1 \\ k_2 \end{pmatrix}.(2)$$

has no solution, which is the case since $1 = 2.k_1$ has no solution.

□

Unidirectional arrays have specific properties. In particular, the so-called Locally Parallel Globally Sequential (LPGS) partitioning scheme is directly applicable on such arrays. Its advantage is to use the external memory instead of the local memory as with the LSGP scheme [Kun87,Mol83]. Unidirectional arrays can be automatically realized, by means of additional constraints. For example, the l^{th} component of $\mathbf{c}_{u,v}$ is positive if and only if

$$\mathbf{P}_l.\vartheta_{u,v} + \beta_{v,l} - \beta_{u,l} \geq 0.$$

5 Optimization of the schedule

In a system of URE, the index domain $\mathcal{D}_s$ is parameterized with s. Let $\mathcal{D}$ denote an instance of $\mathcal{D}_s$ when s has a given value. Given a finite dependence graph and an affine schedule, the *latency*, which will be noted Δt, is

$$\begin{aligned}\Delta t &= max_{\mathbf{z}\in\mathcal{D}}(t(\mathbf{z})) - min_{\mathbf{z}\in\mathcal{D}}(t(\mathbf{z})) + 1 \\ &= max_{j\in[1,h]}(\mathbf{T}.\mathbf{V}_j) - min_{i\in[1,h]}(\mathbf{T}.\mathbf{V}_i) + max_{v\in[1,m]}(\alpha_v) - min_{u\in[1,m]}(\alpha_u) + 1 \\ &= \Delta T + max_{v\in[1,m]}(\alpha_v) - min_{u\in[1,m]}(\alpha_u)\end{aligned}$$

with

$$\Delta T = max_{j\in[1,h]}(\mathbf{T}.\mathbf{V}_j) - min_{i\in[1,h]}(\mathbf{T}.\mathbf{V}_i) + 1$$

For large computation domains $\mathcal{D}$, the minimization of ΔT leads to a minimal value of Δt.

The problem of minimizing ΔT can be solved by means of integer programming. Let $D_{i,j}$ be the set of all non-null vectors $\mathbf{X}$ such that

$$\forall \mathbf{z} \in \mathcal{D} \rightarrow \mathbf{X}.\mathbf{V}_i \leq \mathbf{X}.\mathbf{z} \leq \mathbf{X}.\mathbf{V}_j.$$

The later condition is equivalent to the following set of linear constraints:

$$\begin{cases} \forall a \in [1, h_i] \rightarrow (\mathbf{V}_{i_a} - \mathbf{V}_i).\mathbf{X} \geq 0 \\ \forall b \in [1, h_j] \rightarrow (\mathbf{V}_j - \mathbf{V}_{j_b}).\mathbf{X} \geq 0 \end{cases},$$

where $\mathbf{V}_{i_a}$ and $\mathbf{V}_{j_b}$ denote the vertices of $\mathcal{D}$ connected respectively to $\mathbf{V}_i$ and to $\mathbf{V}_j$. By definition of $D_{i,j}$ it comes:

$$\forall \mathbf{T} \in D_{i,j} \rightarrow \Delta T = \mathbf{T}.(\mathbf{V}_j - \mathbf{V}_i) + 1. \tag{20}$$

Thus, the optimization can be achieved for any possible pair of vertices (V_i, V_j) of $\mathcal{D}$, and the global minimum will simply be the minimum of the local solutions.

Example 2.2 (cont'd) Assume that all $\Delta_{u,v}$'s are one. It can be shown that the only pair of vertices whose $D_{i,j}$ gives a solution is $(\mathbf{V}_i, \mathbf{V}_j) = ((0,0,0),(N-1,N-1,N-1))$. The optimal schedule is found by minimizing

$$\mathbf{T}.(1,1,1)$$

under the constraints $\mathbf{T}.(1,0,0) \geq 1$, $\mathbf{T}.(0,1,0) \geq 1$, and $\mathbf{T}.(0,0,1) \geq 1$. The solution is given by:

$$t(i,j,k) = i + j + k + \alpha_v, \tag{21}$$

with $\alpha_A = \alpha_B = 0$ and $\alpha_C = 1$. The associated number of time steps is $3.(N-1)+1$.

□

Instead of looking for $\mathbf{T}$ that minimizes ΔT, one may want to find $\mathbf{T}$ that minimizes some other cost function. This can easily be achieved when the solution space of $\mathbf{T}$ is

bounded. In that case, one can compute all possible solutions, and find the one that minimizes the cost.

The solution space of $\mathbf{T}$ can be bounded by imposing ΔT to be in a given range. Let $[\Delta T^u, \Delta T^v]$ denote this range. Because of (20), any slope $\mathbf{T}$ of $D_{i,j}$ has its ΔT in that range if and only if:

$$\Delta T^u - 1 \leq \mathbf{T}.(\mathbf{V}_j - \mathbf{V}_i) \leq \Delta T^v - 1. \tag{22}$$

Example 2.2 (cont'd) Let us for example compute all the slopes $\mathbf{T}$ whose $\Delta T \in [30, 40]$, when $N = 8$. The set of solutions is:

$$\{(1,1,3),(1,2,2),(1,3,1),(2,1,2),(2,2,1),(3,1,1)\}. \tag{23}$$

□

The same technique is also applicable to infinite computation domains. Yet, in that case, the quantity

$$max_{\mathbf{z}\in\mathcal{D}}(t(\mathbf{z})) - min_{\mathbf{z}\in\mathcal{D}}(t(\mathbf{z})) + 1$$

is always infinite. Hence, it cannot be used for minimizing the schedule. Given an infinite domain $\mathcal{D}$ of vertices $\mathbf{V}_1$, $\mathbf{V}_2$, ..., $\mathbf{V}_h$, and of ray $\mathbf{R}$, we define $\mathcal{D}'$ as the finite convex domain of vertices $\mathbf{V}_k$ and $\mathbf{V}_{h+k} = \mathbf{V}_k + \mathbf{R}$, $k \in [1, h]$. The edges of $\mathcal{D}'$ are the edges of $\mathcal{D}$, plus the edges between $\mathbf{V}_k$ and $\mathbf{V}_{h+k}$, plus the edges between $\mathbf{V}_{h+i}$ and $\mathbf{V}_{h+j}$ whenever there is an edge between $\mathbf{V}_i$ and $\mathbf{V}_j$.

The value of Δt can be defined on this restricted domain as

$$\Delta t = max_{\mathbf{z}\in\mathcal{D}'}(t(\mathbf{z})) - min_{\mathbf{z}\in\mathcal{D}'}(t(\mathbf{z})) + 1.$$

According to this definition, Δt is now a finite quantity which can be used in the comparison of different schedules of infinite dependence graphs.

Example 2.1 (cont'd) The restricted domain $\mathcal{D}'$ is here a rectangle of vertices $(1,1)$, $(1,N)$, $(2,N)$ and $(2,1)$. Let us assume that all $\Delta_{u,v}$'s are one. The only pair of vertices which lead to a valid schedule is $(\mathbf{V}_i, \mathbf{V}_j) = ((1,1),(2,N))$. The schedule that minimizes ΔT is found by minimizing $\mathbf{T}.(1, N-1)$ under the constraints $\mathbf{T}.(1,0) \geq 1$, $\mathbf{T}.(0,1) \geq 1$, and $\mathbf{T}.(1,1) \geq 1$. The solution is given by:

$$t(i,k) = i + k + \alpha_v,$$

with $\alpha_x = \alpha_w = -2$, $\alpha_p = -1$ and $\alpha_y = 0$.

□

6 Processor allocation optimization

Let us define ΔP_l as follows:

$$\Delta P_l = \left\lceil \frac{max_{\mathbf{z}\in\mathcal{D}}(\mathbf{P}_l.\mathbf{z}) - min_{\mathbf{z}\in\mathcal{D}}(\mathbf{P}_l.\mathbf{z}) + 1}{d_l} \right\rceil. \tag{24}$$

When the components of $\mathbf{P}_l$ are relatively prime, ΔP_l represents the number of cells along l, for each value of v. Else, it is a good approximation of the number of cells. The value of $\mathbf{P}_l$ that minimizes

$$max_{\mathbf{z}\in\mathcal{D}}(\mathbf{P}_l.\mathbf{z}) - min_{\mathbf{z}\in\mathcal{D}}(\mathbf{P}_l.\mathbf{z}) + 1 \tag{25}$$

clearly minimizes (24). Thus, when d_l is fixed, the optimization of ΔP_l becomes similar to the one of ΔT. For a given $D_{i,j}$, we have:

$$\forall \mathbf{P}_l \in D_{i,j} \rightarrow max_{\mathbf{z}\in\mathcal{D}}(\mathbf{P}_l.\mathbf{z}) - min_{\mathbf{z}\in\mathcal{D}}(\mathbf{P}_l.\mathbf{z}) + 1 = \mathbf{P}_l.(\mathbf{V}_j - \mathbf{V}_i) + 1.$$

This later quantity can be minimized by means of integer programming.

Example 2.2 (cont'd) Assume again that $\mathbf{s} = (N, w_1, w_2) = (8, 2, 3)$. In that case, $\mathcal{D}$ has 16 vertices as shown in figure 4. Consider the pair $(\mathbf{V}_i, \mathbf{V}_j) = ((0, w_1 + w_2, w_1), (w_1 + w_2, 0, w_2))$. The associated integer programming problem is to minimize

$$(w_1 + w_2, -w_1 - w_2, w_2 - w_1).\mathbf{P}_l = (5, -5, 1).\mathbf{P}_l,$$

under the constraints

$$(5,-5,1).\mathbf{P}_l \geq 1 \text{ (to avoid } \mathbf{P}_l = \mathbf{0}),\ (1,1,1).\mathbf{P}_l \geq 0,\ (0,-1,0).\mathbf{P}_l \geq 0,$$
$$(0,-1,-1).\mathbf{P}_l \geq 0,\ (-1,-1,-1).\mathbf{P}_l \geq 0, (1,0,0).\mathbf{P}_l \geq 0,\ (1,0,1).\mathbf{P}_l \geq 0.$$

The optimal slope is $\mathbf{P}_l = (1, 0, -1)$. The associated ΔP_l is $2.w_1 + 1 = 5$. The optimal rational solution is therefore $\mathbf{P}_l/d_l = (1, 0, -1)/d_l$. It can be shown that this solution is a global optimum; no other pair of vertices can yield to a better solution. (There is only one equivalent solution which is $(-1, 0, 1)/d_l$.)

□

It is also possible to find all the slopes $\mathbf{P}_l$ such that ΔP_l is in a given range. Let $[\Delta P_l^u, \Delta P_l^v]$ denote this range. It can be verified that any slope $\mathbf{P}_l \in D_{i,j}$ has its associated ΔP_l in that range if and only if:

$$d_l.\Delta P_l^u - d_l \leq \mathbf{P}_l.(\mathbf{V}_j - \mathbf{V}_i) \leq d_l.\Delta P_l^v - 1. \tag{26}$$

This is direct generalization of what was done on the schedule, and an application of:

$$\lceil a/b \rceil \geq c \iff a \geq b.(c-1) + 1$$
$$\lceil a/b \rceil \leq c \iff a \leq b.c$$

The constraint (26) bounds the solution space of $\mathbf{P}_l$.

Example 2.2 (cont'd) Consider the problem of finding all the slopes $\mathbf{P}_l$ such that ΔP_l is in $[5,10]$ and $d_l = 1$. When fixing $(\mathbf{V}_i, \mathbf{V}_j)$ to $((0, w_1 + w_2, w_1), (w_1 + w_2, 0, w_2))$, the set of solutions is:

$$\{(1,0,-1),(0,-1,1),(2,0,-2)\}.$$

When considering all pair of vertices, the complete set of solutions for $\mathbf{P}_l$ is:

$$\{\pm(1,0,-1),\pm(0,-1,1),\pm(2,0,-2),\pm(1,0,0),\pm(0,1,0),\pm(0,0,1)\}. \qquad (27)$$

□

7 Design methodology

The problem is to find a mapping of the form (6) that verifies a particular number of design constraints. As explained in the previous sections, a set of slopes $\mathbf{T}$ that verifies a number of constraints and that minimizes ΔT can be found by means of integer programming. Also, the set of slopes whose ΔT is in a given range can be found with a similar technique. The same applies on each axis of the processor allocation separately (for each $\mathbf{P}_l$).

Once a set of solutions has been found for $\mathbf{T}$ and for each $\mathbf{P}_l$, the mappings of the form (6) can be found by simply combining the different solutions. For each combination, one can verify that the mapping is one-to-one, as explained in section 4; only the one-to-one mappings should be kept. If no compatible solution exists, one can either modify the value of one d_l or modify one range of values, and restart the process. The search for one-to-one mappings is clearly an iterative process.

Finally, a cost function can be evaluated on every one-to-one mapping to select the optimal solutions. The methodology is summarized in figure 5. It has been implemented with success in the design tool Presage [VP90].

Example 2.2 (cont'd) The minimal value of ΔT is 22. It is achieved with $\mathbf{T} = (1,1,1)$. When $d_1 = d_2 = 1$, the minimal value of ΔP_l is 5; $\mathbf{P}_l = \pm(1,0,-1)$. Yet, the mapping such that $\mathbf{T} = (1,1,1)$, $\mathbf{P}_1 = (1,0,-1)$ and $\mathbf{P}_2 = (-1,0,1)$ is not one-to-one. (The corresponding value of a' is zero.)

A first solution consists in relaxing the admissible range for ΔP_2. For example, if this range is $[5,10]$, the set of solutions for $\mathbf{P}_2$ is given by (27). Out of all combinations, four mappings only are one-to-one; they are all equivalent and given by

$$\begin{pmatrix} \mathbf{T} \\ \mathbf{P}_1 \\ \mathbf{P}_2 \end{pmatrix} = \begin{pmatrix} i+j+k+\alpha_v \\ \pm(1,0,-1).(i,j,k) \\ \pm(0,1,-1).(i,j,k) \end{pmatrix},$$

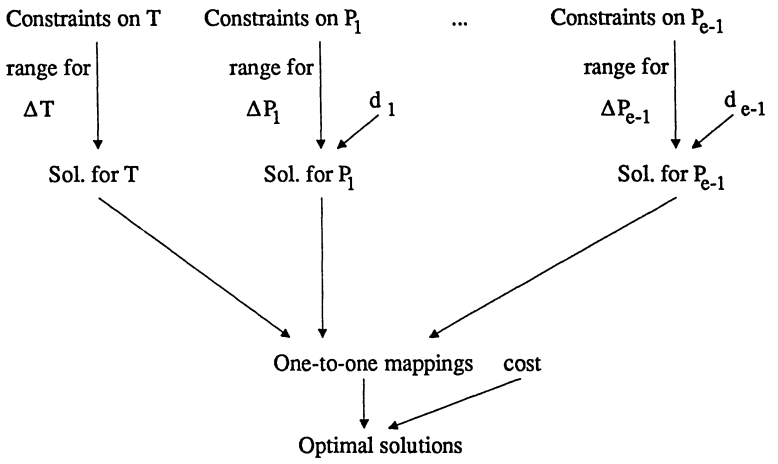


Figure 5: The main structure of the adopted methodology.

with $\alpha_A = \alpha_B = 0$ and $\alpha_C = 1$. All solutions lead to the same well-known systolic array presented by Kung and Leiserson in [MC80].

□

Example 2.2 (cont'd) The value of a' corresponding to the previous solution is 3, while $d_1.d_2 = 1$; each processor works once every three steps only. An implementation that requires a third of the cells can be achieved with $d_1 = 3$, e.g. with the mapping:

$$\begin{pmatrix} \mathbf{T} \\ \mathbf{P_1} \\ \mathbf{P_2} \end{pmatrix} = \begin{pmatrix} i+j+k+\alpha_v \\ (1,0,-1).(i,j,k) \text{ div } 3 \\ (0,1,-1).(i,j,k) \end{pmatrix}$$

It can be verified that this mapping is indeed one-to-one. When $w_1 = 2$ and $w_2 = 3$, the number of cells is $((2.2+1) \text{ div } 3) \times (2.3+1)$.

□

Example 2.2 (cont'd) The value of a' corresponding to the previous solution is 3, while $d_1.d_2 = 1$; each processor works once every three steps only. An implementation that requires a third of the cells can be achieved with $d_1 = 3$, e.g. with the mapping:

$$\begin{pmatrix} \mathbf{T} \\ \mathbf{P_1} \\ \mathbf{P_2} \end{pmatrix} = \begin{pmatrix} i+j+k+\alpha_v \\ (1,0,-1).(i,j,k) \text{ div } 3 \\ (0,1,-1).(i,j,k) \end{pmatrix}$$

It can be verified that this mapping is indeed one-to-one. When $w_1 = 2$ and $w_2 = 3$, the number of cells is $((2.2+1) \text{ div } 3) \times (2.3+1)$.

□

Example 2.2 (cont'd) In the previous example, an array of 2×7 cells is used. Assume that the problem is to be mapped on an array of size 8. One can either use an array of size 2×4 or a linear (i.e. one-D) array. In the first case, we can keep $d_1 = 3$ and $p_1(i,j,k) = (1,0,-1).(i,j,k)$ which give $\Delta P_1 = 2$. We can use $d_2 = 2$ and $p_2(i,j,k) = (0,1,-1).(i,j,k)$ div 2 to obtain $\Delta P_2 = 4$. The schedule that minimizes the latency can then be found as follows.

First, one can try with the minimal schedule (21), but the corresponding mapping is not one-to-one. One can then use a range for ΔT. The minimal value being $8.3-2 = 22$, one can first try with the range $[23,30]$. It can be shown that the associated set of solutions is:

$$\{(1,1,2),(1,2,1),(2,1,1)\}.$$

But still, none of these solutions yield to a one-to-one mapping. One can then try with the range $[31,40]$; the set of solutions for $\mathbf{T}$ is given by (23). Still, no schedule gives a one-to-one mapping. One can then try with the range $[41,45]$ whose set of solutions is:

$$\{(1,1,4),(1,2,3),(2,1,3),(1,3,2),(2,2,2),(3,1,2),(1,4,1),(2,3,1),(3,2,1),(4,1,1)\}.$$

It can be verified that the any slope of $\mathbf{T}$ in the set:

$$\{(4,1,1),(1,3,2),(2,1,3),(2,3,1).\}$$

gives a one-to-one mapping.

For a linear array, one can use $d_1 = 5$ and $d_2 = 1$. In that case,

$$\begin{array}{ll} p_1 & = (1,0,-1).(i,j,k) \text{ div } 5 \\ p_2 & = (i,j,k).(0,1,-1) \end{array} \tag{28}$$

The same iterative process can be used to find a compatible schedule. It can be verified that any slope $\mathbf{T}$ in (23) yields to a one-to-one mapping.

□

8 Conclusion

Given a regular application described by a system of uniform recurrence equations, systolic arrays are commonly derived by means of an affine space-time mapping. In this paper, we generalized the associated methodology to design small size arrays, by using an affine schedule and a near-affine processor allocation, a sub-class of quasi-affine mappings [Van91]. We showed that a near-affine processor allocation that uses a given number of processors can be automatically derived; the optimization method is a direct extension to the one using affine mappings.

In the proposed approach, the schedule and each component of the processor allocation are found independently. Ranges for the number of time steps and for the number

of processors are given by the user to delimit each solution space. Compatible solutions, i.e. one-to-one mappings, are then found by exhaustive search. A cost function finally selects the optimal solutions.

The associated partitioning method is the well-known *Locally Sequential Globally Parallel* scheme [ND88,GN89,Kun87]. This method requires large local memories for the processing elements. Other partitioning techniques are known, buth these cannot be found directly with our space-time mapping technique. Yet the advantage of our approach compared to the common one where circuit transformations are applied on systolic arrays derived with affine mappings is that optimal solutions can be found automatically.

As a particular case, we can optimally map e-D (i.e. e-dimensional) recurrences onto E-D arrays, where E is any value between 0 and $e-1$. One simply needs to fix the number of coordinates to 1 along $e-E-1$ axes of the processor allocation, and find a valid near-affine mapping. This method is to be compared with the *multiprojection* technique introduced in [WD85], which consists in applying an affine mapping k times to reduce the dimension of the array to $e-k$. Again, an optimal global solution can hardly be found with the latter technique, while it can when doing the space-time mapping in one pass.

Most of the theory presented in this paper has been implemented with success in the tool named "Presage" [VP90]. In fact, the illustrative mappings were derived with its use.

References

[AAG*87] M. Annaratone, E. Arnould, T. Gross, H.T. Kung, M. Lam, O. Menzilcioglu, and J.A. Webb. The warp computer : architecture, implementation, and performance. *IEEE Trans. on Computers*, C-36(12):1523–1538, December 1987.

[Bra90] H. Brams. *Adaptation du logiciel Presage à la génération de réseaux systoliques implémentables sur un réseau de Transputers.* Technical Report RR 90-23, Université Catholique de Louvain, Novembre 1990.

[Bu90] J. Bu. *Systematic Design of Regular VLSI Processor Arrays.* PhD thesis, Delft University of Technology, May 1990.

[Cla90] P. Clauss. *Synthèse d'Algorithmes Systoliques et Implantation Optimale en Place sur Réseaux de Processeurs Synchrones.* PhD thesis, Université de Franche-Comté, 1990.

[DI88] J.-M. Delosme and I.C.F. Ipsen. Sage and condense : a two-phase approach for the implementation of recurrence equations on multiprocessors architectures. In L.W. Hoevel, editor, *21st Annual Hawai Int. Conf. on System Sciences*, pages 126–130, 1988.

[GN89] M. Garcia and J. Navarro. Systematic hardware adaptation of systolic algorithms. In IEEE, editor, *The 16th Annual Int. Symp. on Comp. Architecture*, pages 96–104, Computer Society Press, 1989.

[KL78] H.T. Kung and C.E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proc. 1978*, pages 256–282, Society for Industrial and Applied Mathematics, 1978.

[KMW67] R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 1967.

[Kun87] S.Y. Kung. *VLSI array processors.* Signal and Image Processing Institute, 1987.

[MC80] C. Mead and L. Conway. *Introduction to VLSI Systems*, chapter 8, Highly Concurrent Systems, pages 263–332. Addison-Wesley Series in Computer Science, 1980.

[Mol83] D.I. Moldovan. On the design of algorithms for VLSI systolic arrays. *IEEE Proceedings*, 1983.

[ND88] H. Nelis and E. Deprettere. Automatic design and partitioning of systolic/wavefront arrays for vlsi. *Circuits Systems and Signal Processing*, Vol. 7(2):235–252, 1988.

[Qui84] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. In *Proc. IEEE 11-th Int. Sym. on Computer Architecture*, 1984.

[Rao85] S.K. Rao. *Regular iterative algorithms and their implementations on processor arrays.* PhD thesis, Information Systems lab., Standford University, 1985.

[The89] The transputer databook. 2nd ed. Inmos, Bristol, 1989.

[Van91] V. Van Dongen. *From Systolic to Periodic Array Design.* PhD thesis, Université Catholique de Louvain, January 1991.

[VP90] V. Van Dongen and M. Petit. Presage: a tool for the parallelization of nested loop programs. In L. Claesen (ed.), editor, *Formal VLSI Specification and Synthesis (VLSI Design Methods-I)*, pages 341–359, North-Holland, 1990.

[WD85] Y. Wong and J.M. Delosme. Optimal systolic implementations of n-dimensional recurrences. In *IEEE Int. Conf. on Computer Design: VLSI in Computers*, pages 618–621, Oct. 7-10 1985.

AREA COMPLEXITY OF MULTILECTIVE MERGING

PAVEL FERIANC
ONDREJ SÝKORA

Computing Centre, Slovak Academy of Sciences, Dúbravská 9
842 35 Bratislava, Czecho-Slovakia

Abstract. Lower bounds on the area A(n,m,k,r) required for merging of two sorted sequences of k-bit numbers with length n and m respectively, when the inputs can be replicated up to r times ($r \leq n$), are given:

$$A(n,m,k,r) = \begin{cases} \Omega\left(\frac{n}{r}\right) \text{ for } 2^k \geq \frac{n}{r} \text{ and } n \geq m \geq \frac{n}{r} \\ \Omega\left(m\left(\left(\log \frac{n}{rm}\right)+1\right)\right) \text{ for } 2^{\frac{3}{8}k} \geq \frac{n}{r} \text{ and } \frac{n}{r} \geq m \\ \Omega\left(m\left(\left(\log \frac{2^k}{m}\right)+1\right)\right) \text{ for } \frac{n}{r} \geq m \text{ and } \frac{n}{r} \geq 2^{\frac{3}{8}k} \text{ and } 2^{\left(\frac{3\cdot(8^K)-1}{8^{K+1}-1}k\right)} \geq m \\ \text{where } K \geq 0 \text{ is the constant} \end{cases}$$

INTRODUCTION

The paper analyzes the following problem: let $m,n \in N$ and $m \leq n$, let $D_n = \{(x_1,...,x_n) | x_1 \leq ... \leq x_n; x_i (i = 1,...,n)$ consist of k bits $\}$ and let $D_m = \{(z_1,...,z_m) | z_1 \leq ... \leq z_m; z_i (i = 1,...,m)$ consist of k bits $\}$. The merging problem can be characterized by the function $f : D_n \times D_m \rightarrow D_{n+m}$, such that $f(x_1,...,x_n,z_1,...,z_m) = (y_1,...,y_{n+m})$, where $X = \{x_1,...,x_n\}$ $\in D_n$, $Z = \{z_1,...,z_m\} \in D_m$, $Y = \{y_1,...,y_{n+m}\} \in D_{n+m}$ ($D_{n+m} = \{(y_1,\ldots,y_{n+m}) | y_1 \leq \cdots \leq y_{n+m}; y_i (i = 1,\ldots,n+m)$ consist of k bits $\}$) and each $y_i (1 \leq i \leq n+m)$ satisfies either $y_i \in X$ or $y_i \in Z$. Without loss of generality let us assume that m,n,r,k be powers of 2.

The memory of our circuit consists of square units of area (each one with area λ^2 ($\lambda > 0$)) and at most one bit can be stored per unit of area. The i/o schedule is assumed to be when- and where-determinate. Determinate schedules, which require prespecified times and locations for the input and output of each bit, are discussed in [**U**]. Tight lower bounds on the area required for merging with semelective inputs (each data is read once) were shown in [**PSV**]. Our area bounds are proved for r-multilective input; i.e. data can be read more than once but at most r-times. All computations and temporary storage of data, however, must be done within the merging device. According to [**Si**] we can assume w.l.o.g. that in a time unit t at most one input bit is supplied or at most one output bit is delivered and only one input or output event occurs.

The motivation for the study of the multilective circuits is that it leads either towards more general techniques for searching area bounds or it brings more general results. It is evident that multilectivity enables substantially diminished area of circuit. For example, in the work [G], a language with the following property is described. The area required for its recognition is $\Omega(\sqrt{n})$, but if allow each input to be read twice, then the area for its recognition is $O(1)$.

According to our knowledge until now there were proven only two results about nontrivial area lower bounds for concrete multilective problems. The area $A(n,k,r) = \Omega(\log n + 2^k \log \frac{n}{r2^{k-1}})$ for $2^k \leq \frac{n}{r}$; $A(n,k,r) = \Omega(\log n + \frac{n}{r} \log \frac{r2^{k+1}}{n})$ for $\frac{n}{r} \leq 2^k \leq n^{O(1)}$ for sorting n k-bit numbers is showed in [Si] and the area $A = \Omega(\frac{n^2}{r^2})$ for the matrix product of $n \times n$ matrices is shown in [Sa].

Now we give a brief introduction to the technique of our proofs(a similar technique was used in [Si]). A time interval τ of the entire time of computation is chosen. Call the inputs (outputs) which are read (delivered to output) during the interval τ τ-inputs (τ-outputs) and the inputs (outputs) for which no copy is read during the interval τ as non-τ-inputs (non-τ-outputs). Set the τ-inputs in a proper way so that the τ-outputs are dependent on the non-τ-inputs. The number of all various vectors of τ-outputs is equal to the number of all various vectors of corresponding non-τ-inputs, which is also the number of possible circuit states. The lower bound on the required area is the logarithm of the number of the circuit states.

LOWER BOUNDS

THEOREM 1. *If* $2^k \geq \frac{n}{r}$ *and* $n \geq m \geq \frac{n}{r}$ *, then* $A(n,m,k,r) = \Omega(\frac{n}{r})$.

To prove this theorem we need the next lemma.

LEMMA. *Let* $b_1 ... b_t$, *where* t *is even, be a string of zeros and ones such that the number of zeros is equal to the number of ones. Then arbitrary* $s \leq t$ *satisfies: there exists* j $(0 \leq j \leq t-s)$ *such that substring* $b_{j+1} ... b_{j+s}$ *contains at least* $\lfloor \frac{1}{3}s \rfloor$ *ones.*

PROOF OF LEMMA: Three cases are possible:

(1) $\lceil \frac{3}{4}t \rceil \geq s > \frac{1}{2}t$
If there are at least $\lfloor \frac{1}{3}s \rfloor$ ones in the substring $b_{s+1} ... b_t$ then $j = t - s$. Otherwise the substring $b_1 ... b_s$ must contain at least $\frac{t}{2} - (\lfloor \frac{1}{3}s \rfloor - 1) > \lfloor \frac{1}{3}s \rfloor$ ones.

(2) $t \geq s > \lceil \frac{3}{4}t \rceil$
Let $t = s + i$. Hence $i < \lfloor \frac{1}{3}s \rfloor$. Let us assume the substring $b_{s+1} ... b_t$ contains only ones (it means i ones). Then the string $b_1 ... b_s$ must contain at least $\frac{s+i}{2} - i > \lfloor \frac{1}{3}s \rfloor$ ones.

(3) $s \leq \frac{1}{2}t$
Let us divide the string $b_1 ... b_t$ to three substrings so that two substrings of them have the length s and the last one has the length i. Let in each of these substrings be at most $(\lfloor \frac{1}{3}s \rfloor - 1)$ ones i.e. $3(\lfloor \frac{1}{3}s \rfloor - 1)$ ones together. However, in the whole string $b_1 ... b_t$ there must be $\frac{2s+i}{2}$ ones. Since $3(\lfloor \frac{1}{3}s \rfloor - 1) < \frac{2s+i}{2}$ there exists one substring, which contains at least $\lfloor \frac{1}{3}s \rfloor$ ones. ∎

PROOF OF THEOREM 1: W.l.o.g. assume that $n \geq 12r$. Let us divide the time interval $< 0, T >$, where T is the time of the computation, into 2r intervals so that in each of them exactly $\frac{n}{4r}$ least significant bits of variables $y_{v+1}, y_{v+2}, ..., y_{v+\frac{n}{2}}$ are delivered to output, where $v \geq \frac{m}{2}$ and $v + \frac{n}{2} \leq n$. There exists an interval τ such that during τ at most $\frac{m}{4}$ copies of the least significant bits of the variables $z_1, ..., z_{\frac{m}{2}}$ are read. This implies that at least $\frac{m}{4}$ least significant bits of variables $z_1, ..., z_{\frac{m}{2}}$ are non-τ-inputs. Let $z_{j_1,k}; ...; z_{j_{\frac{m}{4}},k}$, where $j_{\frac{m}{4}} \leq \frac{m}{2}$ and $j_1 < ... < j_{\frac{m}{4}}$, be these bits. Let us indicate the least significant bits of variables $y_{v+1}, y_{v+2}, ..., y_{v+\frac{n}{2}}$, delivered to output during the interval τ (τ-outputs) as $y_{i_1,k}; ...; y_{i_{\frac{n}{4r}},k}$, where $i_1 < ... < i_{\frac{n}{4r}}$.

Now assign

$$x_{1,k} = ... = x_{n,k} = 0$$

$$z_{l,k} = 0 \text{ for } l \in \{1, ..., m\} - \{j_1, ..., j_{\frac{m}{4}}\}.$$

We are going to set so the bits $z_{i,1}; ...; z_{i,k-1}$ for $i = 1, ..., m$ and $x_{j,1}; ...; x_{j,k-1}$ for $j = 1, ..., n$ that as many input bits $z_{j_1,k}; ...; z_{j_{\frac{m}{4}},k}$ as possible are delivered to the output as $y_{i_1,k}; ...; y_{i_{\frac{n}{4r}},k}$. So, if $z_{t,k}$ for $t \in \{j_1, j_2, ..., j_{\frac{m}{4}}\}$ is delivered to output as $y_{s,k}$ for $s \in \{i_1, i_2, ..., i_{\frac{n}{4r}}\}$ then:

$$\text{if } z_{t,k} = 0 \text{ then } y_{s,k} = 0$$

and

$$\text{if } z_{t,k} = 1 \text{ then } y_{s,k} = 1.$$

By lemma 1 there exists j ($0 \leq j \leq \frac{m}{2} - \frac{n}{4r}$) such that the string $z_{j+1,k}; ...; z_{j+\frac{n}{4r},k}$ contains $\lfloor \frac{1}{3} \frac{n}{4r} \rfloor = \lfloor \frac{n}{12r} \rfloor$ bits from the set $\{z_{j_1,k}; ...; z_{j_{\frac{m}{4}},k}\}$. Let us indicate these bits by $z_{q_1,k}; ...; z_{q_{\lfloor \frac{n}{12r} \rfloor},k}$, where $q_1 < ... < q_{\lfloor \frac{n}{12r} \rfloor}$. Now we must find a setting for bits $z_{i,1}; ...; z_{i,k-1}$ where $i = 1, ..., m$ and for the bits $x_{j,1}; ...; x_{j,k-1}$ where $j = 1, ..., n$ such that

$$z_{j+1,k} \text{ is delivered to the output as } y_{i_1,k}$$

$$z_{j+2,k} \text{ is delivered to the output as } y_{i_2,k}$$

$$...$$

$$z_{j+\frac{n}{4r},k} \text{ is delivered to the output as } y_{i_{\frac{n}{4r}},k}$$

and the bits $\{z_{q_1,k}; ...; z_{q_{\lfloor \frac{n}{12r} \rfloor},k}\} \subset \{z_{j+1}, ..., z_{j+\frac{n}{4r}}\}$ are non-τ-inputs, whereby the bit

$$z_{q_1,k} \text{ is delivered to the output as } y_{i_{d_1},k}$$

$$z_{q_2,k} \text{ is delivered to the output as } y_{i_{d_2},k}$$

$$...$$

$$z_{q_{\lfloor \frac{n}{12r} \rfloor},k} \text{ is delivered to the output as } y_{i_{d_{\lfloor \frac{n}{12r} \rfloor}},k}$$

for $d_1 < d_2 < \cdots < d_{\lfloor \frac{n}{12r} \rfloor}$ and $\{d_1, d_2, \ldots, d_{\lfloor \frac{n}{12r} \rfloor}\} \subset \{1, \ldots, \frac{n}{4r}\}$. Hence the bits $y_{i_{d_1},k}, \ldots,$

$y_{i_{d_{\lfloor \frac{n}{12r} \rfloor}},k}$ depend on the values of the bits $z_{q_1,k};\ldots;z_{q_{\lfloor \frac{n}{12r} \rfloor},k}$ which means that if the value $z_{q_j,k}$ is changed then the value of $y_{i_{d_j},k}$ is changed too. Let $w_{i,1};w_{i,2};\ldots;w_{i,k-1}$ be the bits of the k bit variable w_i where $w_{i,1}$ is the most significant bit. We indicate the expression $w_{i,1}2^{k-2}+w_{i,2}2^{k-3}+\cdots+w_{i,k-1}2^0$ as H_{w_i}. Now we assign to the bits $z_{i,1};\ldots;z_{i,k-1}$ for $i=1,\ldots,m$ and to the bits $x_{j,1};\ldots;x_{j,k-1}$ for $j=1,\ldots,n$ such values that the following is fulfiled:

$$
\begin{aligned}
H_{z_i} &= 0 \text{ for } i=1,\ldots,q_1\\
H_{x_j} &= 0 \text{ for } j=1,\ldots,i_{d_1}-q_1\\
H_{z_i} &= 1 \text{ for } i=q_1+1,\ldots,q_2\\
H_{x_j} &= 1 \text{ for } j=i_{d_1}-q_1+1,\ldots,i_{d_2}-q_2\\
H_{z_i} &= 2 \text{ for } i=q_2+1,\ldots,q_3\\
H_{x_j} &= 2 \text{ for } j=i_{d_2}-q_2+1,\ldots,i_{d_3}-q_3\\
&\ldots\\
H_{z_i} &= (\left\lfloor \frac{n}{12r} \right\rfloor -1) \text{ for } i=q_{\lfloor \frac{n}{12r} \rfloor -1}+1,\ldots,q_{\lfloor \frac{n}{12r} \rfloor}\\
H_{x_j} &= (\left\lfloor \frac{n}{12r} \right\rfloor -1) \text{ for } j=i_{d_{\lfloor \frac{n}{12r} \rfloor -1}}-q_{\lfloor \frac{n}{12r} \rfloor -1}+1,\ldots,i_{d_{\lfloor \frac{n}{12r} \rfloor}}-q_{\lfloor \frac{n}{12r} \rfloor}\\
H_{z_i} &= \left\lfloor \frac{n}{12r} \right\rfloor \text{ for } i=q_{\lfloor \frac{n}{12r} \rfloor}+1,\ldots,m\\
H_{x_j} &= \left\lfloor \frac{n}{12r} \right\rfloor \text{ for } j=i_{d_{\lfloor \frac{n}{12r} \rfloor}}-q_{\lfloor \frac{n}{12r} \rfloor}+1,\ldots,n
\end{aligned}
$$

and we set $z_{l,k}=0$ for $l\in\{j_1,\ldots,j_{\frac{m}{4}}\}-\{q_1,\ldots,q_{\lfloor \frac{n}{12r} \rfloor}\}$. The vector $(z_{q_1,k};\ldots;z_{q_{\lfloor \frac{n}{12r} \rfloor},k})$ can attain $2^{\lfloor \frac{n}{12r} \rfloor}$ different values. So, various non-τ-inputs cause various τ-outputs and each of τ-inputs is set. This implies at least $2^{\lfloor \frac{n}{12r} \rfloor}$ various states for the circuit and therefore the area is at least $\log 2^{\lfloor \frac{n}{12r} \rfloor}$. Hence: $A(n,m,k,r)=\Omega(\frac{n}{r})$. ∎

REMARK 1. *The theorem 1 holds for $n\geq m\geq\frac{n}{2r}$ too. This means that for $\frac{n}{r}\geq m\geq\frac{n}{2r}$ it is $A(n,m,k,r)=\Omega(m)$. The constant does not play a role at all. Evidently for $\frac{n}{r}\geq m\geq\frac{n}{Kr}$ where $K>1$ is an arbitrary constant $A(n,m,k,r)=\Omega(m)=\Omega(\frac{n}{r})$.*

THEOREM 2. *If $2^{\frac{3}{8}k}\geq\frac{n}{r}\geq m\geq 8$, then $A(n,m,k,r)=\Omega(m((log\frac{n}{rm})+1))$.*

PROOF OF THEOREM 2: W.l.o.g. assume that $\frac{n}{8r}\geq m$. Divide the time of computation into $8r$ time intervals so that in each of them either $\lfloor\frac{n+m}{8r}\rfloor$ or $\lceil\frac{n+m}{8r}\rceil$ least significant bits of variables $y_1,\ldots,y_{n+m}$ are delivered to the output. Among these $8r$ intervals there exists an interval τ such that during τ at most $\frac{(k-1)mr}{8r}=\frac{(k-1)m}{8}$ of the $k-1$ most significant bits of variables $z_1,\ldots,z_m$ are input. This means at least $\frac{7}{8}m(k-1)$ of these bits are non-τ-inputs. There are at least $\frac{m}{2}$ variables $z_{l_1},\ldots,z_{l_{\frac{m}{2}}}$ from $z_1,\ldots,z_m$ such that at least $\lceil\frac{3}{4}(k-1)\rceil=\frac{3}{4}k$ bits of their $k-1$ most significant bits are non-τ-inputs. If there were $\frac{m}{2}+1$ variables whose $\frac{1}{4}k$ of the $k-1$ most significant bits were τ-inputs, then during interval τ there were read $(\frac{m}{2}+1)\frac{1}{4}k>\frac{(k-1)m}{8}$ bits, which contradicts the definition of the interval τ. Let us indicate the least significant bits

which are τ-outputs as: $y_{i_1,k};\ldots;y_{i_{\lfloor\frac{n+m}{8r}\rfloor},k}$ where $i_1<\cdots<i_{\lfloor\frac{n+m}{8r}\rfloor}$. We set $\frac{3}{8}k$ most significant bits of variables $z_1,\ldots,z_m$ in the following way(see Fig. 1.). We assign:

$$H_i=z_{i,1}2^{\frac{3}{8}k-1}+z_{i,2}2^{\frac{3}{8}k-2}+\cdots+z_{i,\frac{3}{8}k}2^0=i-1$$

for $i=1,\ldots,m$. This assignment ensures that: $z_1<\cdots<z_m$. At least $\lceil\frac{3}{4}(k-1)\rceil-\frac{3}{8}k=\frac{3}{8}k$ bits among the bits:$z_{i,\frac{3}{8}k+1};\ldots;z_{i,k-1}$ for $i\in\{l_1,\ldots,l_{\frac{m}{2}}\}$ are such that they are non-τ-inputs and whatever choice their values maintains relation: $z_{l_1}<\cdots<z_{l_{\frac{m}{2}}}$. Further we set the bits $z_{i,j}$ for $i=1,\ldots,m$ and $j=\frac{3}{8}k+1,\ldots,k$ in the following way

$$z_{1,k}=\cdots=z_{m,k}=1$$

$$z_{i,j}=0$$

for $i\in\{1,\ldots,m\}-\{l_1,\ldots,l_{\frac{m}{2}}\}$ and for $j=\frac{3}{8}k+1,\ldots,k-1$. Let us indicate by $z_{i,f_{i_1}},\ldots,z_{i,f_{i_{\frac{3}{8}k}}}$ for $i\in\{l_1,\ldots,l_{\frac{m}{2}}\}$ those bits from the bits $z_{i,\frac{3}{8}k+1};\ldots;z_{i,k-1}$ which are non-τ-inputs. Then we set $z_{i,j}=0$ for $i\in\{l_1,\ldots,l_{\frac{m}{2}}\}$ and for $j\in\{\frac{3}{8}k+1,\ldots,k-1\}-\{f_{i_1},\ldots,f_{i_{\frac{3}{8}k}}\}$. Let us indicate by H_i' the following numbers:

$$H_i'=z_{i,f_{i_1}}2^{\frac{3}{8}k-1}+z_{i,f_{i_2}}2^{\frac{3}{8}k-2}+\cdots+z_{i,f_{i_{\frac{3}{8}k}}}2^0.$$

There exist $2^{\frac{3}{8}k}$ possible values for H_i' depending on the values of the variables $z_{i,f_{i_1}};\ldots;z_{i,f_{i_{\frac{3}{8}k}}}$. Now consider the output Y: $y_1\le y_2\le\cdots\le y_{m+n}$. Let us divide it to m (not necessary equally long) parts: $C_1=\{y_1,\ldots,y_{i_{c_1}}\}$, $C_2=\{y_{i_{c_1}+1},\ldots,y_{i_{c_2}}\}$, $\ldots$, $C_m=\{y_{i_{c_{m-1}}+1},\ldots,y_{i_{c_m}}\}$ where $1<\cdots<i_{c_1}<\cdots<i_{c_2}<\cdots<i_{c_m}$, $i_{c_m}=n+m$ and $\bigcup_{i=1}^m C_i=Y$ so, that each part of them contains at least $\lfloor\frac{n+m}{8rm}\rfloor$ variables from $y_{i_1},\ldots,y_{i_{\lfloor\frac{n+m}{8r}\rfloor}}$ where $i_1<\cdots<i_{\lfloor\frac{n+m}{8r}\rfloor}$ (see Fig. 1.) and so that the variables $y_{i_1},\ldots,y_{i_{\lfloor\frac{n+m}{8rm}\rfloor}}$ belong to the C_1, the variables $y_{i_{\lfloor\frac{n+m}{8rm}\rfloor+1}},\ldots,y_{i_{2\lfloor\frac{n+m}{8rm}\rfloor}}$ belong to the C_2 etc. and the variables $y_{i_{(m-1)(\lfloor\frac{n+m}{8rm}\rfloor)+1}},\ldots,y_{i_{\lfloor m(\frac{n+m}{8rm})\rfloor}}$ belong to the C_m. We proceed by the assignment of values to variables $x_1,\ldots,x_n$ as follows(see Fig. 1.): Let us also divide $x_1,\ldots,x_n$ into m parts so that the h-th part (for $h=1,\ldots,m$) consists of $x_{(\sum_{q=1}^{h-1}(|C_q|-1))+1},\ldots,x_{(\sum_{q=1}^{h}(|C_q|-1))}$ where $|C_q|$ is the number of elements of the set C_q. So the h-th part corresponds to the set C_h in the output Y. This latter set contains at least $\lfloor\frac{n+m}{8rm}\rfloor$ output in τ variables:

$$y_{i_{(h-1)(\lfloor\frac{n+m}{8rm}\rfloor)+1}},\ldots,y_{i_{h\lfloor\frac{n+m}{8rm}\rfloor}}.$$

The assignment to the bits of the h-th part (for $h=1,\ldots,m$) of the variables $x_1,\ldots,x_n$ is the following. The $\frac{3}{8}k$ most significant bits of all variables in the h-th part are assigned the values so that $x_{i,1};\ldots;x_{i,\frac{3}{8}k}$ fulfill the equality:

$$x_{i,1}2^{\frac{3}{8}k-1}+x_{i,2}2^{\frac{3}{8}k-2}+\cdots+x_{i,\frac{3}{8}k}2^0=h-1.$$

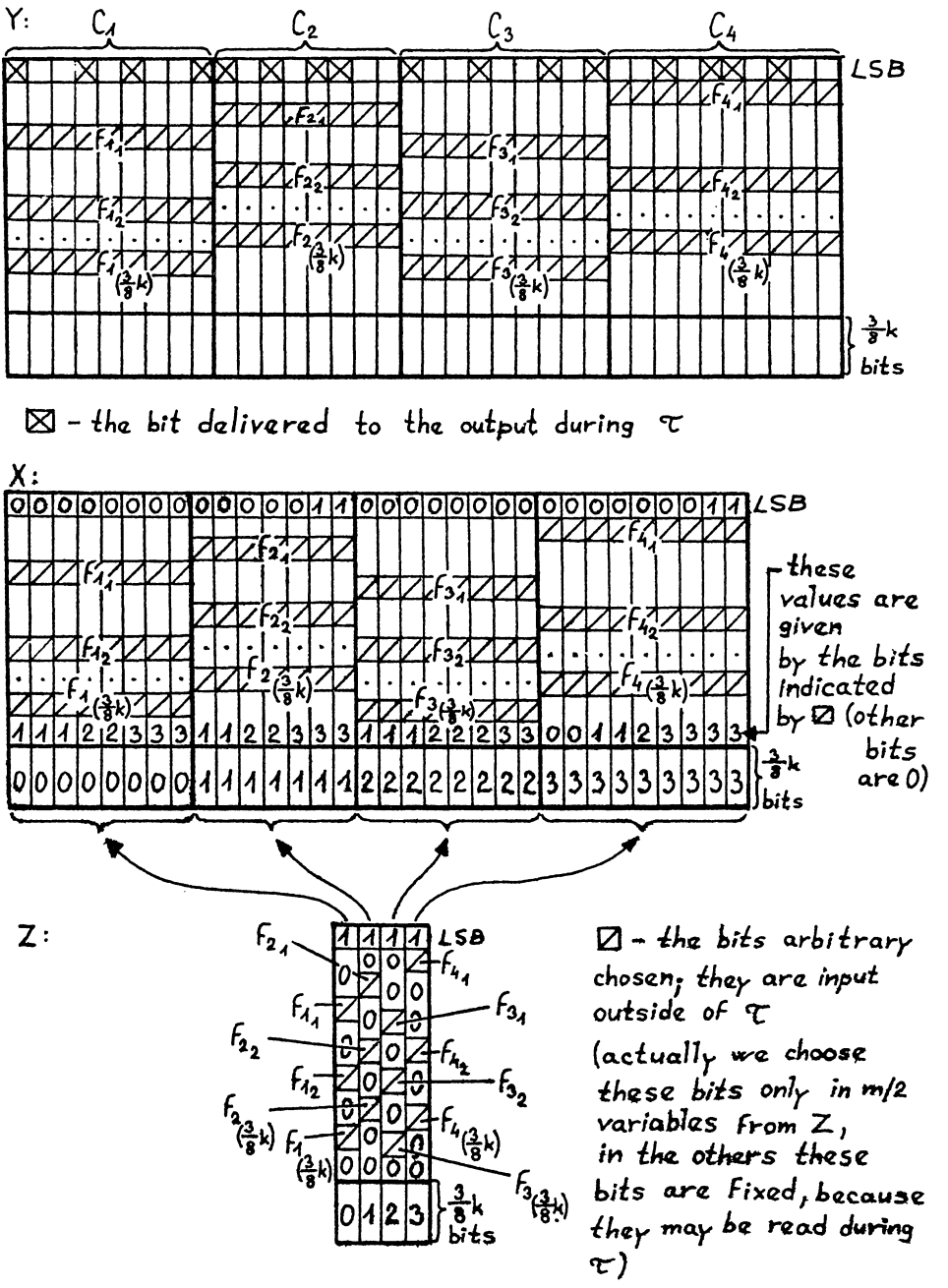


Fig. 1. The assignment to the inputs from X and Z for $m = 4$, $\frac{n}{r} = 16$, $n = 32$.

Let us define:

$$dif_h^j = i_{(h-1)\frac{n+m}{8rm}+j} - \sum_{q=1}^{h-1}(|C_q|),\ dif_h^0 = 1.$$

Let the positions $f_{h_1},\ldots,f_{h_{\frac{3}{8}k}}$ be corresponding to the positions $f_{i_1},\ldots,f_{i_{\frac{3}{8}k}}$ for $i=h$ in the variable $z_h (h=1,\ldots,m)$. Let us assign the values to the $\frac{3}{8}k$ bits on these positions so that:

$$\text{for } j=1,\ldots,\left\lfloor\frac{n+m}{8rm}\right\rfloor \text{ it holds}$$

$$x_{i,f_{h_1}}2^{\frac{3}{8}k-1}+x_{i,f_{h_2}}2^{\frac{3}{8}k-2}+\cdots+x_{i,f_{h_{\frac{3}{8}k}}}2^0 = j-1$$

$$\text{where } i=(\sum_{q=1}^{h-1}(|C_q|-1))+dif_h^{j-1},\ldots,(\sum_{q=1}^{h-1}(|C_q|-1))+dif_h^j-1$$

$$\text{and it holds } x_{i,f_{h_1}}2^{\frac{3}{8}k-1}+x_{i,f_{h_2}}2^{\frac{3}{8}k-2}+\cdots+x_{i,f_{h_{\frac{3}{8}k}}}2^0 = \left\lfloor\frac{n+m}{8rm}\right\rfloor-1$$

$$\text{where } i=(\sum_{q=1}^{h-1}(|C_q|-1))+dif_h^{\frac{n+m}{8rm}},\ldots,\sum_{q=1}^{h}(|C_q|-1).$$

The assignment to the least significant bits of the h-th part is the following

$$x_{j,k}=0$$

$$\text{for } j=(\sum_{q=1}^{h-1}(|C_q|-1))+1,\ldots,(\sum_{q=1}^{h-1}(|C_q|-1))+dif_h^{\lfloor\frac{n+m}{8rm}\rfloor}-1$$

and

$$x_{j,k}=1$$

$$\text{for } j=(\sum_{q=1}^{h-1}(|C_q|-1))+dif_h^{\lfloor\frac{n+m}{8rm}\rfloor},\ldots,\sum_{q=1}^{h}(|C_q|-1).$$

The other bits are set to 0. For $j=1,\ldots,\frac{m}{2}$ we have: To variable z_{l_j} we assign a value $H'_{l_j}\in\{0,1,\ldots,\lfloor\frac{n+m}{8rm}\rfloor-1\}$. Then we get the following outputs. Let $L_j=(l_j-1)(\lfloor\frac{n+m}{8rm}\rfloor)$ and $H'_{l_j}=u$. Then it holds:

$$y_{i_{L_j+1},k}=0,\ldots,y_{i_{L_j+u},k}=0,\ y_{i_{L_j+u+1},k}=1,\ y_{i_{L_j+u+2},k}=0,\ldots,y_{i_{L_j+\lfloor\frac{n+m}{8rm}\rfloor},k}=0$$

There are $\lfloor\frac{n+m}{8rm}\rfloor$ different assignments of a value H'_{l_j} for $j\in\{1,\ldots,\frac{m}{2}\}$. It means there are $\lfloor\frac{n+m}{8rm}\rfloor^{\frac{m}{2}}$ various assignments for $H'_{l_1},H'_{l_2},\ldots,H'_{l_{\frac{m}{2}}}$ in summary and this implies the same number of different outputs. So, the circuit should differentiate among at least $\lfloor\frac{n+m}{8rm}\rfloor^{\frac{m}{2}}$ various states. Therefore $A(n,m,k,r)\geq\frac{m}{2}\log\lfloor\frac{n+m}{8rm}\rfloor$.

According to Remark 1 we can write

$$A(n,m,k,r)=\Omega\left(m\left(\left(\log\frac{n}{rm}\right)+1\right)\right). \quad \blacksquare$$

REMARK 2. *One can prove in a similar way that* $A(n,m,k,r) = \Omega\left(\log \frac{n}{r}\right)$ *also for* $m < 8$.

THEOREM 3. *If* $\frac{n}{r} \geq 2^{\frac{3}{8}k} \geq m$, *then* $A(n,m,k,r) = \Omega(m((log\, 2^{\frac{3}{8}k}_{m}) + 1))$.

PROOF OF THEOREM 3: The proof is analogous to the proof of Theorem 2. The difference is in consideration of only $2^{\frac{3}{8}k}$ (from $\left\lfloor \frac{n+m}{8r} \right\rfloor$ bits) least significant bits which are τ-outputs.

The groups C_i for $i = 1, ..., m$ (see the proof of Theorem 2) of variables $y_1, ..., y_{n+m}$ are such that each of them contains $\frac{2^{\frac{3}{8}k}}{m}$ variables (from the considered ones) that their least significant bits are τ-outputs. Therefore:

$$A \geq \log\left(\frac{2^{\frac{3}{8}k}}{m}\right)^{\frac{m}{2}} \Longrightarrow A = \Omega\left(m \log \frac{2^{\frac{3}{8}k}}{m}\right).$$

Similarly as in Theorem 2, we have:

$$A = \Omega\left(m\left(\left(\log \frac{2^{\frac{3}{8}k}}{m}\right) + 1\right)\right). \quad \blacksquare$$

COROLLARY 1. *If* $\frac{n}{r} \geq 2^{\frac{3}{8}k}$ *and* $\frac{3(8^K)-1}{8^{K+1}-1}k \geq \log m$ *where* $K \geq 0$ *then :*

$$A = \Omega\left(m\left(\left(\log \frac{2^k}{m}\right) + 1\right)\right).$$

PROOF OF COROLLARY 1: If $\frac{3(8^K)-1}{8^{K+1}-1}k \geq \log m$ then there exists $c > 0$ such that $cm \log \frac{2^k}{m} \leq m \log \frac{2^{\frac{3}{8}k}}{m}$. Let us take $c = \frac{1}{8^{K+1}}$. The inequality $\frac{3(8^K)-1}{8^{K+1}-1}k \geq \log m$ holds if and only if

$$\frac{1}{8^{K+1}} m \log \frac{2^k}{m} \leq m \log \frac{2^{\frac{3}{8}k}}{m}. \quad \blacksquare$$

CONCLUSION

Figure 2. contains the survey of the results of the paper and shows the open questions about the multilective merging.

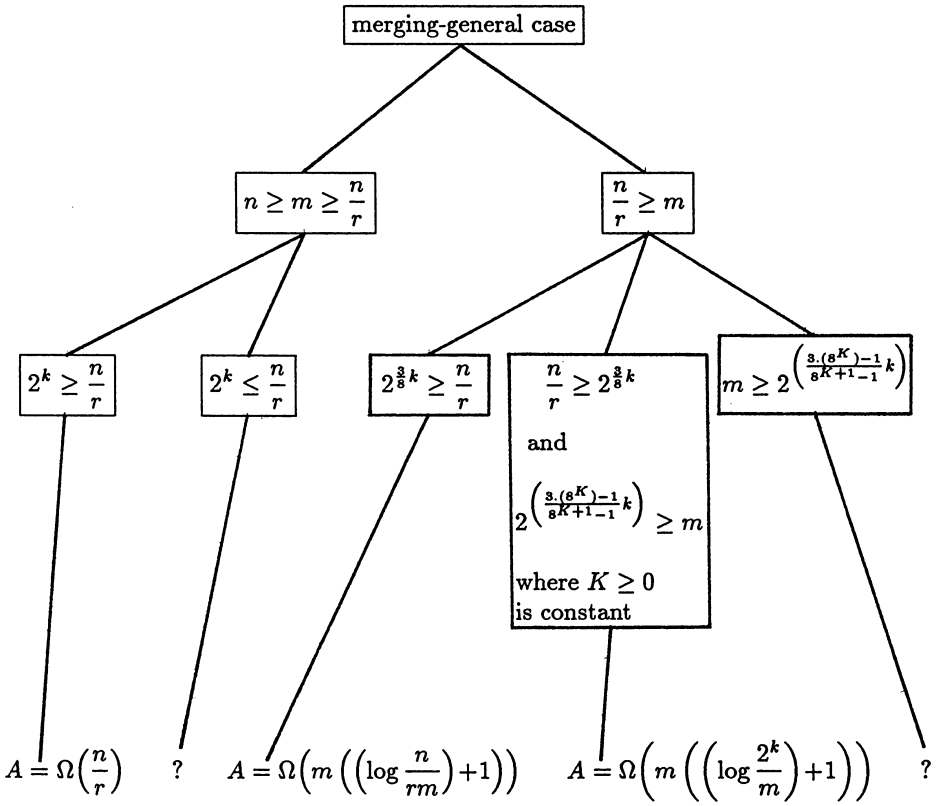


Fig. 2.

Our conjecture is that if $n \geq m \geq \frac{n}{r} \geq 2^k$ then $A = \Omega(2^k((\log\frac{n}{2^k}) + 1))$ and if $\frac{n}{r} \geq m \geq 2^k$ then $A = \Omega(2^k((\log\frac{m}{2^k}) + 1))$.

We hope we could prove all our lower bounds tight by using similar techniques as they are used for semelective merging [**PSV**].

Acknowledgment

The second author thanks Professor Kurt Mehlhorn, Max–Planck–Institut für Informatik, Saarbrücken and Alexander von Humboldt Foundation, Bonn, Germany who partly supported this research.

REFERENCES

[U] Ullman,J.D., *Computational Aspects of VLSI*, Computer Science Press, Rockville, Md. 1983.

[Sa] Savage,J.E., *The Performance of Multilective VLSI Algorithms*, in "Journal of Computer and System Sciences Vol. 29, No. 2, October 1984," Academic Press, New York and London.

[Si] Siegel,A., *Tight Area Bounds and Provably Good AT^2 Bounds for Sorting Circuits.TR*, CS Dept.,New York University, New York 1984.

[G] Gubáš,X., *Close properties of the communication and the area complexity of VLSI circuits (in Slovak), Master thesis*, Comenius University, Bratislava 1988.

[PSV] Palko,V., Sýkora,O., Vrťo,I., *Area complexity of merging*, In: MFCS' 89, Springer Verlag 1989, 390–396.

Deriving Fully Efficient Systolic Arrays by Quasi-Linear Allocation Functions*

Xiaoxiong Zhong and Sanjay Rajopadhye
[xzhong, sanjay]@cs.uoregon.edu
Computer Science Department
University of Oregon
Eugene, Oregon 97403-1202

Abstract

We address the problem of deriving systolic arrays in which the processor utilization is 100%. We first address this problem in the context of *synthesis* from Uniform Recurrence Equations (UREs), and then generalize our result to deal with *arbitrary* systolic arrays (outside the context of synthesis). We show that in a systolic array, it is always possible to merge a parallelepiped of neighboring processors which are active at different clock cycles. The new array is fully efficient and its processors have almost the same cost as the original one. Such merging corresponds exactly to the transformation by a **quasi-linear** function. When the original array is derived by integral linear projections of systems of UREs, we give a method to mechanically determine the quasi-linear allocation function which yields the efficient array. The technique can also be extended to any (piece-wise) systolic array to derive a fully efficient array by "post-processing" it with a (piece-wise) quasi-linear function.

1 Introduction

In the past few years, a relatively mature synthesis technique for systolic arrays has emerged [Rao85, Qui87, Mol83]. The specification is a system of Uniform Recurrence Equations (UREs) [KMW67] (or Affine Recurrence Equations, AREs*). The standard technique is to transform these UREs into a systolic algorithm which has a direct correspondence to a systolic array implementation. Typically, one uses an integral linear transformation [CS84, Mol83, Qui87], and can be described as follows: **specify** the algorithm as a system of UREs; determine a **linear schedule** (represented by its *norm*

*Supported by NSF grant MIP-8802454

*It has been shown [RF90] that even if one starts from AREs, it is essential to localize the data dependencies and convert the ARE into a URE. Hence we assume that the specification has been processed in this manner.

λ); determine an **allocation function** (represented by an integral projection vector u) which does not conflict with the timing function (i.e. satisfies the constraint $\lambda u \neq 0$).

To evaluate the derived arrays, there are three common criteria—the computation time, the number of processors and the efficiency. The efficiency of the array derived from the above technique is $\eta = 1/\delta$ where $\delta = |\lambda^t u|$. The problem of finding optimal timing functions, under some standard assumptions, can be reduced to a linear programming problem [LW85, Qui87, Mol83]. Also, the problem of finding linear allocation functions which can yield minimum number of processors has been studied [WD89, ZWR90]. The third criterion, namely the **efficiency** of a systolic array, however, has not received much attention.

In this paper, we address the problem of deriving efficient systolic arrays. We first show that for any systolic array which is derived by the conventional linear transformations, it is always possible to **merge** every δ neighboring processors which are active at different time units to derive a new array. These δ processors form a parallelepiped which can be constructed by a new basis of the processor space. The **efficiency** of the new array is 100%. It has the same computation time as the original one and has only $1/\delta$ number of processors. Furthermore, the cost of the processor in the new array remains the same, except for a few additional registers and wires (no additional functional units are required). This method is equivalent to using allocation functions that are not integral but rational matrices, and then obtaining integral processor labels by using the floor function. Such functions are called *quasi-affine* functions by Quinton [Qui87].

Based on the standard view of the systolic array, we also show that the above mathematical technique can be applied to **any** systolic array to derive a fully efficient systolic array. This can also be easily generalized to transform piece-wise systolic arrays [Thi89] to fully efficient ones, by applying a *piece-wise* quasi-affine function.

The rest of this paper is arranged as follows. After a preliminary discussion of the standard synthesis methods and its constraints we prove a standard result about the efficiency of synthesized systolic arrays (Sec. 3). In Sec. 4, we will show that one can always "postprocess" the array derived by the conventional linear transformation to yield a new array which has less processors and is fully efficient. We will also show this corresponds exactly to an array derived by a quasi-linear allocation function applied to the original UREs. We extend these results to arbitrary systolic arrays in Sec 5, and conclude with a discussion and comparison.

2 Preliminaries

A systolic array consists of identical processors which are connected locally. Processors process data from input channels and send out the output through output channels to other processors at every clock cycle. To synchronize the data flow, it is possible that a processor has to be idle during some clock cycles. This leads to the concept of **extrinsic**

iteration interval [Rao85] which is defined as follows:

Definition 1 The **extrinsic iteration interval** of a systolic array is defined to be δ if every processor is active at exactly one out of every δ consecutive clock cycles. The **efficiency** η of the array defined as $\eta = 1/\delta$.

For example, in the Kung-Leiserson systolic array for banded matrix multiplication [KL80] (see Figure 1), the processors are active once every 3 clock cycles. Therefore, $\delta = 3$ and $\eta = 1/3$.

Definition 2 A $m \times n$ matrix U ($m \leq n$) is said to be **e-unimodular** (for extended unimodular) if the gcd of the determinants of all its $m \times m$ submatrices is 1.

It is well known [Sch88] (pp. 47, Cor 4.1c), that a system of diophantine equations $Ux = I$ has an integral solution for *any* integral vector I, iff U is e-unimodular. One of the useful properties of e-unimodular matrices is that the column Hermite form of an e-unimodular $m \times n$ matrix, A, is* $[E_m\ 0]$. We say that a vector is a normalized vector if it is e-unimodular (i.e. the gcd of all its components is 1).

Throughout this paper, we will assume that the UREs are defined on an n-dimension integral domain $\mathcal{D}$. The standard linear transformation technique to synthesize systolic arrays from UREs involves in the two linear transformations, namely, the timing function (represented by its norm λ) and the allocation function (represented by a normalized projection vector u or an integral $(n-1) \times n$ matrix A (satisfying $Au = 0$, and u is the basis of the right null space of A). The following constraints must be satisfied.

- **Causality of the timing function:** If, evaluating $f(I+d)$ needs value $f(I)$, then $\lambda^t d > 0$.
- **Non-conflict:** The allocation function does not conflict with the timing function, i.e. no two points are mapped to the same processor at the same time. This is equivalent to $\lambda^t u \neq 0$.
- **Dense array** The derived array must be dense i.e. every integral point in the processor space must be the image of an integral point in the index space of the problem. This is equivalent to the constraint that A should be e-unimodular.

Besides the above constraints, we can make the following assumptions without any loss of generality.

- λ is a normalized vector. This is because if λ is not a normalized vector, it can be written as $s\lambda'$ for some positive integer, s and normalized vector, λ'. It is easy to verify that λ' still represents a valid timing function (see [RF90]) and it yields a faster schedule than λ.

*E_i denotes the $i \times i$ identity matrix.

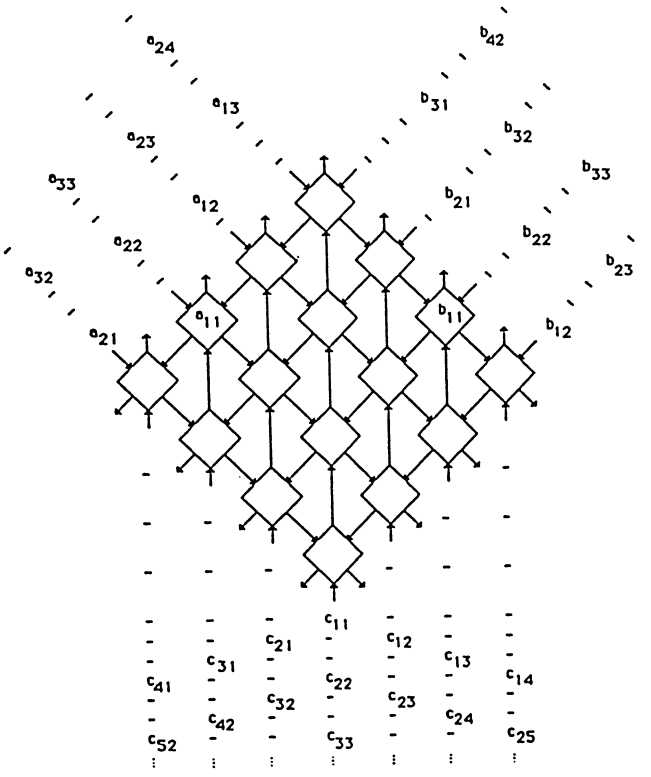


Figure 1: Kung-Leiserson Band Matrix Multiplication

- The dependency graph of the computation is connected. (Otherwise, the computation consists of more than one totally independent computation, and we can rewrite them as separate UREs). This constraint is satisfied iff the dependency matrix $(D_1 \ \ldots \ D_k)$ where $D_1, \ldots, D_k$ are all dependency vectors is e-unimodular (see [ZWR90]).

Geometrically, the timing and allocation functions can be unified as a single transformation from the computation domain $\mathcal{D}$ to the processor-time domain $\mathcal{T}$. This can be described by an $n \times n$ matrix $T = \begin{pmatrix} A \\ \lambda^t \end{pmatrix}$. We denote the processor space (i.e. the first $n-1$ dimensions of $\mathcal{T}$) as $\mathcal{P}$. Furthermore, we say a processor-time point $(P\,t)^t \in \mathcal{T}$ is *active* if it is the image of a computation point, i.e. there exists an $I \in \mathcal{D}$ such that $TI = (P\,t)^t$. Otherwise, $(P\,t)^t$ is said *inactive* (or "hole"). Intuitively, a processor-time point $(P\,t)^t$ is active iff the processor in the location P is active at time t.

3 Efficiency of Synthesized Arrays

We will now prove that the extrinsic iteration interval of the systolic array is $|\lambda^t u|$. Although this is well known, it is necessary to prove this in our formal setting. Furthermore, our proof establishes certain lemmata that are useful later.

The active points in a derived array can be characterized by certain properties of T. Let $C = (C_1 C_2 \ldots C_n)$ be an $n \times n$ unimodular matrix such that $AC = (E_{n-1}\ 0)$ (since A is e-unimodular, its column Hermite form is $(E_{n-1}\ 0)$). Let $v^t = \lambda^t (C_1 \ldots C_{n-1})$ and $k = \lambda^t C_n$ (thus, $\lambda^t C = (v^t\ k)$). Note that since $AC = (E_{n-1}\ 0)$, we have $AC_n = 0$, i.e. C_n is a right null vector of A. Moreover, since C is unimodular, C_n must be normalized. Therefore, $C_n = \pm u$ and $k = \lambda^t C_n = \pm \lambda^t u$. The following lemma gives an important characterization of active points.

Lemma 1 A processor-time point $(P\,t)^t \in \mathcal{T}$ is active iff the following equation has an integral solution j.

$$v^t P + jk = t \tag{1}$$

Proof: Consider transformation matrix T. We have

$$T = \begin{pmatrix} A \\ \lambda^t \end{pmatrix} = \begin{pmatrix} (E_{n-1} 0) C^{-1} \\ \lambda^t \end{pmatrix} = \begin{pmatrix} E_{n-1}\ 0 \\ \lambda^t C \end{pmatrix} C^{-1}$$

So

$$T = \begin{pmatrix} E_{n-1} & 0 \\ v^t & k \end{pmatrix} C^{-1} \tag{2}$$

Now, a processor-time point $(P\,t)^t \in \mathcal{T}$ is active iff there exists $I \in \mathcal{Z}^n$ such that

$$\begin{pmatrix} P \\ t \end{pmatrix} = TI = \begin{pmatrix} E_{n-1} & 0 \\ v^t & k \end{pmatrix} C^{-1} I$$

Let $J = C^{-1}I$. This defines a bijection $\mathcal{Z}^n \rightarrow \mathcal{Z}^n$ because C (and hence, C^{-1}) is unimodular. Hence the above system of equations has an integral solution I iff the following system has an integral solution J.

$$\begin{pmatrix} E_{n-1} & 0 \\ v^t & k \end{pmatrix} J = \begin{pmatrix} P \\ t \end{pmatrix}$$

Letting $J = (j_1 \ \ldots \ j_{n-1}\ j)^t$, we see that the solution for $(j_1 \ \ldots \ j_{n-1})^t$ is simply P. Substituting this into the last equation and simplifying, yields Eqn. 1. ∎

Theorem 1 The extrinsic iteration interval δ, of the systolic array derived from a URE is $|\lambda^t u|$.

Proof: We first prove that if an processor-time point $P_{t_0} = (P\,t_0)^t$ is active, then for any scalar integer α, the processor-time point $(P\,t_0 + \alpha k)^t$ is also active. Since $(P\,t_0)^t$ is active, there exists an integer j_0 such that

$$v^t P + j_0 k = t_0$$

Adding αk to both sides,

$$v^t P + (j_0 + \alpha)k = t_0 + \alpha k$$

Hence, processor P is active at $t_0 + \alpha k$, and the extrinsic iteration interval, δ, must be a factor of k, i.e. $\delta | k$. To show that $|k|$ is exactly δ, we now prove that if processor P is active at t_1 and t_2, then $(t_1 - t_2)$ must be a multiple of k. Indeed, based on Lemma 1, there exist integers j_1 and j_2 such that

$$v^t P + j_1 k = t_1$$

and

$$v^t P + j_2 k = t_2$$

Hence,

$$(j_1 - j_2)k = t_1 - t_2$$

and therefore, the extrinsic iteration interval, δ, is $|k| = |\lambda^t u|$. ∎

Corollary 1 From Eqn 2, we further have $\det T = \pm\delta$.

The "holes" in the processor-time space occur when T is not bijective. This is true only when T is not a unimodular matrix. The above corollary conforms to this. Theoretically, it is always possible to select an integral vector u such that $\lambda^t u = \pm 1$. This is due to the following lemma.

Lemma 2 For any normalized n-dimensional vector w^t, it is always possible to choose an $(n-1) \times n$ integral matrix M such that the matrix $(w\ M^t)^t$ is unimodular.

Proof: Since $w^t = (w_1 \ldots w_n)$ is a normalized vector, there always exists an integral vector u such that $w^t u = \gcd(w_1 \ldots w_n) = 1$ and correspondingly we can choose an e-unimodular matrix M such that $Mu = 0$. It is easy to see that Eqn. 2 holds for M too. Therefore, $|M| = \pm w^t u = \pm$, and hence M is unimodular. ∎

In practice, however, there may be other factors which prohibit the choice of such allocation functions. The first factor is when the computation domain $\mathcal{D}$ is infinite. In this situation, there is only one projection vector (the extremal ray of $\mathcal{D}$) which yields a finite array, as shown by Quinton [Qui87] and this projection vector may not yield fully efficient array (i.e. $\delta = 1$). Consider banded matrix multiplication (for possibly infinitely large matrices), the computation domain is infinite and the only valid array derived by the above transformation is Kung-Leiserson array whose efficiency is only 1/3 (Figure 1).

The second factor is related to the minimization of the number of the processors. The projection vectors which yield more efficient arrays may not yield the arrays with fewer processors. For example, if we consider banded (but bounded) matrix multiplication, one of the projection vector yields a fully efficient array with n^2 processors while the Kung-Leiserson array uses only $w_1 w_2$ (w_1, w_2 are the bandwidths of the matrices) processors. Even though this array is not fully efficient, it may still be preferable to the other array.

Therefore, it is inherently restrictive to derive a fully efficient array by merely using integral linear transformations. An obvious extension to this is to permit the allocation function to be *rational*. But to guarantee that the transformation yields a regular array, we have to change the locations (labels) of the processors denoted by rationals to integers. One of the possibilities is to use the floor function. Such functions are called *quasi-linear* functions by Quinton [Qui87] and are used as timing functions in the same paper.

4 Synthesizing Fully Efficient Systolic Arrays

Our main idea in "postprocessing" the systolic array obtained by the integral linear transformation is to try to merge as many neighboring processors as possible. In order not to increase the functional units in the resultant processor, only those neighboring processors which are active at different clock cycles will be merged. Furthermore, while the shape of the clusters to be merged may be arbitrary, we are particularly interested in merging processors that form parallelepipeds in the processor space.

We therefore need $(n-1)$ directions $\nu_1, \ldots, \nu_{n-1}$ along which the parallelepipeds are formed. It is clear that we are only interested in those parallelepipeds of volume δ (i.e. parallelepipeds with δ integral points in them). To be more precise, we factorize δ such that $\delta = \delta_1 \ldots \delta_{n-1}$ and these factors together with the basis vectors define a family of parallelepipeds.

Definition 3 A *snapshot* at time t_0 is the space $\{(P\, t_0)^t \mid P \in \mathcal{P}\}$.

Definition 4 A factorization $\delta = \delta_1 \delta_2 \ldots \delta_{n-1}$ of δ, a matrix N of $(n-1)$ basis vectors $(N = (\nu_1, \ldots, \nu_{n-1})$, for the processor space $\mathcal{P}$, and a processor-time point $(P\, t)^t$ define a *parallelepiped*, denoted by $\Pr(N, P, t)$ of volume δ as follows.

$$\Pr(N, P, t) = \{(P + N(l_1 \ldots l_{n-1})^t \; t)^t \mid 0 \leq l_i < \delta_i, i = 1, .., n-1\}$$

It should be noted that the requirement that $\nu_1, \nu_2, \ldots, \nu_{n-1}$ is a basis for the processor space $\mathcal{P}$ is necessary. Otherwise, there may be some integral points in the parallelepiped $\Pr(N, P, t)$ but they can not be written as $l_1\nu_1 + \ldots + l_{n-1}\nu_{n-1}$ for some $0 \leq l_i < \delta_i$ (i.e. the number of integral points in the parallelepiped may be greater than δ). Our main result is that such clustering into parallelepipeds is always possible, provided that the factors of δ are mutually co-prime.

Let $k_i = \frac{\delta}{\delta_i}$. We will require our basis vectors to satisfy an additional constraint that* $v^t \nu_i = g c_i k_i$ where $c_1, \ldots, c_{n-1}$ are integers satisfying $\gcd(c_i, \delta) = 1$ for $i = 1, \ldots, (n-1)$ and $\gcd(c_1, \ldots, c_{n-1}) = 1$. This constraint will be used later. Thus each ν_i must be a solution of the i-th equation in the following system of diophantine equations.

$$\begin{aligned} v^t x_1 &= g c_1 k_1 \\ v^t x_2 &= g c_2 k_2 \\ &\vdots \\ v^t x_{n-1} &= g c_{n-1} k_{n-1} \end{aligned}$$

As shown by Banerjee ([Ban88], Th. 5.4.2, p.81), the general solution to the i-th equation is given by

$$x_i = U \begin{pmatrix} c_i k_i \\ t^i_1 \\ \vdots \\ t^i_{n-2} \end{pmatrix}$$

where t^i_j for $j = 1, \ldots, n-2$ are arbitrary integers and U is a unimodular matrix satisfying $Uv = (g, 0, \ldots, 0)^t$. Such a U can always be found, and thus, our ν_i's satisfy the desired

*Recall that $\lambda^t C = (v^t\ k)$; we let g be the gcd of all components of v.

constraint iff the matrix

$$V = \begin{pmatrix} c_1k_1 & c_2k_2 & \dots & c_{n-1}k_{n-1} \\ t_1^1 & t_1^2 & \dots & t_1^{n-1} \\ \vdots & \vdots & \vdots & \vdots \\ t_{n-2}^1 & t_{n-2}^2 & \dots & t_{n-2}^{n-1} \end{pmatrix}$$

is unimodular because $(\nu_1 \dots \nu_{n-1}) = UV$. The following lemma gives us necessary and sufficient conditions for this.

Lemma 3 V is unimodular iff $\gcd(\delta_i, \delta_j) = 1$ for all $i \neq j$.

Proof:

If part. Let, $\gcd(\delta_i, \delta_j) = 1$, for all $i \neq j$. First, we prove that $(k_1 \dots k_{n-1})$ is a normalized vector. Suppose p is a common prime divisor for $k_1 \dots k_{n-1}$. Obviously, $p|k_1$. There exists $i \neq 1$ such that $p|\delta_i$, since $k_i = \delta_1 \dots \delta_{i-1}\delta_{i+1} \dots \delta_{n-1}$. Similarly, since $p|k_i$, there exists $j \neq i$ such that $p|\delta_j$. Because $\gcd(\delta_i, \delta_j) = 1$, p must be 1. Therefore, $(k_1, \dots, k_{n-1})$ is normalized. We now prove that $(c_1k_1 \dots c_{n-1}k_{k-1})$ is also normalized. Again, suppose p is a common prime divisor for all c_ik_i, then p either divides c_i or k_i. Since $\gcd(c_1, \dots, c_{n-1}) = 1$ and $\gcd(k_1, \dots, k_{n-1}) = 1$, there exist i, j $(i \neq j)$ such that $p|c_i, p|k_j$. But from $\gcd(c_i, \delta) = 1$, we know $\gcd(c_i, k_j) = 1$. Hence, $p = 1$. By Lemma 2, we know it is always possible to choose t_i^j's such that V is unimodular.

Only If Part. Without loss of generality, suppose $\gcd(\delta_1, \delta_2) = d \neq 1$. $d|k_1$ because $\delta_2|k_1$. Also, $d|k_i$ for $i \neq 1$ because $\delta_1|k_i$ for all $i \neq 1$. Therefore, we know $d|c_ik_i$ for all $i = 1, \dots, n-1$. Hence, V is not unimodular. ∎

We now prove that any snapshot of any parallelepiped formed by the basis vectors as chosen above contains exactly one active point.

Lemma 4 For any time instant t, there is exactly one active point in any parallelepiped $\Pr(N, P, t)$.

Proof: First, we prove that there is at most one active point in $\Pr(N, P, t)$. Suppose there are two active points, say, $(P_1\ t)^t$ and $(P_2\ t)^t$ in this parallelepiped. It is easy to see that the point $(P_1\ t)^t - (P_2\ t)^t$ is also active. This means that there are integers $l_1, l_2, \dots, l_{n-1}$ where $0 \leq l_i < \delta_i$ for $i = 1, \dots, n-1$ such that $\begin{pmatrix} l_1\nu_1 + \dots + l_{n-1}\nu_{n-1} \\ 0 \end{pmatrix}$ is active. From Lemma 1, we know that there is an integral solution to the Eqn. 1, i.e. there is an integer J_n such that

$$l_1v^t\nu_1 + \dots + l_{n-1}v^t\nu_{n-1} + J_nk = 0$$

This can be further simplified to

$$g(l_1c_1k_1 + \dots + l_{n-1}c_{n-1}k_{n-1}) + J_nk = 0$$

Notice that for any $i \neq j$, $\delta_i | k_j$. Therefore, dividing both sides of the above equation by δ_i, we have $\delta_i | g l_i c_i k_i$. Because $\gcd(g, \delta) = 1$, hence $\gcd(g, \delta_i) = 1$. Also, $\gcd(c_i, \delta) = 1$, and hence $\gcd(c_i, \delta_i) = 1$. Furthermore, $\gcd(\delta_i, \delta_j) = 1$ (for $i \neq j$), and hence $\gcd(\delta_i, k_i) = 1$. So $\delta_i | l_i$. Because $0 \leq l_i < \delta_i$, l_i must be 0, i.e. P_t^1 is the same as P_t^2.

To prove that there is at least one active point in the parallelepiped, again, consider Eqn. 1 which can be simplified to

$$g(l_1 c_1 k_1 + \ldots + l_{n-1} c_{n-1} k_{n-1}) + J_n k = t - v^t P \tag{3}$$

We want to prove that there are integers $l_1, \ldots, l_{n-1}$ for $0 \leq l_i < \delta_i$ and some integer J_n as the solution to the above equation. To prove this, we first prove $\gcd(g c_1 k_1, \ldots, g c_{n-1} k_{n-1}, k) = 1$. Consider any prime common divisor p of these integers, we prove $p = 1$. Because $\gcd(g, k) = 1$, if $p | g$, then we already prove that $p = 1$ (because $p | k$ too). If p is not a factor of g, we have $p | c_i k_i$ for every $i = 1, \ldots, n-1$. Because $\gcd(c_1 k_1, \ldots, c_{n-1} k_{n-1}) = 1$, p must be 1.

Therefore, there exist integers $l'_1, \ldots, l'_{n-1}, J'_n$ such that

$$l'_1 g c_1 k_1 + \ldots + l'_{n-1} g c_{n-1} k_{n-1} + J'_n k = 1$$

Denoting $t - v^t P$ as t' and multiplying both sides of the equation by t', we have

$$t' l'_1 g c_1 k_1 + \ldots + t' l'_{n-1} g c_{n-1} k_{n-1} + J'_n t' k = t'$$

For $t' l'_i$, we can always find two integers q_i and l_i where $0 \leq l_i, \delta_i$ such that $t' l'_i = q_i \delta_i + l_i$ (i.e. l_i is the remainder of dividing $t' l_i$ by δ_i). Notice that $\delta_i k_i = \delta = \pm k$. The left hand side of the above equation becomes

$$(\sum_{i=1}^{n-1} l_i g c_i k_i) + J'_n t' k + q_1 g c_1 \delta_1 k_1 + \ldots + q_{n-1} g c_{n-1} \delta_{n-1} k_{n-1}$$

$$= (\sum_{i=1}^{n-1} l_i g c_i k_i) + (J'_n t' \pm (q_1 g c_1 + \ldots + q_{n-1} g c_{n-1}))k$$

Let $J_n = (J'_n t' \pm (q_1 g c_1 + \ldots + q_{n-1} g c_{n-1}))$, l_i and J_n together satisfy Eqn. 3. ∎

Therefore, We can merge all the δ processors in such parallelepipeds into one processor. The new processor does not need to have extra processing function units (but may need some additional links and registers) and it will be active in the whole computation. This leads to the following theorem.

Theorem 2 It is always possible to merge δ neighboring processors which form a parallelepiped in the processor space to derive a fully efficient array. The new array has the same computation time as the original one and has the same cost except for some additional links and registers.

It should be clear that there is much freedom to choose the basis vectors $\nu_1, \ldots, \nu_{n-1}$ to form the parallelepiped. In practice, the selection of the basis vectors should try to minimize the number of the extra links, and the number of specialized boundary processors required.

4.1 Quasi-Linear Allocation Functions

To place the resultant array onto the integral grid, we need an explicit allocation function which assigns computations to processors denoted by integral points. To find such a function for the resultant array, we observe that the procedure we described so far can be viewed as the composition of three parts: the original linear allocation function (represented by an $(n-1) \times n$ e-unimodular matrix A); a basis transformation function which transforms the derived processor space $\mathcal{P}$ into another space with $\nu_1, \ldots, \nu_{n-1}$ as the bases; and clustering of processors into rectangular parallelepipeds. This clustering is obtained by dividing the processor coordinate along the basis ν_i by δ_i and taking the floor function $\lfloor\ \rfloor$. Note that we have assumed that processor 0 is always in $\mathcal{P}$. If this were not so, we can simply add a constant to the allocation function derived here and we get a quasi-affine rather than a quasi-linear function.

Because the matrix $V = (\nu_1 \ \ldots \ \nu_{n-1})$ is the basis transformation matrix from the new basis vectors to the old basis vectors, V^{-1} is the basis transformation from the old basis vectors to the new basis vectors. Therefore, we define the allocation function as follows:

For any $I \in \mathcal{D}$, $Q(I) = \lfloor DV^{-1}A(I) \rfloor$ where* D is an $(n-1) \times (n-1)$ diagonal matrix which is defined as $D = diag(1/\delta_1, \ldots, 1/\delta_{n-1})$. The following theorem states that the allocation function Q yields a dense array.

Theorem 3 The allocation function $Q = \lfloor DV^{-1}A \rfloor$ yields a dense array.

Proof: We need to show that Q is a surjective function from $\mathcal{Z}^n$ to $\mathcal{Z}^{n-1}$, i.e. for any $P = (p_1\, p_2\, \ldots\, p_{n-1})^t \in \mathcal{Z}^{n-1}$, there is at least an $I \in \mathcal{Z}^n$ such that $P = QI$. Because $DP_1 = P$ where $P_1 = (\delta_1 p_1\ \ \delta_2 p_2\ \ \ldots\ \ \delta_{n-1} p_{n-1})^t$ (actually, there are δ possible P_1s which are mapped to P under D, but for the purpose of our proof, we just choose one of them). Furthermore, there is $P_2 = VP_1 \in \mathcal{Z}^{n-1}$ (because V^{-1} is unimodular) such that $DV^{-1}P_2 = P$. Now, since A is e-unimodular and therefore, there always exists integral solution for the equation $P_2 = AI$ (pp. 47, Cor 4.1c, [Sch88]), therefore, Q is surjective. ∎

So far, we have shown that a particular class of quasi-linear functions (those that can be decomposed into the three factors above) can be used as allocation functions to

*$\lfloor M \rfloor$ for a matrix M is defined as taking floor for every entry in M

obtain efficient arrays. We hypothesize that one could use general quasi-linear functions as allocation functions. For this to be valid, we need to show that arrays obtained by such functions are still regular. An informal argument supporting this is as follows. First, we note that any $(n-1) \times n$ rational matrix Q can be written as $\frac{1}{d}P$ where d is an integer and P is an $(n-1) \times n$ integral matrix. Hence, if I depends on $I+D$ in the computation, then processor $\lfloor QI \rfloor$ should have a link from processor $\lfloor QI+QD \rfloor = \lfloor \frac{1}{d}(PI) + \frac{1}{d}(PD) \rfloor$. Let $QI = \lfloor QI \rfloor + f(I)$ where $f(I)$ is the fractional part of QI. We conjecture that $f(I)$ is periodic and bounded by d, and therefore, $\lfloor QI+QD \rfloor = \lfloor QI \rfloor + \lfloor f(I)+QD \rfloor$. Hence, processor QI has at most d different links (corresponding to dependency D) and this is true for any I. Therefore, the resultant array is regular. Furthermore, we also expect that one can always factorize any valid quasi-linear allocation function into the three parts as above.

5 Optimal Clustering of Arbitrary Systolic Arrays

So far, we have addressed the problem of deriving efficient systolic arrays in the context of synthesis. There are, however, many systolic arrays which are not derived from UREs by the conventional linear transformation. To transform such arrays into efficient ones, we study how to apply our theory to an arbitrary systolic array. First, let us recall the standard points of view of a systolic array.

By Rao and Kailath [RK86], a systolic array implements an RIA . This RIA is defined in a processor-time space. More precisely, suppose the processor space is defined in $\mathcal{P} \subset \mathcal{Z}^{n-1}$, then the RIA is defined in $\mathcal{Z}^n$ as follows:

If there is a link with a delay D_t ($D_t \geq 1$) from processor p_1 to processor $p_1 + D$ in the processor space, then for any time t, processor-time point $(p_1 + D\,t + D^t)^t$ depends on $(p_1\,t)^t$. Hence, there is a uniform dependency $\begin{pmatrix} D \\ D_t \end{pmatrix}$.

Generally, if $D_1, \ldots, D_k$ are all the link vectors in the processor space (we assume $k \geq n$, otherwise, the RIA can be transformed into a lower dimension space) and $D_1^t, \ldots, D_k^t$ are the time delayed along the i-th link respectively, then the RIA implemented by the array is defined by the matrix D formed by the dependency vectors as $\begin{pmatrix} D_1 & \ldots & D_k \\ D_1^t & \ldots & D_k^t \end{pmatrix}$.

It might seem straightforward to use the above technique because the array is derived by a projection of the RIA along the time axis. But if the array is not 100% efficient, then the computation dag of this RIA consists of k **disconnected** components in $\mathcal{Z}^n$, which violates the assumption we made in Section 2. Hence our previous results cannot be applied directly.

Let us examine processor-time points in $\mathcal{Z}^n$. Some of these points correspond to useful computations (i.e. the processors are active) and others do not. Moreover, if the extrinsic iteration interval is δ, then along the time axis, there is an active processor-time

point every δ points.

Basically, if we assume that the origin point (i.e. $0 \in \mathcal{Z}^n$) is active, then any active point will be connected to the origin in the dag of the RIA. Thus, a processor-time point $(Pt)^t$ is active iff it can be represented by a linear combination of the dependency vectors of the dag. This leads to the following proposition.

Remark 1 A processor-time point $(Pt)^t$ is active iff there exists a k-dimensional integral vector J such that $DJ = (P\,t)^t$.

Let $D = \begin{pmatrix} D_p \\ D_t \end{pmatrix}$ where $D_p = (D_1 \ldots D_k)$ and $D_t = (D_1^t \ldots D_k^t)$. D_p is the $n-1 \times k$ connection matrix for the processor space. To guarantee that the array is dense (i.e. every integral point in $\mathcal{P}$ is a valid processor), D_p must be e-unimodular (Lemma 4, in [ZWR90]). Moreover, it is reasonable to assume that D_t is a normalized vector because, otherwise, we can get a faster array by just simply replacing D_t with D_t' where $D_t = cD_t'$ and D_t' is normalized. The matrix D is thus analogous to the transformation matrix T that we have studied so far, except that it is not square.

Because D_p is e-unimodular, there exists a $k \times k$ unimodular matrix $U = (U_1 \ldots U_k)$ such that $D_p U = (E_{n-1}\, 0)$. Define $w^t = D_t(U_1 \ldots U_{n-1})$ and $l_i = D_t U_i$ for $i = n, \ldots, k$, thus $D_t U = (w^t\, l_n \ldots l_k)$. We therefore have the following analogue of Lemma 1 (the proof is also analogous, and omitted for brevity).

Lemma 5 A processor-time point $(Pt)^t \in \mathcal{T}$ is an active point iff the following equation has an integral solution $J_n, \ldots, J_k$.

$$w^t P + J_n l_n + \ldots + J_k l_k = t \tag{4}$$

The key difference between Eqn. 1 and Eqn. 4 is that in Eqn. 4, there are $l_n, \ldots, l_k$ instead of k in Eqn. 1. The following Lemma enables us to eliminate this difference too.

Lemma 6 Eqn. 4 has integral solution $J_n, \ldots, J_k$ iff the following equation has an integral solution J

$$w^t P + Jl = t \tag{5}$$

where $l = \gcd(l_n, \ldots, l_k)$

Proof: Let $(l_n, \ldots, l_k) = l(l_n', \ldots, l_k')$ and $\gcd(l_n', \ldots, l_k') = 1$. It is easy to see that if Eqn. 4 has integral solution $J_n, \ldots, J_k$, then $J = l_n' J_n + \ldots + l_k' J_k$ is an integral solution to Eqn. 5.

Conversely, suppose Eqn. 5 has an integral solution J. Because $\gcd(l_n', \ldots, l_k') = 1$, there are integers $m_n, \ldots, m_k$ such that $m_n l_n' + \ldots + m_k l_k' = 1$. Therefore, $J m_n l_n' + \ldots + J m_k l_k' = J$ and Eqn. 5 becomes

$$w^t P + l(J m_n l_n' + \ldots + J m_k l_k') = t \tag{6}$$

Therefore, $J_n = Jm_n, \ldots, J_k = Jm_k$ is an integral solution to Eqn. 4. ∎

We then have the following analogue (proof omitted for brevity) of Th. 1

Theorem 4 The extrinsic iteration interval δ of any systolic array is l.

We can use the technique of Th. 2 to merge δ neighboring processors within a parallelepiped and derive for **any** systolic array a 100% efficient array.

Matrix Multiplication Example

For the Kung-Leiserson matrix multiplication array, $\delta = 3$, which is a prime number. All possible parallelepipeds which we can merge have a 1×3 aspect ratio. Furthermore, we have the following:

$$D = \begin{pmatrix} D_p \\ D_t \end{pmatrix} = \begin{pmatrix} 0 & 1 & -1 \\ 1 & 0 & -1 \\ 1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 1 & 3 \end{pmatrix} \begin{pmatrix} 0 & 1 & -1 \\ 1 & 0 & -1 \\ 0 & 0 & 1 \end{pmatrix} \tag{7}$$

Hence, we have $w^t = (1\ 1)$, $g = 1$ and the following two equations.

$$\begin{aligned} x + y &= c_1 \\ x + y &= 3c_2 \end{aligned}$$

where $\gcd(c_1, c_2) = 1$ and $\gcd(c_1, 3) = \gcd(c_2, 3) = 1$.

There are many ways to pick the basis vectors. For example, if we choose $c_1 = c_2 = 1$, we have $\nu_1 = (1\ 0)^t$ and $\nu_2 = (4\ -1)$ as the solutions to the above two equations respectively. Also is $\nu_1 = (0\,1)$ and $\nu_2 = (-1\,4)$. That is, we can cluster the 3 neighboring processors vertically or horizontally. Further, if we choose $c_1 = 2$ and $c_2 = 1$, we get $\nu_1 = (1\ 1)$ and $\nu_2 = (1\ 2)$ as a solution. ν_1 stands for the main diagonal. It should be noted that $(1\ -1)$ is not a solution to the first equation, because if it were, c_1 must be 0, hence $\gcd(c_1, 3) = 3$ which does not satisfy the condition as stated above. In fact, in the array, all processors along the direction $(1\ -1)$ are active or inactive at the same time.

6 Discussion and Conclusions

Recently, there has been some work in the context of merging processors of systolic arrays derived by conventional linear transformation to yield a fully efficient systolic array as reported by Bu and Deprettere [BDD90] and also by Clauss et al. [CMP90]. Both of them adopt the same approach, namely, selecting $\delta - 1$ vectors $\lambda_1, \ldots, \lambda_{\delta-1}$ in the problem domain and merging processors $p_0, p_0 + A\lambda_1, \ldots, p_0 + A\lambda_{\delta-1}$ (where A is the allocation

matrix). We call this kind of approaches "enumeration approach". Our work differs from these approaches in two aspects.

First, due to the selection of vectors $\lambda_1, \ldots, \lambda_{\delta-1}$, it is very possible that $p_0, p_0 + A\lambda_1, \ldots, p_0 + A\lambda_{\delta-1}$ don't form a parallelepiped and may form a cluster of arbitrary shape. Although the resultant array is still regular (because the original array is regular), it is difficult (if not impossible) to come up with an **explicit** allocation function for the final array. Moreover it is not clear how the dense array constraint can be satisfied. In contrast, our approach is constructive and guarantees that an explicit allocation function satisfying the dense array constraint can be found.

Second, the enumeration approaches can only be applied to systolic arrays derived by the conventional linear transformation. In contrast, by studying the activation patterns of the RIA that is implemented by an arbitrary systolic array, our approach can be applied to any (pure) systolic array. Furthermore, the method can be extended to piece-wise systolic arrays as follows. A piece-wise array consists of a constant number of pieces of pure systolic arrays. We can thus apply our technique to each piece of the array and adjust the connections between boundary processors accordingly. This corresponds to using piece-wise quasi-linear functions, and is especially useful when dealing with ingenuous arrays designed by ad-hoc manner, outside a standard synthesis methodology. Such arrays are most likely to be piece-wise systolic, and our approach can be used to improve the efficiency of such arrays.

The results presented here generalize a previous result reported elsewhere [ZR91]. In that paper we show that there always exist some directions along which one can always merge δ neighboring processors to obtain a fully efficient array for any systolic array derived by the standard linear transformation. It is thus an instance of a parallelepiped, where the factorization of δ is $\delta = 11 \ldots 1\delta$ (i.e. all except one of the factors are one, and the last one is δ itself).

Our work reported here raises an interesting question regarding the cost (and hence optimality) of systolic arrays derived by linear transformations. Traditionally, the two cost measures that have been used are the computation time and the number of processors. However, by using the results in this paper, one can always reduce the processor count by a factor of δ. Thus, the "raw" processor count by itself is not an accurate measure. This corresponds to the volume v of a convex polyhedron (the domain of computation, $\mathcal{D}$), under the transformation to space-time, T. Except for two-dimensional recurrences, this is not a linear function and hence the optimal solution can be obtained only be enumeration. It would be interesting to investigate how such methods [WD89] can be adapted to use the new cost function which is $v/|T|$.

References

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic

Publishers, 1988.

[BDD90] J. Bu, E. F. Deprettere, and P. Dewilde. A design methodology for fixed-size systolic arrays. In S. Y. Kung and E. Swartzlander, editors, *International Conference on Application Specific Array Processing*, pages 591–602, Princeton, New Jersey, Sept 1990. IEEE Computer Society.

[CMP90] P. Clauss, C. Mongenet, and G. R. Perrin. Calculus of space-optimal mappings of systolic algorithms on processor arrays. In S. Y. Kung and E. Swartzlander, editors, *International Conference on Application Specific Array Processing*, pages 4–18, Princeton, New Jersey, Sept 1990. IEEE Computer Society.

[CS84] Peter R. Cappello and Kenneth Steiglitz. Unifying VLSI designs with linear transformations of space-time. *Advances in Computing Research*, 2:23–65, 1984.

[KL80] H. T. Kung and C. E. Leiserson. *Algorithms for VLSI Processor Arrays*, chapter 8.3, 'Introduction to VLSI Systems,' Mead, C. and Conway, L., pages 271–292. Addison-Wesley, Reading, Ma, 1980.

[KMW67] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *JACM*, 14(3):563–590, July 1967.

[LW85] G. J. Li and B. W. Wah. Design of optimal systolic arrays. *IEEE Transactions on Computers*, C-35(1):66–77, 1985.

[Mol83] D. I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proceedings of the IEEE*, 71(1):113–120, January 1983.

[Qui87] Patrice Quinton. *The Systematic Design of Systolic Arrays*, chapter 9, Automata Networks in Computer Science, pages 229–260. Princeton University Press, 1987. Preliminary versions appear as IRISA Tech Reports 193 and 216, 1983.

[Rao85] Sailesh Rao. *Regular Iterative Algorithms and their Implementations on Processor Arrays.* PhD thesis, Stanford University, Information Systems Lab., Stanford, Ca, October 1985.

[RF90] Sanjay V. Rajopadhye and Richard M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14:163–189, June 1990.

[RK86] Sailesh Rao and Thomas Kailath. What is a systolic algorithm. In *Proceedings, Highly Parallel Signal Processing Architectures*, pages 34–48, Los Angeles, Ca, Jan 1986. SPIE.

[Sch88] A. Schrijver. *Theory of Integer and Linear Programming.* John Wiley and Sons, 1988.

[Thi89] Lothar Thiele. On the design of picecwise regular processor arrays. In *International Symposium on Circuits and Systems*, pages 2239–2542. IEEE CAS, IEEE Press, 1989.

[WD89] Jiwan Wong and Jean-Marc Delosme. Optimization of the processor count for systolic arrays. Technical Report YALEU-DCS-RR-697, Computer Science Dept. Yale University, May 1989.

[ZR91] Xiaoxiong Zhong and Sanjay V. Rajopadhye. Synthesizing efficient systolic arrays. In *International Conference on Acoustics, Speech and Signal Processing*, pages accepted, Toronto, Canada, May 1991. IEEE.

[ZWR90] Xiaoxiong Zhong, Ivan M. Wong, and Sanjay V. Rajopadhye. Bounds on the solution space in systolic array design. Technical Report CIS-TR-90-10, University of Oregon, Computer Science Department, 120 Deschutes Hall, Eugene OR 97403-1202, April 1990. (submitted to Journal of VLSI Signal Processing; preliminary results reported in VLSI Signal Processing IV, San Diego CA, Nov 1990).

Affine Timings for Systems of Affine Recurrence Equations †

C. MONGENET
Université Louis Pasteur, Département d'Informatique
7 rue René Descartes, 67084 STRASBOURG, FRANCE

Abstract. — This paper is devoted to the problem of the existence of affine timings for problems defined by Systems of Affine Recurrence Equations. After a first analysis, such problems may have no affine timing not because the problem is uncomputable but only because of the initial system of equations. This system can induce dependencies organized in an inappropiate way. We give conditions for a dependency to be well-organized in such a way that an affine timing may exist. When a dependency does not satisfy these conditions, we describe how to transform it in order to meet the conditions. A problem defined by a system of equations is analyzed by a step-by-step examination of its dependencies. For each dependency organized in an inappropriate way, a transformation is applied. The whole transformation process yields to the determination of a new equivalent system of equations from which an affine timing can usually be computed. Many practical problems need such transformations. We illustrate this transformation process on the Algebraic Path Problem.

Keywords. — systems of affine recurrence equations, affine timing functions, mapping, systolic arrays, processor arrays.

Introduction

Many research efforts have focused on methods to map problems on systolic arrays ([KUN 82]) and on processor arrays architectures ([MOL 83], [QUI 84], [DEI 85], [MON 85], [MOF 87], [FFW 87], [MOP 87], [YAC 88]).

The first approaches mainly dealt with systems of Uniform Recurrence Equations introduced by KARP, MILLER and WINOGRAD ([KMW 67]). However this restrictive class of equations cannot naturally express lots of problems. So, researchers now focus on a larger class of equations : the Systems of Affine Recurrence Equations (SARE in the following) ([CLP 88], [QVD 89]).

† This work has been supported by the Laboratoire d'Informatique de Besançon and by the French Coordinated Research Program C^3 of the CNRS.

The goal of these different methods is the same. Using the information given by the dependency analysis of the problem, they determine first a *timing* or *scheduling* of the elementary calculation points of this problem, then an *allocation* of these calculations on a set of processors. The timing is generally defined by an *affine timing function.*

This paper is concerned with the problem of the affine timing determination.

A problem defined by a given initial SARE can have no parallel solutions because the dependencies between the calculations are such that no possible affine scheduling of these calculations exists. This can be intrinsically related to the problem and the interdependencies between the calculations. But experience proves that in some cases, the lack of affine timing is not due to strong interdependencies but only to one ill-conditioned dependency. That is to say, the initial SARE is not appropriate. In such cases, a transformation of this initial SARE into an equivalent one can then yield to the existence of affine schedulings. This could be achieved by applying formal methods of recurrence equations transformations.

Instead of this formal approach, our objective is to include the problem of SARE transformation in our method for the mapping of SARE onto regular arrays ([MCP 91]). We shall use its general geometrical framework to discuss the existence of affine timings and the necessity of dependency transformations when no such timings exist. We propose to analyze individually each dependency in order to determine if it is ill-conditioned. When it is, i.e. when no affine timing exists according to this dependency, we present domain transformations in order to get a timing. Since each dependency of the problem will be examined separately, the existence of a timing for the whole problem will not be known and guarantied before the complete examination of all the dependencies. In practice, this approach is very fruitful for lots of problems (Gauss-Jordan, Algebraic Path Problem, etc) where the initial SAREs contain ill-conditioned dependencies. After application of the process described here, a well-conditioned description of the problem, i.e. an equivalent SARE, is obtained from which we easily deduce an affine timing.

We recall the different steps of our mapping method. We first define the notion of affine timing. We emphasize the interest of constant timing, i.e. timing independent on the problem size. The dependency analysis is then realized in terms of two classes of vectors : the *generating vectors* and the *inductive vectors.* Using this information, we first present the condition for an affine timing to be valid according to one dependency. We differentiate two validity conditions or constraints : the weak condition and the stronger one where broadcast is removed. When one of these validity conditions is not satisfied we describe how to transform the domain and the current dependency in order to get a valid timing according to this dependency.

This paper is organized as follows. Section 1 presents the concept of SARE. The notion of affine timing and the way we realize the dependency analysis are described in section 2. Section 3 focuses on the conditions for the existence of a valid affine timing according to one dependency and presents the appropriate domain transformation to be applied in case of non-existence of a timing. Finally section 4 illustrates these results on a classical problem, the Algebraic Path Problem ([ROT 87], [DEL 88], [BE-all 89]). We show why the initial problem specification does not yield to any affine timing. The transformations

described in this paper are then applied on this problem and the resulting specification can be scheduled.

1. Problems definition

We consider problems that can be expressed in terms of a system of affine recurrence equations. Each equation of the system is of the form :

$$X_u[z] = f_u\Big(\ldots, X_v[\rho_{u,v,p}(z)], \ldots\Big) \qquad z \in D_{u,p} \tag{E}$$

where

- X_u and X_v are variable names. X_u is a result variable (possibly an intermediary one) whose items $X_u[z]$, $z \in D_{u,p}$, are calculated in terms of argument variables X_v appearing in the right part of (E). X_v is either a data or a result variable (possibly X_u). The variables X_u and X_v are indexed with integral indexes z and $z' = \rho_{u,v,p}(z)$ of constant dimension. If X_v is a result variable it must be fully indexed, i.e. $\rho_{u,v,p}(z)$ and z must be of the same dimension n.

- $D_{u,p} \subset \mathbb{Z}^n$ is a convex bounded polyhedron or a union of convex bounded polyhedra associated with equation (E). It is called the *index domain*. It defines the set of integral coordinates points on which the equation (E) is defined. Each point z of $D_{u,p}$ corresponds to the calculation of one item $X_u[z]$. The domains of all the equations (E) of the system are of the same dimension. If several equations (E) define the same result variable X_u, their domains must be disjoint to avoid a result to be defined twice. In practice we have $n = 2$ or 3 to get by projection of the domain a linear or bidimensional array. We call *domain* of the problem the union of the $D_{u,p}$.

- $p \in \mathbb{Z}^q$ is the *size parameter*. It characterizes the bounds of the index domain $D_{u,p}$.

- f_u is any function. We assume its complexity is O(1).

- the ... express that they are other arguments of the same form as $X_v[\rho_{u,v,p}(z)]$.

- $\rho_{u,v,p}$ is an affine function from $\mathbb{Z}^n \to \mathbb{Z}^{n'}$ $(n' \leq n)$. It is called the *index mapping*. It defines the item of variable X_v used in equation (E) to calculate the item $X_u[z]$. It is of the form :

$$\rho_{u,v,p}(z) = A(z) + B(p) + C$$

The initialization of the recurrence defined by equation (E) has to be specified. This is done by the *initialization* or *input equations*. There is at least one such equation associated with each result X_u of the problem. It is of the form :

$$X_u[z_o] = d \qquad z_o \in D_{u_o,p} \subset \mathbb{Z}^n \tag{Init}$$

where

- z_o corresponds to the initial step of the recurrence. In practice the initialization domain $D_{u_o,p}$ is adjacent to the calculation domain $D_{u,p}$.

- d is a data of the problem. It is either a constant or a data variable over a set of integral indexes.

From such a problem definition we analyze the dependencies in order to characterize conditions for the existence of affine timings associated with each dependency of the problem. The dependency analysis is realized in terms of 2 classes of vectors : the *generating vectors* denoted Φ and the *inductive vectors* denoted Ψ.

2. Affine timing and dependency analysis

The dependency analysis information is used to determine the possible timing functions and at a later stage (not discussed in this paper) the allocation of the calculations to the processors.

In this paper we only consider affine timing functions. The timing function defines, for any point z of the domain, its execution instant $t(z)$ in the following way :

DEFINITION 1. — *An* affine timing function *associated with a problem is a function of the form* :

$$\begin{array}{rlll} t: & D \subset \mathbf{Z}^n & \rightarrow & \mathbf{N} \\ & z & \rightarrow & t(z) = \lambda \cdot z + \alpha \end{array}$$

where $\lambda = (\lambda_1, \ldots, \lambda_n)$ *and* $\lambda_1, \ldots, \lambda_n, \alpha$ *are integral constants.*

The vector λ is orthogonal to hyperplanes of $\mathbf{Z}^n$ such that all the points belonging to one of them have the same execution instant. We call these hyperplanes the *timing surfaces.*

We now introduce the notion of efficient timing. Let z_1 and z_2 be any two points of the domain. An efficient timing function should guaranty that the difference between their execution instants $t(z_1) - t(z_2)$ is independent on the parameter size p of the problem. Hence if this difference depends on p , the delay between the execution instants of z_1 and z_2 would increase when p increases. That is why the only interesting timing functions are the constant timings defined by :

DEFINITION 2. — *An affine timing is* constant *if and only if it is independent on the size parameter p of the problem.*

From now one we shall only consider constant affine timing functions.

Because of the dependencies between elementary calculations of a problem, a timing must satisfy some conditions. We call them the *causal constraints.* They express that an elementary calculation point z_1 which uses as data a result item calculated by a point z_0 must be executed *after* z_0. Therefore these constraints restrict the timing and are only associated with the result variables whose items are used as data.

The problem also contains dependencies which are associated with data variables. These dependencies do not impose any causal constraints : any scheduling of all the calculations using a same data is valid.

To take into account the different incidence on the timing of the result and data variables, we analyze their dependencies differently.

2.1. Dependencies for a data

A data variable X_v does not impose any intrinsic causal constraint. Since a data variable does not need to be fully indexed we use $\zeta_0 \in \mathbf{Z}^{n'}$ as its index. The only information we need to know about X_v is on which subset of the domain D any given data item $X_v[\zeta_0]$ is used. This set of points is called a *utilization set* and is defined in the following way :

DEFINITION 3. — *The* utilization set *associated with a given data item* $X_v[\zeta_0]$ *that occurs in equation* (E) *defining the variable* X_u *is*

$$\mathrm{Util}_E(X_v, \zeta_0) = \{z \in D_{u,p} \mid \rho_{u,v,p}(z) = \zeta_0\}$$

Since the index mappings $\rho_{u,v,p}$ are affine functions, the solutions of any $\rho_{u,v,p}(z) = \zeta_0$ form an affine subspace of $\mathbf{Z}^n$. $\mathrm{Util}_E(X_v, \zeta_0)$ is then the intersection of this affine subspace with the domain $D_{u,p}$ which is a convex polyhedron or a union of convex bounded polyhedra. Therefore we have the following property :

PROPERTY 1. — *Every utilization set* $\mathrm{Util}_E(X_v, \zeta_0)$ *is a convex polyhedron or a union of convex bounded polyhedra.*

We introduce the notion of *generating vectors* to characterize these utilization sets.

DEFINITION 4. — *We call* generating vectors *associated with a data variable* X_v *that occurs in equation* (E) *the* m *vectors of a basis of* $\mathrm{Util}_E(X_v, \zeta_0)$ *for any* ζ_0. *We denote them* $\Phi_{E,X_v,i} (i = 1 \ldots m)$.

Note that if for a given equation (E), the utilization set $\mathrm{Util}_E(X_v, \zeta_0)$ is of dimension 0, it is reduced to one point. This means that a given data item $X_v(\zeta_0)$ is used only on one calculation point z. Such a situation is characterized by a null generating vector $\Phi_{E,X_v} = 0$.

2.2. Dependencies for a result

We consider now the case where the variable X_v appearing in the recurrence equation (E) is a result. Recall that since X_v is a result variable, it is fully indexed. We use $z_0 \in \mathbf{Z}^n$ as an index of X_v.

The recurrence equation $X_u[z] = f_u(\ldots, X_v[\rho_{u,v,p}(z)], \ldots)$ expresses that the computation of variable X_u at point z causally depends on the computation of variable X_v at point $\rho_{u,v,p}(z)$, i.e. the computation of X_u at point z must occur *after* the one of X_v at point $\rho_{u,v,p}(z)$. For a given result item $X_v[z_0]$ with $z_0 = \rho_{u,v,p}(z)$ we need to characterize the set of all the points z depending causally on point z_0, i.e. using the result provided by z_0. This set is called a *reception set* and defined by :

DEFINITION 5. — *The* reception set *associated with a given result item* $X_v[z_0]$ *that occurs in equation* (E) *defining the variable* X_u *is*

$$\mathrm{Rec}_E(X_v, z_0) = \{z \in D_{u,p} \mid \rho_{u,v,p}(z) = z_0\}$$

Similarly to property 1 we have :

PROPERTY 2. — *Every reception set* $\mathrm{Rec}_E(X_v, z_0)$ *is a convex bounded polyhedron or a union of convex bounded polyhedra.*

These reception sets are defined in the same way as the utilization sets. However they have different conceptual meanings. A utilization set is associated with a data while a reception set is associated with a result. There is an intrinsic causal dependency underlying the notion of reception set which does not exist in the notion of utilization set. We will represent this causal dependency by *inductive vectors* defined in the following way :

DEFINITION 6. — *We call* inductive vector *associated with a result variable* X_v *that occurs in equation* (E), *any vector denoted by* $\Psi_{X_v,\rho_{u,v,p}(z)}$ *and defined by* :

$$\Psi_{X_v,\rho_{u,v,p}(z)} = z - \rho_{u,v,p}(z)$$

A reception set is then characterized by its basis of generating vectors (in the same way as a utilization set) and by the inductive vectors. These inductive vectors connect a point $z_0 = \rho_{u,v,p}(z)$ computing a result item $X_v[z_0]$ to all the points z, $z \in \mathrm{Rec}_E(X_v, z_0)$, that causally depend on z_0, i.e. using the result provided by z_0 for their own computation. Notice that there are as many sets of inductive vectors associated with a given result X_v as there are occurrences of X_v in the right side of any equation (E).

All the points $z_0 = \rho_{u,v,p}(z)$ which are origin of a vector of a given set of inductive vectors form a set. This set is called the *emission set* and defined by :

DEFINITION 7. — *The* emission set *associated with a given result variable* X_v *that occurs in equation* (E) *defining the variable* X_u *is*

$$\mathrm{Emit}_E(X_v) = \{z_0 \in D \mid \exists z \in D_{u,p} \text{ such that } \rho_{u,v,p}(z) = z_0\} = \rho_{u,v,p}(D_{u,p})$$

Since $\mathrm{Emit}_E(X_v)$ is the image by the affine function $\rho_{u,v,p}$ of a convex bounded polyhedron or of the union of convex bounded polyhedra, we have the following property :

PROPERTY 3. — *Every emission set* $\mathrm{Emit}_E(X_v)$ *is a convex bounded polyhedron or a union of convex bounded polyhedra.*

3. Validity conditions for the existence of affine timings

We now use the data dependencies information to determine the validity conditions on the timing regarding to one dependency associated with a result variable.

We distinguish two validity conditions :
• the weak validity condition. It characterizes timings which are valid according to one dependency, i.e. timings that only satisfy the causal constraints.
• the stronger validity condition. It characterizes timings which are not only valid but which also remove the broadcast.

3.1. Weak validity condition and existence of constant affine timings

PROPERTY 4. Weak validity condition. — *An affine timing is* valid according to a given dependency associated with a result variable *if and only if it satisfies*

$$\lambda \cdot \Psi > 0$$

for all the inductive vectors Ψ *characterizing this dependency.*

DEMONSTRATION : A causal constraint expresses that the point z depending causally on a point z_0 must be executed after z, i.e. $t(z) > t(z_0)$. This is equivalent to $\lambda.(z-z_0) > 0$. By definition 6, $z-z_0 = \Psi$. The validity condition is therefore $\lambda.\Psi > 0$. □

This validity condition can be satisfied or not depending on the respective organization of the reception and emission sets. It expresses the existence of affine timings valid according to one dependency. Let us now discuss this problem in the context of the only efficient timings : the constant ones.

Let (S) be the affine subspace of equation $\rho_{u,v,p}(z) = z_0$. By definition of a reception set, $\mathrm{Rec}_E(X_v, z_0)$ is included in (S). Two cases can occur for the corresponding emission point z_0 : either z_0 belongs to (S) or it does not. Let us study this two cases regarding to the existence of constant affine timings.

3.1.1. $z_0 \in (S)$

In this case all the corresponding inductive vectors Ψ_{X_v,z_0} are colinear and belong to the subspace (S). These vectors associated to one point z_0 can be organized in 2 different ways :

• either they all belong to the same open half-subspace. Therefore we have the following property :

PROPERTY 5. — *When an emission point* z_0 *belongs to the subspace* (S) *of equation* $\rho_{u,v,p}(z) = z_0$ *and has all its inductive vectors* Ψ_{X_v,z_0} *in the same open half-subspace, these vectors define a* cone *whose origin is* z_0. *It is denoted* $C(\Psi_{X_v,z_0})$.

The existence of a timing for this dependency will depend on the way all the cones (for the different emission points z_0) are organized. This situation is a particular case of the problem discussed in section 3.1.2.

• or they belong to the whole subspace. In this case the validity condition $\lambda \cdot \Psi > 0$ can not be satisfied since there exist opposite inductive vectors. This situation does not allow any affine timing as stated by the following property.

PROPERTY 6. — *When an emission point z_0 belongs to the subspace (S) of equation $\rho_{u,v,p}(z) = z_0$ and is such that its inductive vectors Ψ_{X_v,z_0} describe the whole subspace, there is no affine timing.*

This situation is characterized by the fact the reception set is divided into 2 disjoints subsets. By translating one of them as it is presented on figure 1, we transform the problem into the first situation where all the vectors Ψ_{X_v,z_0} belong to the same open half-subspace. The existence of a timing will eventually be possible. Such a situation is illustrated on the example of the Algebraic Path Problem presented in section 4.

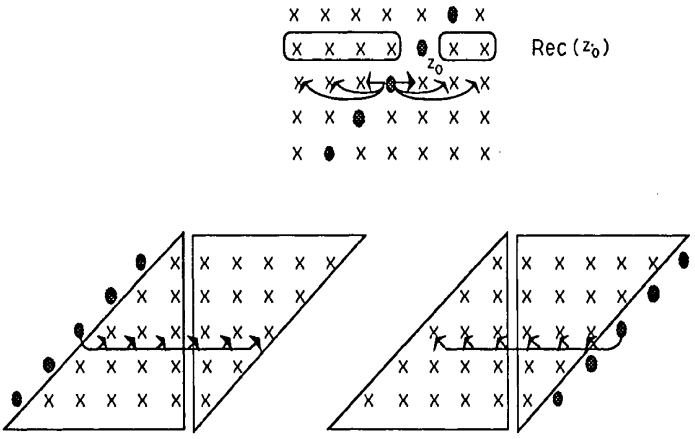


Figure 1 : transformation on a reception set when no affine timing exists

3.1.2. $z_0 \notin (S)$

In this case all the inductive vectors Ψ_{X_v,z_0} have the same origin z_0 and their extremities belong to the union of convex bounded polyhedra $\mathrm{Rec}_E(X_v, z_0)$. We therefore deduce the following property :

PROPERTY 7. — *When an emission point z_0 does not belong to the subspace of equation $\rho_{u,v,p}(z) = z_0$ the set of inductive vectors Ψ_{X_v,z_0} associated with z_0 define a* cone *whose origin is z_0. It is denoted $C(\Psi_{X_v,z_0})$.*

We now consider the union of all these cones.

DEFINITION 8. — *We call* extremal cone *C_p associated with a dependency on the result variable X_v the convex hull of the union of all the Ψ-cones $C(\Psi_{X_v,z_0})$, $z_0 \in \mathrm{Emit}_E(X_v)$ where p is the parameter size.*

PROPERTY 8. — *Consider one dependency on a result variable of a problem characterized by an extremal cone $\mathcal{C}_p$. A constant affine timing valid according to this dependency exists if and only if the extremal cone $\mathcal{C}_p$ is strictly included in a half-subspace of $\mathbb{R}^n$.*

DEMONSTRATION : If $\mathcal{C}_p$ is strictly included in a half-subspace of $\mathbb{R}^n$, then its complementary cone defined by $\{\mu \mid \mu \cdot \Psi < 0, \forall \Psi \in \mathcal{C}_p\}$ is not empty. Since it contains the vectors μ such that $\mu \cdot \Psi < 0$, any vector $\lambda = -\mu$ satisfies $\lambda \cdot \Psi > 0, \forall \Psi \in \mathcal{C}_p$. Therefore these vectors λ define affine timings. Among them, there exist constant ones. Vice versa, if $\mathcal{C}_p$ is not strictly included in a half-subspace of $\mathbb{R}^n$ (i.e. its either the half-subspace or the whole subspace) its complementary cone is empty and no affine timing exists. □

When no constant affine timing exists, domain transformations are necessary. Translations will not solve the problem. We need to apply transformations such as foldings. This point is not studied in this paper. We now examine the following question : When a constant affine timing does exist, under which conditions does a constant broadcast-removing (abbreviated BR in the following) affine timing exist ?

3.2. Stronger validity condition and existence of constant BR affine timings

On many parallel architectures, the broadcast of data is not easily or efficiently implementable. We may wish to remove such a broadcast. This can be realized by restricting the timings to timings which do not allow broadcast. We call such timings the Broascast Removing timings (BR timings). They are easily characterized by the following property for problems whose reception sets are of dimension 1, i.e. described by only one generating vector Φ. In the general case, when the reception sets are of dimension greater than 1, a transformation is first necessary to partly remove the broadcast. This point is discussed in [MCP 90].

PROPERTY 9. Stronger validity condition. — *A BR affine timing associated with a given dependency characterized by a set of extremal vectors Ψ_{X_v, ex_i} and one generating vector Φ must satisfy*

$$\lambda \cdot \Psi_{ex_i} > 0 \quad \forall i \in \mathbb{N}$$
$$\lambda \cdot \Phi \neq 0$$

DEMONSTRATION : Since the extremal cone $\mathcal{C}_p$ is characterized by a finite set of extremal vectors Ψ_{ex_i} any inductive vector Ψ is equal to $\sum_i \alpha_i \Psi_{ex_i}$ with $\alpha_i \geq 0$. Therefore the condition $\lambda \cdot \Psi > 0, \quad \forall \Psi$ is equivalent to $\lambda \cdot \Psi_{ex_i} > 0, \quad \forall \Psi_{ex_i}$. Moreover the generating vector Φ directs lines whose points use the same item of X_v. To avoid a broadcast, these points must belong to different timing surfaces, i.e. $\lambda \cdot \Phi \neq 0$. □

PROPERTY 10. — *Consider one dependency on a result variable of a problem characterized by an extremal cone $\mathcal{C}_p$ strictly included in a half-subspace of $\mathbb{R}^n$. There exists a constant broadcast-removing affine timing valid according to this dependency if and only if $\lim_{p \to \infty} \mathcal{C}_p$ is strictly included in a half-subspace of $\mathbb{R}^n$.*

DEMONSTRATION : Let us consider the complementary cone $\overline{C_p}$ of C_p defined by $\overline{C_p} = \{\mu \mid \mu \cdot \Psi < 0, \forall \Psi \in C_p\}$. If $\lim_{p\to\infty} C_p$ is not strictly included in a half-subspace of $\mathbf{R}^n$, i.e. if it is the whole half-subspace, $\lim_{p\to\infty} \overline{C_p}$ is reduced to a half-subspace. Let us show that this half-subspace only contains timings creating broadcast.

Since the extremal cone C_p is strictly included in a half-subspace of $\mathbf{R}^n$, by property 8 there exist constant affine timings. By definition , any reception set is included in a subspace of $\mathbf{Z}^n$. Therefore the set of all vectors ν orthogonal to a reception set define a subspace of $\mathbf{Z}^n$. These vectors are characterized by $\nu \cdot \Phi = 0$ for any generating vector Φ of the reception set. Moreover half of this subspace of vectors ν is characterized by $\nu \cdot \Psi > 0$. Therefore these vectors ν characterize affine timings. Since they satisfy $\nu \cdot \Phi = 0$, these timings create broadcast. Hence we have the following : whenever affine timings exist, the half-subspace of broadcast-creating timings defines solutions. Therefore if $\lim_{p\to\infty} \overline{C_p}$ is reduced to a half-subspace, this subspace is necessarily the set of the broadcast-creating timings. In this case there is no broadcast-removing timings.

Vice-versa, if $\lim_{p\to\infty} C_p$ is strictly included in a half-subspace, $\lim_{p\to\infty} \overline{C_p}$ is not reduced to the half-subspace of broadcast-creating timings. Therefore in this case, there exist constant broadcast-removing timings in the cone $\lim_{p\to\infty} \overline{C_p}$. □

REMARK : When the extremal cone C_p is strictly included in a half-subspace of $\mathbf{R}^n$, it is characterized by a finite set of extremal vectors $\Psi_{X_v,ex_i}, i \in \mathbf{N}$. Hence property 8 can be expressed by the following : there exists a constant broadcast-removing affine timing valid according to this dependency if and only if every pair of extremal vectors $(\Psi_{ex_1}, \Psi_{ex_2})$ is such that $\lim_{p\to\infty} \widehat{\Psi_{ex_1}, \Psi_{ex_2}} < 180°$ where $\widehat{\Psi_{ex_1}, \Psi_{ex_2}}$ denotes the angle of the two vectors.

When no constant BR affine timing exists, it is because the position of the emission set regarding to the reception set yields to a too wide extremal cone when the parameter size p increases. We can reduce this width by changing the relative positions of the emission and reception sets. This is realized by applying a transformation such as a translation directed by the generating vector on part of the reception set. An example of such a situation is presented on figure 2. This problem of non existence of a constant BR affine timing also occurs in the Algebraic Path Problem presented in section 4.

4. Example : the Algebraic Path Problem

Let us consider a weighted oriented graph $G = (V, E, w)$ where V is a finite set of n vertices, E is the set of edges and w is a function which associates a weight with each edge. These weights are defined in an $n \times n$ matrix A associated with G. $A = (a_{ij})$ is defined by $a_{ij} = w(i,j)$ if $(i,j) \in E$, $a_{ij} = 0$ otherwise.

The Algebraic Path Problem (APP) consists in finding, for all pairs of vertices (i,j), the quantities $d_{ij} = \oplus_{p \in M_{ij}} w(p)$ where M_{ij} denotes the set of all paths from i to j. The weight of a path $p \in M_{ij}$ is defined by $w(p) = \otimes_{e \in p} w(e)$.

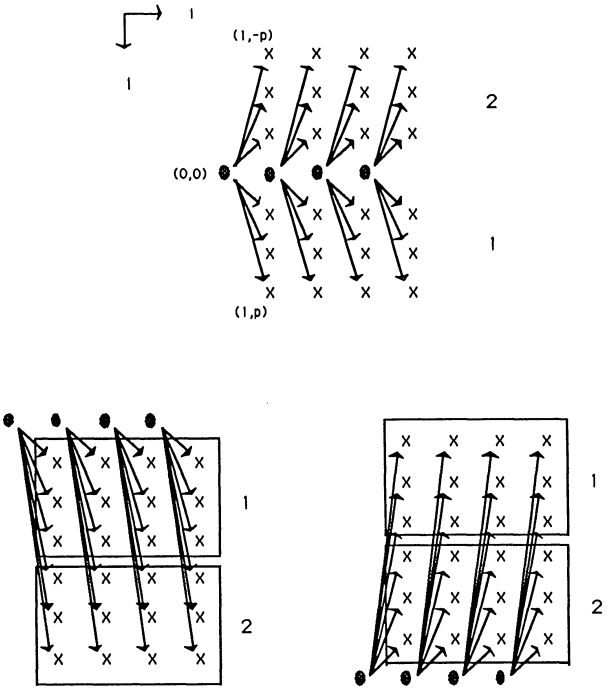


Figure 2 : transformation on a reception set when no constant BR affine timing exists

We denote $M_{ij}^{(k)}$ the set of all paths from i to j whose only intermediate vertices v are such that $1 \leq v \leq k$. When computing the recurrence $a_{ij}^{(k)} = \oplus_{p \in M_{ij}^k} w(p)$, we obtain $d_{ij} = a_{ij}^{(n)}$. The problem is defined by the following system of recurrence equations :

$$
\begin{array}{lll}
(1) & a(k,k,k) & = a(k,k,k-1) \quad 1 \leq k \leq n \\
(2) & a(i,j,k) & = a(i,j,k-1) \otimes a(k,j,k) \quad 1 \leq i,k \leq n,\ i \neq k,\ j = k \\
(3) & a(i,j,k) & = a(i,j,k-1) \otimes a(i,k,k) \quad 1 \leq j,k \leq n,\ j \neq k,\ i = k \\
(4) & a(i,j,k) & = a(i,j,k-1) \oplus a(i,k,k) \otimes a(k,j,k-1) \\
 & & \qquad 1 \leq i,j,k \leq n,\ i \neq k,\ j \neq k
\end{array}
$$

This problem is characterized by 4 subdomains respectively associated with the 4 equations :

$$
\begin{aligned}
D_1 &= \{(k,k,k) \in \mathbb{Z}^3 \mid 1 \leq k \leq n\} \\
D_2 &= \{(i,k,k) \in \mathbb{Z}^3 \mid 1 \leq i,k \leq n,\ i \neq k\} \\
D_3 &= \{(k,j,k) \in \mathbb{Z}^3 \mid 1 \leq j,k \leq n,\ j \neq k\} \\
D_4 &= \{(i,j,k) \in \mathbb{Z}^3 \mid 1 \leq i,j,k \leq n,\ i \neq k,\ j \neq k\}
\end{aligned}
$$

Notice that D_1 is a convex domain while D_2 and D_3 are the union of 2 convex disjoints subsets and D_4 the union of 4 subsets. They are presented on figure 3.

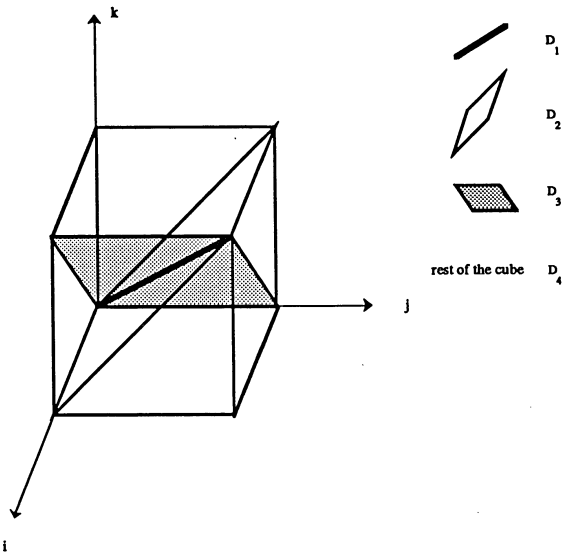


Figure 3 : domain of the Algebraic Path problem

On all four domains, we have the same dependency expressed by a constant inductive vector $\Psi_1 = (0,0,1)$. For each emission point $z_0 = (i_0, j_0, k_0)$ the corresponding reception set is reduced to one point $z = (i_0, j_0, k_0+1)$. Its generating vector is therefore the null vector.

On D_2 the use of $a(k,j,k)$ determines a reception set $\mathrm{Rec}_{(2)}\ (a, z_0) = \{(i, k_0, k_0) \mid 1 \leq i \leq n,\ i \neq k_0\}$ characterized by one generating vector $\Phi_{(2)} = (1,0,0)$ and a set of inductive vectors $\Psi_{(2)}(z_0) = (i - k_0, 0, 0)$ with $1 \leq i \leq n, i \neq k_0$. The emission set is $\mathrm{Emit}_{(2)} = \{(k_0, k_0, k_0) \mid 1 \leq k_0 \leq n\} = D_1$.

On D_3 the use of $a(i,k,k)$ determines a reception set $\mathrm{Rec}_{(3)}\ (a, z_0) = \{(k_0, j, k_0) \mid 1 \leq j \leq n,\ j \neq k_0\}$ characterized by one generating vector $\Phi_{(3)} = (0,1,0)$ and a set of inductive vectors $\Psi_{(3)}(z_0) = (0, j - k_0, 0)$ with $1 \leq j \leq n, j \neq k_0$. The emission set is $\mathrm{Emit}_{(3)} = \{(k_0, k_0, k_0) \mid 1 \leq k_0 \leq n\} = D_1$.

On D_4 the use of $a(i,k,k)$ determines a reception set $\mathrm{Rec}_{(4),1}\ (a, z_0) = \{(i_0, j, k_0) \mid 1 \leq j \leq n,\ j \neq k_0\}$ characterized by one generating vector $\Phi_{(4),1} = (0,1,0)$ and a set of inductive vectors $\Psi_{(4),1}(z_0) = (0, j - k_0, 0)$ with $1 \leq j \leq n, j \neq k_0$. The emission set is $\mathrm{Emit}_{(4),1} = \{(i_0, k_0, k_0) \mid 1 \leq i_0, k_0 \leq n, i_0 \neq k_0\} = D_2$.

Finally on D_4 too, the use of $a(k,j,k-1)$ determines a reception set $\mathrm{Rec}_{(4),2}\ (a, z_0) = \{(i, j_0, i_0) \mid 1 \leq i \leq n,\ i \neq i_0\}$ characterized by one generating vector $\Phi_{(4),2} = (1,0,0)$

and a set of inductive vectors $\Psi_{(4),2}(z_0) = (i - i_0, 0, 1)$ with $1 \leq i \leq n, i \neq i_0$. The emission set is $\text{Emit}_{(4),2} = \{(i_0, j_0, i_0 - 1) \mid 1 \leq i_0, j_0 \leq n, i_0 \neq j_0\}$.

The 3 dependencies defined by $\Psi_{(2)}$, $\Psi_{(3)}$ and $\Psi_{(4),1}$ are characterized by the fact that their inductive and generating vectors are colinear. Moreover their inductive vectors define a line, i.e. there are opposite inductive vectors such as $\Psi_{(2)}(n, n, n) = (1-n, 0, 0)$ for $z = (1, n, n)$ and $\Psi_{(2)}(1, 1, 1) = (n - 1, 0, 0)$ for $z = (n, 1, 1)$. This situation corresponds to case 3.1.1 mentioned above and forbids any affine timing. Translations on the respective domains are necessary to transform the sets of inductive vectors into cones strictly included into a half-line and therefore allow eventually a timing. These translations are presented on figure 4.

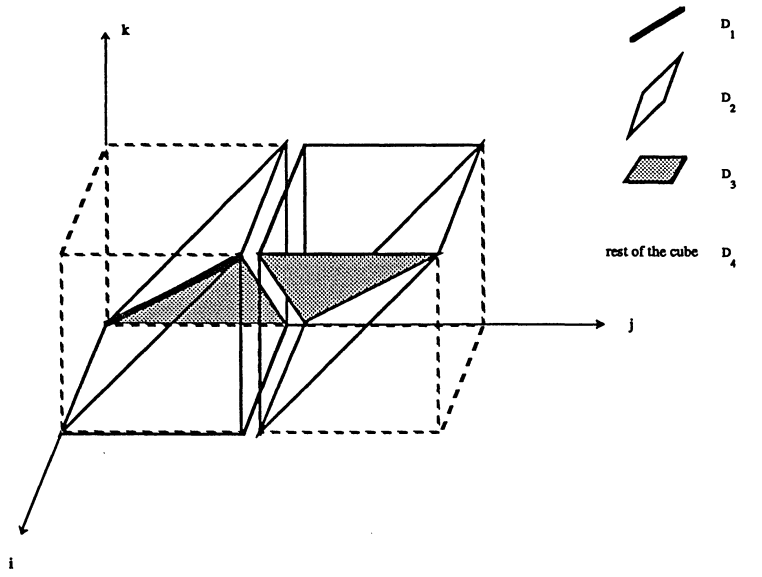


Figure 4 : first transformation on the domain of the APP

The last dependency defined by $\Psi_{(4),2}$ is characterized by a cone of inductive vectors associated with each emission point $z_0 : C(\Psi_{(4),2}, z_0) = \{(i - i_0, 0, 1) \mid 1 \leq i \leq n, i \neq i_0\}$. The extremal cone $\mathcal{C}_p$, convex hull of the cones $C(\Psi_{(4),2}, z_0)$, $z_0 \in \text{Emit}_{(4),2}$ is strictly included in a half-subspace of $\mathbf{R}^3$ and characterized by 2 extremal vectors $\Psi_{ex_1} = (1-n, 0, 1)$ and $\Psi_{ex_2} = (n-1, 0, 1)$. By property 8 there exists a constant affine timing for this dependency. However $\lim_{n\to\infty} \widehat{\Psi_{ex_1}, \Psi_{ex_2}} = 180°$ and by property 10 there is no constant BR affine timing. When applying a translation on domain D_4 as it is presented on figure 5, the angle $\widehat{\Psi_{ex_1}, \Psi_{ex_2}}$ is reduced to less than 45° and a constant BR affine timing now exists for this dependency.

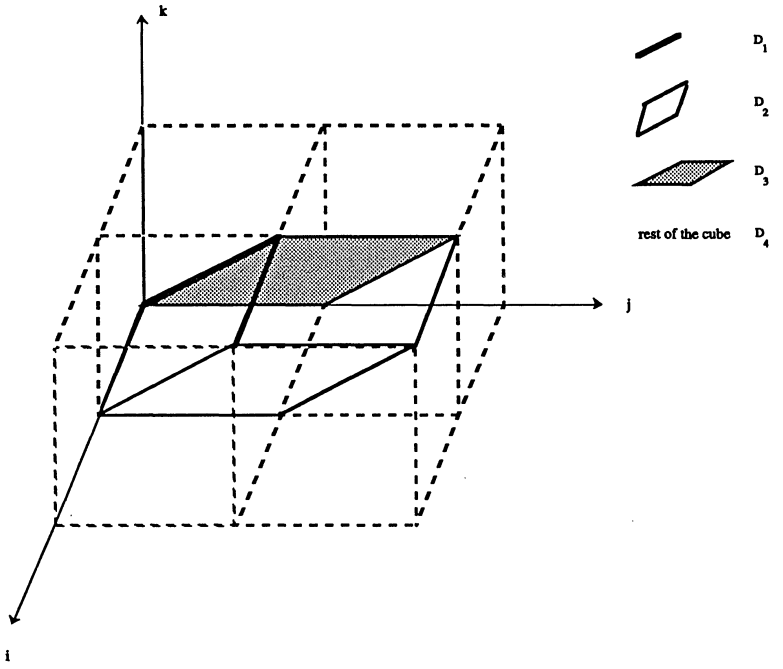


Figure 5 : second transformation on the domain of the APP

After all the transformations mentioned above, the dependencies are now expressed by :

$$\Psi_1 = (0,0,1)$$
$$\Psi_{(2)}(z_0) = (i - k_0, 0, 0) \quad for \quad k_0 + 1 \leq i \leq n + k_0$$
$$\Psi_{(3)}(z_0) = (0, j - k_0, 0) \quad for \quad k_0 + 1 \leq j \leq n + k_0$$
$$\Psi_{(4),1}(z_0) = (0, j - k_0, 0) \quad for \quad k_0 + 1 \leq j \leq n + k_0$$
$$\Psi_{(4),2}(z_0) = (i - i_0, 0, 1) \quad for \quad i_0 + 1 \leq i \leq n + i_0$$

The extremal vectors are then

$$\Psi_{(2),ex_1} = (1,0,0) \qquad \Psi_{(2),ex_2} = (n,0,0)$$
$$\Psi_{(3),ex_1} = (0,1,0) \qquad \Psi_{(3),ex_2} = (0,n,0)$$
$$\Psi_{(4),1,ex_1} = (0,1,0) \qquad \Psi_{(4),1,ex_2} = (0,n,0)$$
$$\Psi_{(4),2,ex_1} = (1,0,1) \qquad \Psi_{(4),2,ex_2} = (n,0,1)$$

The successive examination of each dependency has yield to some transformations of the problem. The existence of a global affine timing can now be tackled with the new values of the inductive vectors.Using these vectors, any affine timing $\lambda = (\lambda_1, \lambda_2, \lambda_3)$ must then verify $\lambda_1, \lambda_2, \lambda_3 > 0$. Since the generating vectors are $\Phi_{(2)} = \Phi_{(4),2} = (1,0,0)$ and $\Phi_{(3)} = \Phi_{(4),1} = (0,1,0)$, all the affine timing are broadcast removing timings. The most

efficient affine timing is then defined by $\lambda = (1,1,1)$. The corresponding computation time is $5n-4$. The initial specification of the APP did not allow any affine timing. The appropriate transformations on the different dependencies now yield to affine timings.

Conclusion

We have presented a geometrical framework to analyze the existence of affine timing functions for a given SARE. Using a dependency analysis in terms of generating and inductive vectors, we express conditions for the existence of affine timings valid according to one dependency. We characterize different validity conditions : the weak condition and the broadcast-removing condition. We emphasize the notions of constant and broadcast-removing timings. When these conditions are not satisfied by an ill-conditioned dependency, transformations are proposed in order to get an affine timing valid according to this dependency. Notice that there is not a unique transformation (for example the translation on figure 2 can be applied either on set 1 or on set 2). The different possibilities should be tried to give the largest chances to find an affine timing for the whole problem since they are not necessarily all compatible.

The process presented here is a step-by-step method. It does not guaranty the existence of an affine timing for the whole problem which is not possible since the computability of such parameterized SAREs has been proved undecidable ([SAQ 90]).

However it is very fruitful in practice where the initial SAREs of many problems often present ill-conditioned dependencies. The results presented here allow to determine a new specification of these problems without any ill-conditioned dependencies. These results have been illustrated on the example of the Algebraic Path Problem.

Bibliography

[BE-all89] Benaini A., Quinton P., Robert Y., Saouter Y., Tourancheau B. — Synthesis of a new Systolic Architecture for the Algebraic Path Problem, *IRISA Research Report,No1094*, 1989.

[CLP 88] Clauss Ph., Perrin G.R. — Synthesis of Process Arrays, *CONPAR'88, Manchester, G.B.*, 1988.

[DEI 85] Delosme J.M., Ipsen I.C.F. — An Illustration of a Methodology for the Construction of Efficient Systolic Architectures in VLSI, *Sd. Int. Symposium on VSLI Technology, Systems and Applications, Taipei, Taiwan, R.O.C.*, 1985, p. 268–273.

[DEL 88] Delosme J.M. — A Parallel Algorithm for the Algebraic Path Problem, *Int. Workshop on Parallel and Distributed Algorithms, M. Cosnard et al. editors, North-Holland*, 1988.

[FFW 87] FORTES J.A.B., FU K.S., WAH B.W. — Systematic Approaches to the Design of Algorithmically Specified Systolic Arrays, *Int. Conf. on Acoustics, Speech and Signal Processing*, 1987.

[KMW 67] KARP R.M., MILLER R.E., WINOGRAD S. — The Organization of Computations for Uniform Recurrence Equations, *JACM*, t. **14,3**, 1967.

[KUN 82] KUNG H.T. — Why systolic architectures ?, *Computer*, t. **15-1**, 1982, p. 37–46.

[MCP 90] MONGENET C., CLAUSS Ph., PERRIN G.R. — Geometrical Tools to map Systems of Affine Recurrence Equations on Regular Arrays, *Research Report, LIB, Université de Franche-Comté*, 1990.

[MCP 91] MONGENET C., CLAUSS Ph., PERRIN G.R. — A Geometrical Cding to Compile Affine Recurrence Equations on Regular Arrays, *Fifth Int. Parallel Processing Symposium, Anaheim, California*, 1991.

[MOF 86] MOLDOVAN D.I., FORTES J.A.B. — Partitioning and Mapping Algorithms into Fixed Size Systolic Arrays, *IEEE Transactions on Computers*, t. **35-1**, 1986, p. 1–12.

[MOL 83] MOLDOVAN D.I. — On the Design of Algorithms for VLSI Systolic Arrays, *Proc IEEE conf.*, t. **71-1**, 1983, p. 113–120.

[MON 85] MONGENET C. — Une Méthode de Conception d'Algorithmes Systoliques, Résultats Théoriques et réalisation, *Thèse INPL, Nancy*, 1985.

[MOP 87] MONGENET C., PERRIN G.R. — Synthesis of Systolic Arrays for Inductive Problems, *Conf. PARLE, LNCS 259*, 1987.

[QUI 84] QUINTON P. — Automatic Synthesis of Systolic Arrays from Uniform Recurrence Equations, *Proc. IEEE 11th Int. Symp. on Computer Architecture,Ann Arbor,MI,USA*, 1984, p. 208–214.

[QVD 89] QUINTON P., VAN DONGEN V. — The Mapping of Linear Recurrence Equations on Regular Arrays, *The Journal of VLSI Signal Processing*, t. **1**, 1989, p. 95–113.

[ROT 87] ROBERT Y., TRYSTRAM D. — Systolic Solution of the Algebraic Problem, *Int. Workshop on Systolic Arrays, Oxford, Adam-Hilger*, 1987, p. 171–180.

[SAQ 90] SAOUTER Y., QUINTON P. — Computability of Recurrence Equations, *TR-1203, IRISA, Rennes*, 1990.

[YAC 88] YAACOBI Y., CAPPELLO P.R. — Scheduling a System of Affine Recurrence Equations onto a Systolic Array, *Int. Conf. on Systolic Arrays, San Diego, USA*, 1988, p. 373–382.

On the Computational Complexity of Optimal Sorting Network Verification

Ian Parberry*
Department of Computer Science
The Pennsylvania State University

Abstract

A *sorting network* is a combinational circuit for sorting, constructed from comparison-swap units. The depth of such a circuit is a measure of its running time. It is reasonable to hypothesize that only the fastest (that is, the shallowest) networks are likely to be fabricated. It is shown that the problem of verifying that a given sorting network actually sorts is Co-$\mathcal{NP}$ complete even for sorting networks of depth only $4\lceil \log n \rceil + O(1)$ greater than optimal. This is shallower than previous depth bounds by a factor of two.

1 Introduction

A *comparator network* is a combinational circuit constructed from comparison-swap units called *comparators*. A *sorting network* is a comparator network which sorts. The *size* of a comparator network is the number of comparators used. The *depth* is the number of layers of comparators, where each layer receives input only from the layers above it. Comparator networks can be fabricated relatively easily using VLSI techniques. It would be useful to be able to verify whether a given sorting network actually works. It is well known that in order to test whether a given n-input comparator network is a sorting network, it is sufficient to check that it sorts the $2^n - n - 1$ nonsorted zero-one inputs (which we will call bit-strings). This observation is called the *zero-one principle*.

Comparator networks which sort all but a single nonsorted bit-string are known. That is, for all nonsorted sequences of n bits x, there exists an n-input comparator network which sorts all

*Research supported by NSF Grant CCR-8801659. Author's current address: Department of Computer Sciences, P.O. Box 13886, University of North Texas, Denton, TX. 76203-3886, U.S.A. Electronic mail: ian@dept.csci.unt.edu.

bit-strings except x. These are called *single exception sorting networks.* Chung and Ravikumar [5] give a recursive construction of an n-input single exception sorting network of polynomial size and depth. They further deduce in [6] that the sorting network verification problem is Co-$\mathcal{NP}$ complete. Parberry [16] gave a non-recursive construction for a single exception sorting network of depth $D(n-1) + 2\lceil\log(n-1)\rceil + 2$, where $D(n)$ is the minimum depth of an n-input sorting network, and deduced, using the construction of Chung and Ravikumar [6], that the problem of verifying sorting networks of depth $2D(n) + 6\lceil\log n\rceil + O(1)$ is Co-$\mathcal{NP}$ complete. We will show that the sorting network verification problem remains Co-$\mathcal{NP}$ complete even for sorting networks of depth $D(n) + 4\lceil\log n\rceil + O(1)$.

The remainder of this paper is divided into six sections. The first section contains a more formal definition of a sorting network, and briefly describes some standard results. The second section contains a proof that a modified version of the satisfiability problem is $\mathcal{NP}$ complete. The third section contains a sketch of the reduction from that problem to the sorting network verification problem. The fourth section contains the details of the construction of an important component used in that reduction — a comparator network that sorts all except a specific set of inputs. The construction of this component uses the single exception sorting network of Parberry [16]. A slightly improved single exception sorting network is given in the fifth section of this paper. The sixth section contains details on how to reduce the depth of the construction to give the required result.

Let $\mathbf{N}$ denote the natural numbers, and $\mathbf{B}$ denote the Boolean set $\{0,1\}$. Members of $\mathbf{B}^n$ (the set of n-tuples of bits) will be called *bit-strings.* We will use the standard regular-expression notation to describe certain sets of bit-strings, for example, 0^n1^m denotes a single bit-string consisting of n ones followed by m zeros, and $(00 \cup 11)^n$ denotes the set of n pairs of bits, where each pair is either 00 or 11, that is,

$$\{x_1y_1 \cdots x_ny_n \mid x_i = y_i \in \mathbf{B} \text{ for } 1 \leq i \leq n\}.$$

If A and B are sets, $A \backslash B$ denotes $\{x \mid x \in A, \text{ but } x \notin B\}$.

2 Sorting Networks

One of the early investigations into parallel sorting concerned the *Bose-Nelson sorting problem*, named by Floyd and Knuth [9], after Bose and Nelson [4]. The problem involves sorting n values by using a sequence of *oblivious* in-situ comparison-and-swap operations; that is, a sequence of comparisons between the ith and jth value, where i and j are independent of the values being sorted. The obliviousness property allows the following elegant hardware interpretation of the problem. Suppose that we are given a basic unit of hardware called a *comparator.* A comparator

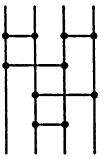


Figure 1: A 4-input sorting network of depth 3 and size 5.

takes two values as input and outputs them in ascending order. A *comparator network* consists of n parallel *channels*, which can be thought of as wires carrying values, to which comparators are attached. The network is divided into a finite number of *levels*. Each level consists of one or more comparators. Each comparator is attached to two channels. At most one comparator is placed on any channel at each level. Channels, in our diagrams, will be drawn as vertical lines, and comparators as horizontal lines with heavy dots emphasizing the connection-points. Levels are numbered vertically from top-to-bottom, and channels are numbered horizontally from left-to-right. Level 0 will be used to denote the inputs.

Let C be a comparator network. We define the value carried by channel i of C at level j on input $x = (x_1, \ldots, x_n)$, written $V(C, x, i, j)$, as follows. $V(C, x, i, 0) = x_i$ and for $j > 0$:

- If there is a comparator between channels i and $k > i$ at level j, then

$$V(C, x, i, j) = \min(V(C, x, i, j-1), V(C, x, k, j-1)).$$

- If there is a comparator between channels i and $k < i$ at level j, then

$$V(C, x, i, j) = \max(V(C, x, i, j-1), V(C, x, k, j-1)).$$

- Otherwise, $V(C, x, i, j) = V(C, x, i, j-1)$.

The output of an n-input, d-level comparator network C on input x is

$$V(C, x) = (V(C, x, 1, d), V(C, x, 2, d), \ldots, V(C, x, n, d)).$$

If for all inputs x, $V(C, x)$ is in nondecreasing order from left to right, then C is called a *sorting network*.

For example, Figure 1 shows a 4-input sorting network. The comparators on the first level compare the values in pairs. The second layer of comparators determines the maximum and minimum

values: the minimum of the two minima is the minimum overall, and the maximum of the two maxima is the maximum overall. The third layer puts the remaining two values into the correct order.

Each level of a comparator network consists of a set of independent comparisons which may be performed in parallel. The number of levels is thus a reasonable measure of parallel time. This is called the *depth* of the network. Another resource of interest is its *size*, which is defined to be the number of comparators used. We will call an n-input sorting network *optimal* if it is $\mathcal{P}$-uniform (that is, there is an algorithm which outputs a description of the sorting network, on input n, in time polynomial in n), and has depth $O(\log n)$. Optimal sorting networks have size $O(n \log n)$.

There have been a number of recent results on optimal sorting networks. For a survey of some of these results, see Parberry [13, 14]. Sorting networks of optimal depth are known for $n \leq 10$ (Parberry [15]) and with optimal size for $n \leq 8$ (Knuth [11]). For all practical purposes, the best sorting networks are constructed using the recursive technique of Batcher [3], which gives depth $O(\log^2 n)$ and size $O(n \log^2 n)$, although some small improvement in the lower-order terms of the size have been made by Drysdale [7] and Van Voorhis [19] in exchange for a large increase in depth. Ajtai, Komlós and Szemerédi [1, 2] have demonstrated that asymptotically optimal (logarithmic depth and log-linear size) sorting networks can be constructed; however their method remains impractical for reasonable values of n despite the efforts of Paterson [17], since the constant multiples involved are extremely large.

We will make use of two standard results. Firstly, if we allow channels to be permuted obliviously between each layer, then the model is the same, in the sense that we can remove the permutations in polynomial time in a manner that affects neither the size of the network, the depth of the network, nor its ability to sort. This result is due to Floyd and Knuth [8]. Therefore, the sorting network verification problems with and without permutations allowed are polynomial-time equivalent. The inclusion of such permutations will simplify the presentation of our results since it will allow us to physically group together logically related channels. The second standard result is the *zero-one* principle, which states that a comparator network is a sorting network *iff* it sorts all bit-strings. In the light of this result, we will throughout this paper consider only zero-one inputs. These results are discussed at great length in Knuth [11], and Parberry [13, 16, 14]. We assume that the reader is familiar with the techniques and terminology of the theory of $\mathcal{NP}$ completeness. The reader who is not should consult a standard reference, such as Garey and Johnson [10].

More formally, the sorting network verification problem can be stated as a decision problem as follows:

Sorting Network Verification (NONSORT)

INSTANCE: A comparator network C.

QUESTION: Is there an input which C does not sort?

The number of comparators in C is a reasonable measure of input size for any sorting network verification program. Furthermore, the number of inputs to C is a valid measure of input size for any program which verifies optimal sorting networks. Clearly NONSORT $\in \mathcal{NP}$, since if we are given a comparator network C and an input x, the output of C on x can be determined in time polynomial in the number of comparators in C. It remains to show that NONSORT is $\mathcal{NP}$ hard. This will imply that the original sorting network verification problem is Co-$\mathcal{NP}$ complete.

3 One-in-Three 3SAT

The following set-theoretic problem was shown to be $\mathcal{NP}$ complete by Schaefer [18].

One-in-Three 3SAT (T3SAT)

INSTANCE: Sets $S_1, \ldots, S_n$ with $|S_i| \leq 3$ for $1 \leq i \leq n$.

QUESTION: Does there exist a set S such that $|S \cap S_i| = 1$ for $1 \leq i \leq n$?

Garey and Johnson [10] list Schaefer's result in a slightly different form which more clearly illustrates that it is a restricted case of the Satisfiability Problem. Define a *clause* over a set V to be a subset of $V \times V \times V$. A set $S \subseteq V$ is said to *satisfy* a clause (v_1, v_2, v_3) iff $|\{i \mid v_i \in S\}| = 1$. If C is a list of clauses over V, then $S \subseteq V$ is said to be a *satisfying assignment* for C iff S satisfies all clauses in C. Intuitively, the elements of V are variables, and S is a set of variables to be assigned the value `true`. A clause represents a ternary Boolean function which is `true` iff exactly one of its inputs is `true`. A list of clauses represents the conjunction of a list of these functions.

Modified One-in-Three 3SAT (M3SAT)

INSTANCE: A list of clauses C over a set V.

QUESTION: Is there a satisfying assignment for C?

Lemma 3.1 *M3SAT is $\mathcal{NP}$ complete.*

PROOF: Obviously, M3SAT $\in \mathcal{NP}$. It is easy to show that T3SAT $\leq_m^p$ M3SAT. □

We will find it useful to consider a restricted version of M3SAT.

Balanced One-in-Three 3SAT (B3SAT)

INSTANCE: A set of variables V and a list of clauses C over V in which every variable appears exactly three times.

QUESTION: Is there a satisfying assignment for C?

Note that each instance of B3SAT with n clauses must have n variables. (Since there are n clauses, there are $3n$ instances of variables. Since every variable has exactly 3 instances, there must be n variables.) Also, every *satisfiable* instance of B3SAT with n clauses must have n divisible by 3. (Let V be a variable-set, and $C = (C_1, \ldots, C_n)$ be a list of clauses over V. Suppose $S \subseteq V$, where $|S| = m$. Then C contains $3m$ instances of variables that are members of S. But if S is a satisfying assignment, since there are n clauses, each of which must contain exactly one instance of a variable in S, there must be n instances of variables that are members of S. Therefore $n = 3m$.)

Theorem 3.2 *B3SAT is $\mathcal{NP}$ complete.*

PROOF: Clearly B3SAT $\in \mathcal{NP}$. By Lemma 3.1, it suffices to show that M3SAT $\leq_m^p$ B3SAT.

Suppose $C_1, \ldots, C_n$ is an instance of M3SAT over variable-set V. The corresponding instance of B3SAT is constructed as follows. For every clause C_i, there are three new clauses $(a_{i,1}, a_{i,2}, a_{i,3})$, $(b_{i,1}, b_{i,2}, b_{i,3})$, $(c_{i,1}, c_{i,2}, c_{i,3})$ called *structural enforcers*. For each variable $v \in V$, let

$$X_v = \{a_{i,j}, b_{i,j}, c_{i,j} \mid v \text{ is the } j\text{th variable in } C_i\}.$$

Suppose

$$X_v = \{a_{i_1,j_1}, b_{i_1,j_1}, c_{i_1,j_1}, \ldots, a_{i_m,j_m}, b_{i_m,j_m}, c_{i_m,j_m}\}$$

for some $m \leq n$. Define

$$I_v = \{(a_{i_k,j_k}, b_{i_k,j_k}, c_{i_k,j_k}) \mid 1 \leq k \leq m\}.$$

If $k = 1$, define $E_v = I_v$, otherwise define

$$\begin{aligned} E_v \;=\; & \{(b_{i_1,j_1}, c_{i_1,j_1}, a_{i_2,j_2}), (b_{i_2,j_2}, c_{i_2,j_2}, a_{i_3,j_3}), \\ & \ldots, (a_{i_{m-1},j_{m-1}}, b_{i_{m-1},j_{m-1}}, c_{i_m,j_m}), (a_{i_m,j_m}, b_{i_m,j_m}, a_{i_1,j_1})\} \end{aligned}$$

For each $(p, q, r) \in I_v$ and each $(p, q, r) \in E_v$ there are three clauses (p, x, y), (q, x, y), and (r, x, y), called *equality enforcers*, where x and y are new variables not previously used. It is clear that this transformation can be computed in time polynomial in n.

The new instance of M3SAT consisting of the structural enforcers and the equality enforcers is actually an instance of B3SAT, that is, that every variable appears in exactly three clauses.

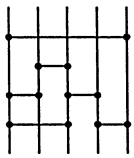


Figure 2: A variable component.

Each of the $a_{i,j}$, $b_{i,j}$, and $c_{i,j}$ variables occurs in exactly one structural enforcer, and two equality enforcers (one corresponding to an element of I_v, and one corresponding to an element of E_v, for some variable v). The extra variables in the equality enforcers also appear in exactly three clauses, by construction.

We claim that $C_1, \ldots, C_n$ is satisfiable iff there is a satisfying assignment for the structural enforcers and the equality enforcers. Clearly if S is a satisfying assignment for $C_1, \ldots, C_n$, then $\cup_{v \in S} X_v$ satisfies every structural enforcer and every equality enforcer. Conversely, suppose that S satisfies the structural enforcers and equality enforcers. Since the equality enforcers corresponding to members of I_v are satisfied by S, then for all variables $v \in V$, either $X_v \cap S = \{\}$ or $X_v \subseteq S$. Thus if we set $T = \{v \mid X_v \subseteq S\}$, then T satisfies $C_1, \ldots, C_n$. □

4 The Reduction

In order to show that NONSORT is $\mathcal{NP}$ complete, it is sufficient to show that B3SAT $\leq_m^p$ NONSORT. Suppose we are given an instance of B3SAT, that is, a list of clauses $C = (C_1, \ldots, C_n)$ over a set of variables $V = \{v_1, \ldots, v_n\}$ such that every variable in V appears exactly three times in C. We will construct a comparator network with $5n$ inputs. An input $x = (x_1, \ldots, x_{5n})$ to the comparator network is said to *correspond to assignment* S for C iff for all $1 \leq i \leq n$, $x \in (0^5 \cup 1^5)^n$, and $v_i \in S$ iff $x_{5i-4} = 1$. Our comparator network will sort only inputs that do not correspond to satisfying assignments for C, that is, inputs that do not correspond to any assignment, and inputs that correspond to nonsatisfying assignments. Therefore, it will be a sorting network iff C is not satisfiable.

For each variable $v \in V$, we have a *variable component*, consisting of five channels and six comparators, of depth three (see Figure 2). For each clause C_i we have a *clause component*, consisting of three channels and three comparators, of depth three (see Figure 3). These components are connected as follows.

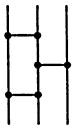


Figure 3: A clause component.

The $5n$ inputs are divided into quintuples and put into n variable components, one for each variable. The center three outputs of each variable component are put into the inputs of the appropriate clause components. Specifically, suppose $C_i = (v_{i,1}, v_{i,2}, v_{i,3})$, for $1 \leq i \leq n$ where $v_{i,1}, v_{i,2}, v_{i,3} \in V$. Let $c(i,j)$ denote the number of previous occurrences of $v_{i,j}$ in $C_1, \ldots, C_i$, that is, $c(i,j) = |\{v_{k,l} \mid k < i \text{ or } (k = i \text{ and } l < j)\}| + 1$, for $1 \leq i \leq n$, $1 \leq j \leq 3$. Note that $0 \leq c(i,j) \leq 2$. Then for each clause $C_i = (v_{i,1}, v_{i,2}, v_{i,3})$, put the $(c(i,k)+2)$th output of the variable component corresponding to $v_{i,k}$ into the kth input of the clause component corresponding to C_i, for $1 \leq k \leq 3$. The first two outputs of the clause components are routed to the far right, and the last output of the clause components is routed to the far left. All of the channels are then put into a special component called a *selector*, which will sort every input except those which correspond to satisfying assignments.

The centre three outputs of a variable component corresponding to variable $v \in V$ are copies of variable v which are to be used in the clauses in which v appears. It can be demonstrated (by case analysis of the 32 different input strings) that if the first and last outputs of a variable component are zero, then all outputs are zero, and if the first and last outputs of a variable component are one, then all outputs are one. Thus the centre three outputs of a variable component carry a valid truth assignment for the variable v if the first and last output of the variable component are identical. The output of a clause component is 001 iff exactly one of its input channels carries a one (since each clause component is actually a three-input sorting network).

Suppose the input to the comparator network corresponds to a satisfying assignment S. Since there is exactly one variable from each clause in S, the first n inputs to the selector will be one, and the last $2n$ inputs to the selector will be zero. The remaining $2n$ inputs will consist of n pairs, each of which is either 00 or 11. Since variable appears in exactly three clauses, S must contain $n/3$ variables. Therefore $2n/3$ pairs will be 00, and $n/3$ pairs will be 11. That is, the input to the selector will be a member of the set $1^n(\mathbf{B}^{2n}_{2n/3} \cap (00 \cup 11)^n)0^{2n}$, where $\mathbf{B}^n_k = \{x \in \mathbf{B}^n \mid x \text{ has exactly } k \text{ ones}\}$. Conversely, if the input to the comparator network does not correspond to satisfying assignment, then it is clear that the input to the selector will not be of this form. The above reduction can be carried out in time polynomial in n provided there exist $\mathcal{P}$-uniform selectors. The construction of

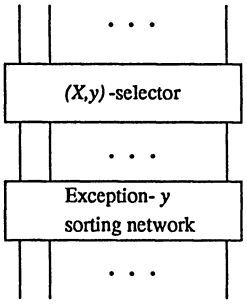


Figure 4: An exception-X sorting network.

the selector is the subject of the next section.

5 Selective Sorting Networks

The *selector* used in the previous section was a comparator network which selectively sorts all bit-strings except for members of a certain, slightly obscure set. We use the generic term *selective sorting network* for such a comparator network. If the comparator network sorts all members of $\mathbf{B}^n \backslash X$, than it is called an *exception-X sorting network.*

Comparator networks which sort all but a single bit-string are known. That is, for all nonsorted $x \in \mathbf{B}^n$, there exists an n-input comparator network which sorts all members of $\mathbf{B}^n$ except x. These are called *single exception sorting networks.* Chung and Ravikumar [5] give a recursive construction of an n-input single exception sorting network of polynomial size and depth. Parberry [16] gives a non-recursive construction for a single exception sorting network of depth $D(n-1)+2\lceil\log(n-1)\rceil+2$, where $D(n)$ is the minimum depth of an optimal n-input sorting network. We wish to find a particular selective sorting network with $5n$ inputs, where n is a multiple of three, and an exception set of size

$$\binom{n}{n/3}.$$

If $X \subseteq \mathbf{B}^n$, and $y \in \mathbf{B}^n$ is nonsorted, a comparator network C is an (X,y)-*selector* iff for all $x \in \mathbf{B}^n$, the output of C on input x is y iff $x \in X$.

Theorem 5.1 *If $X \subseteq \mathbf{B}^n$, and there exists an (X,y)-selector of depth $D_X(n)$, then there exists an exception-X sorting network of depth $D_X(n) + D(n) + 2\lceil\log n\rceil + 2$.*

PROOF: Suppose $X \subseteq \mathbf{B}^n$, and there exists a comparator network C of depth $D_X(n)$ and nonsorted

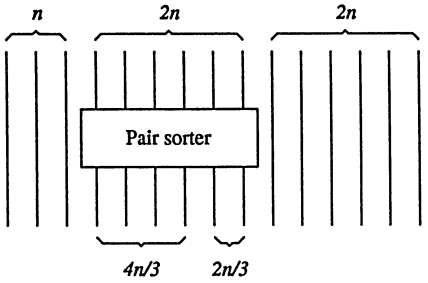


Figure 5: An X-selector.

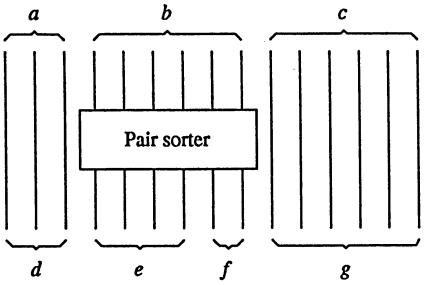


Figure 6: An X-selector labelled for the proof of Theorem 5.2.

$y \in \mathbf{B}^n$ such that for all $x \in \mathbf{B}^n$, the output of C on input x is y iff $x \in X$. Then an exception-X sorting network is constructed by composing C and an exception-y sorting network as in Figure 4. The total depth is bounded above by the depth of C plus the depth of the exception-y sorting network. Parberry [16] gives a construction for exception-y sorting networks for all nonsorted y of depth $D(n) + 2\lceil \log n \rceil + 2$. □

The exception-set from the previous section is the set of bit-strings of the following form: n ones, followed by n pairs of bits, $2n/3$ of which are 00 and $n/3$ of which are 11, followed by $2n$ zeros. That is,

$$X = 1^n(\mathbf{B}^{2n}_{2n/3} \cap (00 \cup 11)^n)0^{2n}.$$

Theorem 5.2 *Suppose $n \in \mathbf{N}$, and $X = 1^n(\mathbf{B}^{2n}_{2n/3} \cap (00 \cup 11)^n)0^{2n}$. There exists an exception-X sorting network of depth $D(n) + D(5n) + 2\lceil \log(5n) \rceil + 3$.*

PROOF: Let $X = 1^n(\mathbf{B}^{2n}_{2n/3} \cap (00 \cup 11)^n)0^{2n} \subseteq \mathbf{B}^{5n}_{5n/3}$. By Theorem 5.1, we can build an exception-

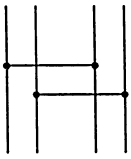


Figure 7: A pair comparator.

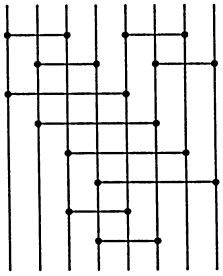


Figure 8: A 4-pair sorter constructed from Figure 1.

X sorting network from an (X,y)-selector. An (X,y)-selector with $y = 1^n0^{4n/3}1^{2n/3}0^{2n}$ can be constructed as in Figure 5, by leaving the first n and the last $2n$ channels alone, and placing a $2n$-input *pair sorter* on the remaining $2n$ channels. A *pair sorter* is a comparator network which has an even number of inputs. The inputs are treated as bit-pairs. Each bit-pair is sorted, and the sorted bit-pairs are then sorted into lexicographic order, that is, the output of the pair sorter is a member of $(00)^*(01)^*(11)^*$. We will return to the construction of the pair sorter later in this proof.

Suppose the input to the network shown in Figure 5 is $abc \in \mathbf{B}^{5n}_{5n/3}$, where $a \in \mathbf{B}^n$, $b, c \in \mathbf{B}^{2n}$, and its output is $defg \in \mathbf{B}^{5n}_{5n/3}$, where $d \in \mathbf{B}^n$, $e \in \mathbf{B}^{4n/3}$, $f \in \mathbf{B}^{2n/3}$, and $g \in \mathbf{B}^{2n}$ (see Figure 6). We claim that $defg = 1^n0^{4n/3}1^{2n/3}0^{2n}$ iff $abc \in X$. Suppose $abc \in X$. then $b \in \mathbf{B}^{2n}_{2n/3} \cap (00 \cup 11)^n$. Since $b \in (00 \cup 11)^n$, the output of the pair sorter, ef, is a sorted sequence of bits. Since $b \in \mathbf{B}^{3n}_{2n/3}$, $ef = 0^{4n/3}1^{2n/3}$. Therefore, since $d = a$ and $g = c$, $defg = 1^n0^{4n/3}1^{2n/3}0^{2n}$, as claimed. Conversely, suppose that $abc \notin X$. We claim that $defg \neq 1^n0^{4n/3}1^{2n/3}0^{2n}$, that is, either $d \neq 1^n$, $e \neq 0^{4n/3}$, $f \neq 1^{2n/3}$, or $g \neq 0^{2n}$. Since $abc \notin X$, either $a \neq 1^n$, (in which case there is a zero in d), $c \neq 0^{2n}$, (in which case there is a one in g), or $b \notin \mathbf{B}^{2n}_{2n/3} \cap (00 \cup 11)^n$. In the latter case, suppose $b \in \mathbf{B}^{2n}_m$. If $m = 2n/3$, then $b \notin (00 \cup 11)^n$, and so there is a one in e. If $m < 2n/3$, then there is a zero in f. If $m > 2n/3$, then there is a one in e. In all cases, $defg \neq 1^n0^{4n/3}1^{2n/3}0^{2n}$, as claimed.

It is clear that the depth of our $5n$-input (X,y)-selector is equal to the depth of a $2n$-input

pair sorter. This pair-sorter is constructed as follows. The pairs are sorted with a single layer of comparators, one per pair. A pair of pairs can be sorted into lexicographic order by comparing the first element of the first pair with the first element of the second pair, and simultaneously comparing the second element of the first pair with the second element of the second pair (see Figure 7). Therefore n pairs can be sorted into lexicographic using a comparator network obtained from an n-input sorting network by doubling all the channels, and replacing every comparator with a pair-comparator (for example, see Figure 8). Thus the depth of the pair sorter is $D(n) + 1$. Therefore by Theorem 5.1, the depth of our $15n$-input exception-X sorting network is $D(n) + D(5n) + 2\lceil\log(5n)\rceil + 3$. □

Theorem 5.3 *NONSORT is $\mathcal{NP}$ complete even for n-input sorting networks of depth* $2D(n) + 2\lceil\log n\rceil + 9$.

PROOF: The reduction is as described in Section 4, using the selector from Theorem 5.2. The depth of an n-input comparator network constructed using this technique is bounded above by 3 for the variable components, plus 3 for the clause components, plus $2D(n) + 2\lceil\log n\rceil + 3$ for the selector, a total of $2D(n) + 2\lceil\log n\rceil + 9$. □

6 Improved Single Exception Sorting Networks

In the construction of the selector in the previous section, we used the single exception sorting network from Parberry [16], which has depth $D(n-1)+2\lceil\log(n-1)\rceil+2$. It is possible to improve that result by a small constant. Suppose $n \in \mathrm{N}$, and $1 \leq k < n$. A better single exception sorting network with exception 1^k0^{n-k} is constructed as follows (see Figure 9). In [16], a single exception sorting network with this exception is called a *canonical k-ones single exception sorting network.* The first k inputs are put into a *min network.* The leftmost output of this network is the minimum of its inputs. The last $k-1$ outputs of this network, and the remaining $n-k$ channels are sorted together. The leftmost channel, and the leftmost $n-k-1$ outputs of the sorting network are put into an *insertion network.* This network takes as input a single value followed by a sorted sequence, and it inserts the new value into the correct place in the sequence. It is straightforward to recursively construct n-input min and insertion networks of depth $\lceil\log n\rceil$.

It is easy to see that this construction gives a single exception sorting network. Suppose the input to the network is 1^k0^{n-k}. Then the leftmost output of the min network is 1, and the output of the sorting network is $0^{n-k}1^{k-1}$, and hence the output of the insertion network is $0^{n-k-1}101^{k-1}$,

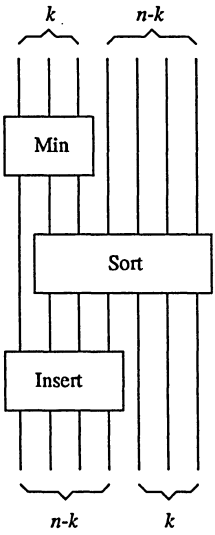


Figure 9: A single exception sorting network.

which is not sorted. Now suppose the input to the network is not 1^k0^{n-k}. In particular, suppose it is $ab \neq 1^k0^{n-k}$, where $a \in \mathbf{B}^k$, $b \in \mathbf{B}^{n-k}$. Then either $a \neq 1^k$ or $b \neq 0^{n-k}$. In the former case, the leftmost output of the min network is 0, hence the values on the channels immediately after the sorting network are sorted, and they remain sorted through the rest of the network. If $a = 1^k$ and $b \neq 0^{n-k}$, then b contains at least one 1, and so the insertion network carries the 1 from the leftmost channel into the correct place.

Theorem 6.1 *For all $n > 1$ and all nonsorted bit-strings x, there exists an n-input comparator network of depth $D(n-1) + 2\lceil \log n \rceil - 1$. which sorts all bit-strings except x.*

PROOF: If $D(n)$ is the depth of the optimal n-input sorting network, then the depth of the canonical k-ones single exception sorting network shown in Figure 9 is

$$D(n-1) + \lceil \log \lceil n/2 \rceil \rceil + \lceil \log \lfloor n/2 \rfloor \rceil \leq D(n-1) + 2\lceil \log n \rceil - 1.$$

If x is an arbitrary nonsorted bit-string with k ones, then a comparator network can be constructed from the canonical k-ones single exception sorting network using the technique of Theorem 7 of [16]. □

This new construction can be used to improve slightly on the results in [16], and to improve

slightly on the bound in Theorem 5.3.

Theorem 6.2 *NONSORT is $\mathcal{NP}$ complete even for n-input sorting networks of depth* $2D(n) + 2\lceil \log n \rceil + 6$.

PROOF: Use Theorem 6.1 in the construction of Theorem 5.3. □

7 Improved Selectors

The selective sorting network described in Section 5 was developed using general techniques which will work for any exception set of the appropriate form. However, the exception set that appears in the reduction of Section 4 has additional special properties which allow a reduction in the depth bound.

Let us re-draw the selector using the canonical single exception sorting network from Section 6 (see Figure 10). The selector consists of the pair sorter of depth $D(n)$, a sorting network of depth $D(5n)$, and a min network and an insertion network of combined depth $2\lceil \log(5n) \rceil - 1$. Thus the total depth of the $5n$-input selector is $2D(5n) + 2\lceil \log(5n) \rceil - 1$. We route the last $2n/3$ outputs from the pair sorter to the right of the first $4n/3$ outputs, so that the exception becomes $1^{5n/3}0^{10n/3}$ instead of $1^n0^{4n/3}1^{2n/3}0^{2n}$.

However, we have not used the fact that the output of the pair sorter is sorted in pairs. Thus there is no need for the sorting network in the part of the selector corresponding to the single exception sorting network. If we sort the first n and last $2n$ values in parallel with the pair sorter, then all we need to do is merge the sorted pairs in two groups after the pair sorter, and merge these with the outputs of the sorters (see Figure 11).

Parberry [12] gives a construction for an n-input pair merger (which is called an *alternating merging network* in that reference), of depth $\lceil \log n \rceil$. Batcher [3] gives a construction for an n-input merging network of depth $\lceil \log n \rceil + 1$. Therefore the new $5n$-input selector has depth bounded above by $D(2n)$ for the sorters, plus $\lceil \log(4n/3) \rceil \leq \lceil \log(5n) \rceil - 1$ for the pair mergers, plus $\lceil \log(5n) \rceil + 1$ for each of two mergers, plus $\lceil \log(5n) \rceil$ for the inserter, giving a total of $D(2n) + 4\lceil \log(5n) \rceil + 1$.

Therefore, we obtain the main result of this paper:

Theorem 7.1 *NONSORT is $\mathcal{NP}$ complete even for n-input sorting networks of depth* $D(n) + 4\lceil \log n \rceil + 7$.

PROOF: The proof is similar to that of Theorem 5.3, substituting the more efficient selector described in this section. The depth in this case is bounded above by 3 for the variable components, 3 for the clause components, plus $D(2n/5) + 4\lceil \log n \rceil + 1$ for the selector. □

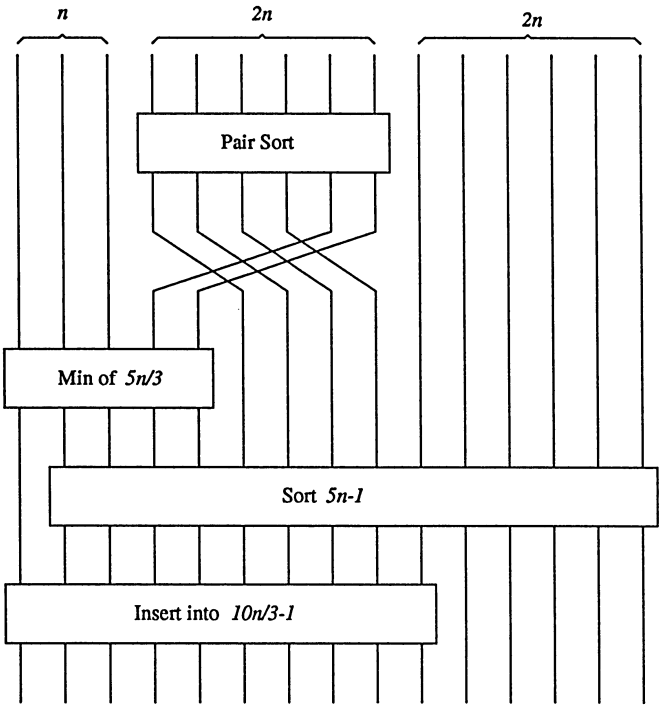


Figure 10: Details of the selector construction.

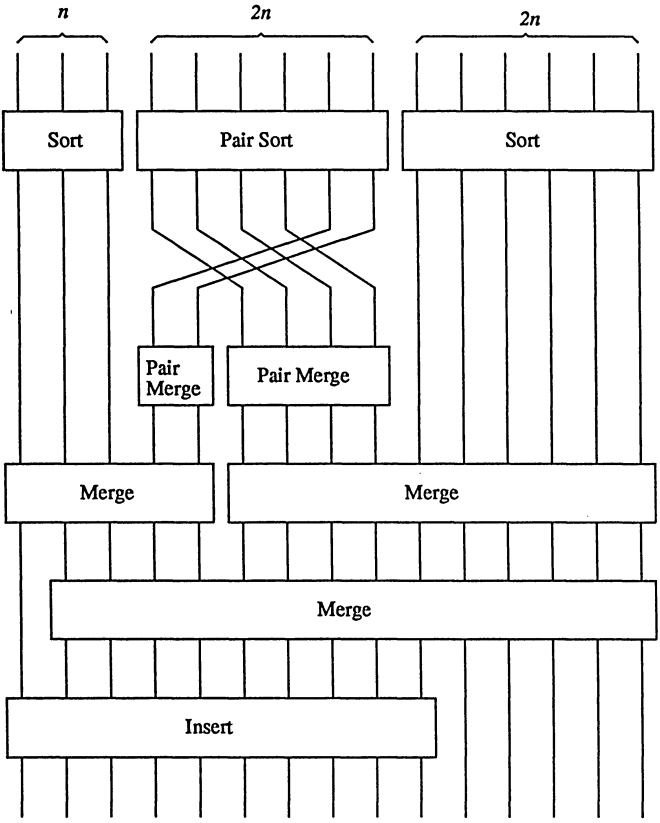


Figure 11: Details of the modified selector construction.

8 Conclusion and Open Problems

We have shown that sorting network verification is intractable even for sorting networks of depth $D(n)+4\lceil\log n\rceil+7$, where $D(n)$ is the depth of an optimal n-input sorting network. This is smaller by a factor of two than previous results. Our result is fairly strong, given the current state of knowledge about $D(n)$, which is that $\lceil\log n\rceil \leq D(n) < 6100\lceil\log n\rceil$ (the lower bound is obvious, and the upper bound is from Paterson [17]). We conjecture that sorting network verification remains intractable even for the shallowest sorting networks, that is, sorting networks of depth $D(n)$.

It is also interesting to consider the depth of single exception sorting networks, since their existence implies an exponential time lower bound for deterministic and probabilistic verification algorithms based on the zero-one principle (Parberry [16]). If $S(n)$ is the minimum depth of an n-input single exception sorting network, we know that $D(n)-1 \leq S(n) \leq D(n)+2\lceil\log n\rceil-1$, where $D(n)$ is the minimum depth of an n-input sorting network. We conjecture that $S(n)=D(n)$.

It should be noted that it is an open problem as to whether the result of Theorem 7.1 is better than that of Theorem 6.2. The former is better than the latter iff $D(n) \geq 2\lceil\log n\rceil-1$, which is the case for large enough n (Yao [20]).

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n.\log n)$ sorting network. *Proc. 15th Ann. ACM Symp. on Theory of Computing*, pages 1–9, April 1983.

[2] M. Ajtai, J. Komlós, and E. Szemerédi. Sorting in $c\log n$ parallel steps. *Combinatorica*, 3:1–48, 1983.

[3] K. E. Batcher. Sorting networks and their applications. In *Proc. AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, April 1968.

[4] R. C. Bose and R. J. Nelson. A sorting problem. *J. Assoc. Comput. Mach.*, 9:282–296, 1962.

[5] M. Chung and B. Ravikumar. On the size of test sets for sorting and related problems. In *Proc. 1987 International Conference on Parallel Processing.* Penn State Press, August 1987.

[6] M. J. Chung and B. Ravikumar. Strong nondeterministic Turing reduction — a technique for proving intractability. *J. Comput. System Sci.*, 39(1):2–20, 1989.

[7] R. L. Drysdale. Sorting networks which generalize batcher's odd-even merge. Honors Paper, Knox College, May 1973.

[8] R. W. Floyd and D. E. Knuth. Improved constructions for the Bose-Nelson sorting problem (preliminary report). *Notices of the AMS*, 14:283, 1967.

[9] R. W. Floyd and D. E. Knuth. The Bose-Nelson sorting problem. In J. N. Srivastava, editor, *A Survey of Combinatorial Theory*. North-Holland, 1973.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[11] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[12] I. Parberry. The alternating sorting network. Technical Report CS-87-26, Dept. of Computer Science, Penn. State Univ., September 1987.

[13] I. Parberry. *Parallel Complexity Theory*. Research Notes in Theoretical Computer Science. Pitman Publishing, London, 1987.

[14] I. Parberry. Sorting networks. Technical Report CS-88-08, Dept. of Computer Science, Penn. State Univ., March 1988.

[15] I. Parberry. A computer-assisted optimal depth lower bound for sorting networks with nine inputs. In *Proceedings of Supercomputing '89*, pages 152–161, 1989.

[16] I. Parberry. Single-exception sorting networks and the computational complexity of optimal sorting network verification. *Mathematical Systems Theory*, 23:81–93, 1990.

[17] M. S. Paterson. Improved sorting networks with $O(\log n)$ depth. *Algorithmica*, 5(4):75–92, 1990.

[18] T. J. Schaefer. The complexity of satisfiability problems. In *Proc. 10th Annual ACM Symposium on Theory of Computing*, pages 216–226. Association for Computing Machinery, 1978.

[19] D. C. Van Voorhis. An economical construction for sorting networks. In *Proc. AFIPS National Computer Conference*, volume 43, pages 921–926, 1974.

[20] A. Yao. Bounds on selection networks. *SIAM Journal on Computing*, 9, 1980.

Managing a Parallel Heap Efficiently

Sajal K. Das[1] and Wen-Bing Horng

Center for Research in Parallel and Distributed Computing
Department of Computer Science
University of North Texas
Denton, TX 76203-3886, USA

Abstract

We design a cost-optimal algorithm for managing a parallel heap on an exclusive-read and exclusive-write (EREW), parallel random access machine (PRAM) model. This is an improvement in space and time over the one recently proposed by Deo and Prasad [4]. Our approach efficiently employs p processors in the range $1 \leq p \leq n$, where n is the maximum number of items in a parallel heap. It is assumed that a *delete-think-insert cycle* is repeatedly performed, and each processor requires an arbitrary but the same amount of time (called the *think time*) for processing an item which in turn generates at most α (a constant) new items. The time required for deleting p items of the highest priority from the parallel heap is $O(1)$, while that for inserting at most αp new items is $O(\log n)$. With or without incorporating the think time, the speedup of our algorithm is proved to be linear, i.e. $O(p)$. Using a global, working data structure for each level of the heap, it is shown that the additional memory space required for our parallel heap is much less than that for the existing one [4]. Furthermore, the proposed algorithm retains the strict priority ordering of a sequential heap.

Index Terms: Algorithm analysis, data structure, EREW PRAM, heap, parallel algorithm, priority queue, optimal speedup.

1 Introduction

This paper concerning parallel heap data structure is an outcome of our research on parallel Branch-and-Bound (B&B) algorithms. In a B&B algorithm [6], a priority queue is often used to store the live nodes of a state space tree. Since the insertion and deletion operations on a heap can be performed efficiently, this data structure is used for implementing a priority queue. In the sequential version of a B&B algorithm, a *delete-think-insert cycle* (called an *iteration*) is repeatedly performed. At the beginning of each iteration, an item of the highest priority is deleted from the top of the heap and some processing (or thinking) is done on that item which possibly generates new items. These items are then inserted back into the heap. On the other hand, in a parallel B&B algorithm, p items (where p is the number of processors) of the highest priority

[1]This work was in part supported by a Junior Faculty Summer Research Fellowship from the University of North Texas at Denton.

are deleted at each iteration for the think phase to start. Therefore, managing heaps in parallel is an important problem.

Existing literature on parallel B&B algorithms mostly deal with upper bounds on the speedup or conditions for anomaly to occur [10-12, 15]. Although it is apparent that p items of the highest priority can be deleted from the priority queue in constant time, it is not clear how to efficiently insert the generated items. So various mechanisms have been proposed with an attempt to parallelize heaps for shared-memory parallel computers [2, 4, 7, 13, 14]. Table 1 summarizes the performance of these mechanisms on various machine models. For a comprehensive review, readers may refer to Horng and Das [5]. Quinn and Yoo [13] presented the *software pipelining* mechanism to delete an item from a heap in $O(1)$ time using $\lceil \log n \rceil$ processors, where n stands for the maximum number of items in the heap. Jones [7] proposed a *concurrent skew heap*, in which heap operations are performed in $O(\log n)$ amortized time. Biswas and Browne [2] also presented a concurrent data structure for heaps. Rao and Kumar [14] developed a practical mechanism by introducing a top-down insertion on a heap. Since at most $O(\log n)$ processors are efficiently utilized in these *concurrent heaps*, a maximum of $O(\log n)$ speedup is attainable. For increased parallelism, two other concurrent heaps — called *pipelined* and *partitioned heaps* — are proposed in [5].

Recently, Deo and Prasad [4] presented a new data structure, called the *parallel heap*, which achieves linear speedup for an n-item binary heap using p processors in the range $1 \leq p \leq n$. However, their approach suffers from several limitations as outlined here.

1. The available processors are partitioned into *general* and *maintenance processors*, which alternatively remain idle during the insert-delete and think phases.
2. At each delete-think-insert cycle, only r processors participate simultaneously in the think phase, where $r < p$ is the number of items in a heap-node.
3. The working memory space for managing the parallel heap is $O(n)$, a large value.
4. Linear speedup is not achieved if the think time is larger than $O(\log p)$.

In this paper, we propose an efficient algorithm for managing a parallel heap on an exclusive-read and exclusive-write (EREW), parallel random access machine (PRAM) model [8]. Our approach overcomes the limitations cited above, and includes improvement in time and space. A parallel heap of n items is represented as a complete k-ary (for $k \geq 2$) tree, each node of which contains x items, for $1 \leq x \leq p$. It is assumed that a delete-think-insert cycle is repeatedly performed, a processing item generates at most a constant number (α) of new items, and each processor requires the same amount of think time, which is arbitrary. Based on the number of processors available, the algorithm is presented into three distinct cases by applying the variants of a single technique. In each of these cases, p processors (for $1 \leq p \leq n$) are effectively utilized such that p items of the highest priority are deleted from the parallel heap in $O(1)$ time, and at most αp new items are inserted in $O(\log n)$ time. With or without incorporating the think time, a linear speedup is guaranteed, and thus the heap-management algorithm is cost-optimal. Furthermore, the strict priority ordering of a sequential heap is retained.

Unlike the algorithm due to Deo and Prasad [4], all p processors in our approach are treated as the general processors during the think phase; while during the insert-delete

Table 1: Summary of concurrent and parallel heap (priority queue) algorithms

Researchers	Model	# of Proc.	Insert_1	Delete_1	N	D	Insert_p	Delete_p	Type	Locking
Quinn, Yoo	MIMD-TC	$\lceil \log n \rceil$	–	$O(1)$	1	1	–	$O(p)$	Concurrent (Software Pipelining)	No
Biswas, Browne	MIMD-TC	$[1, n]$	$O(\log n)$	$O(\log n)$	1	1	$O(p + \log n)$	$O(p + \log n)$	Concurrent	Yes
Rao, Kumar	MIMD-TC	$[1, n]$	$O(\log n)$	$O(\log n)$	1	1	$O(p + \log n)$	$O(p + \log n)$	Concurrent	Yes
Jones	MIMD-TC	$[1, n]$	$O(\log n)$	$O(\log n)$	1	1	$O(p + \log n)$	$O(p + \log n)$	Concurrent (Skew Heap)	Yes
Horng, Das	MIMD-TC	p	$O(n/p + \log p)$	–	1	p	$O(p)$	$O(1)$	Concurrent (Pipelined Heap)	Yes
Horng, Das	MIMD-TC	$[1, n]$	$O(\log n)$	$O(\log n)$	1	p	$O(p + \log n)$	$O(p + \log n)$	Concurrent (Partitioned Heap)	Yes
Deo, Prasad	EREW PRAM	$[1, \log n]$	$O(\log n)$	$O(\log n)$	1	1	$O(p + \log n)$	$O(p + \log n)$	Concurrent (same as [14])	Yes
Deo, Prasad	EREW PRAM	$(\log n, n]$	$O(\log r)$	$O(1)$	r	r	$O(\log n)$	$O(1)$	Parallel	No
This paper	EREW PRAM	$[1, \log n)$	$O(\log n)$	$O(\log n)$	1	p	$O(\log n)$	$O(1)$	Parallel	No
This paper	EREW PRAM	$[\log n, \sqrt{n}]$	$O(\log n)$	$O(1)$	q	p	$O(\log n)$	$O(1)$	Parallel	No
This paper	EREW PRAM	$(\sqrt{n}, n]$	$O(\log n)$	$O(1)$	p	p	$O(\log n)$	$O(1)$	Parallel	No

[Note]: The number of items in a pipelined heap is $n \leq \lceil p^2 / \lceil \log(p+1) \rceil \rceil$.
MIMD-TC : multiple-instruction stream and multiple-data stream, tightly coupled multiprocessors
Insert_p : time required for inserting p items into the heap
Delete_p : time required for deleting p highest-priority items from the heap
N : number of items in a heap-node
D : number of items deleted at a time
Type : classification of the heap
Locking : whether employing semaphore or locking mechanism

phase, they act as the maintenance processors for maintaining the heap property at each level. The working memory space is reduced by exploiting the fact that at each level of a parallel heap, at most one heap-node is accessed at a time. Hence rather than allocating data fields to each node for inserted items or for book-keeping, we associate a global data structure with each level.

2 Preliminaries

A *heap* of n items can be represented by a complete k-ary (for $k \geq 2$) tree of depth $d = \lceil \log_k(n(k-1)+1) \rceil - 1$. The root is assumed to be at level 0. The *min-heap* (respectively, *max-heap*) *property* is one where the value of the item at any node is no greater (respectively, less) than those of the items at each of its children. In this paper, a heap always means a k-ary min-heap. For details on heap operations, refer to [2, 5].

A heap of maximum size n can be conveniently implemented by an array HEAP[0 .. $n-1$] such that the root (i.e., node 0) of the heap occupies location 0. The k children of node i in location i occupy consecutive locations $ki+1, \ldots, k(i+1)$, whereas the parent of node i is at location $\lceil \frac{i}{k} \rceil - 1$. Associated with a heap are the data fields LAST and TARGET. Let u be the current size of the heap. Then LAST $= (u-1)$ is the index of the last non-empty node of the heap, and TARGET $= u$ is the index of the first empty node of the heap. Nodes with index being equal to LAST and TARGET are called the *last node* and the *target node*, respectively. The path from the root of a heap to a target node is called the *insertion path* of the target node [14]. Suppose a target node, $TARGET$, is at level t. Let $FIRST(t) = (k^t - 1)/(k-1)$ be the index of the first (leftmost) node at level t, and let $DP = TARGET - FIRST(t) \geq 0$. Then, as shown in Horng and Das [5], the insertion path IP of the target node can be represented by the sequence of digits obtained by representing DP in radix k. That is, $DP = (IP)_k = (e_1 e_2 \cdots e_t)_k$.

An extremal case of insertion paths is that the root has its insertion path IP equal to $NULL$ with the length $|IP| = 0$. Figure 1 shows an example of 3-ary min-heap of twelve items with the insertion path from the root to the target node 12. In this figure a circle represents a node, the upper half of the circle contains the node number (or array index), the lower half contains the value of the item at that node, and the number outside the node is its path number. Here, we have $k = 3$, $d = 2$, $LAST = 11$, $TARGET = 12$, $FIRST = 4$. The path number of the target node is 022 while the insertion path $IP =$ 22. Note that $DP = TARGET - FIRST = 8 = (22)_3 = (IP)_k$.

3 Parallel Heap Algorithm

Analogous to a sequential heap, a *parallel heap* of n items, with each node containing x (for $1 \leq x \leq p$) items and thus having $m = \lceil \frac{n}{x} \rceil$ nodes, can be represented by a complete k-ary tree of depth $d = \lceil \log_k(m(k-1)+1) \rceil - 1$. The root is assumed to be at level 0, and thus the total number of levels is $d+1$. The *parallel heap property* is that the value of the items at any node of a parallel heap are no greater (or less) than those of the items at each of its children.

The parallel heap algorithm proposed in this paper can effectively utilize p processors

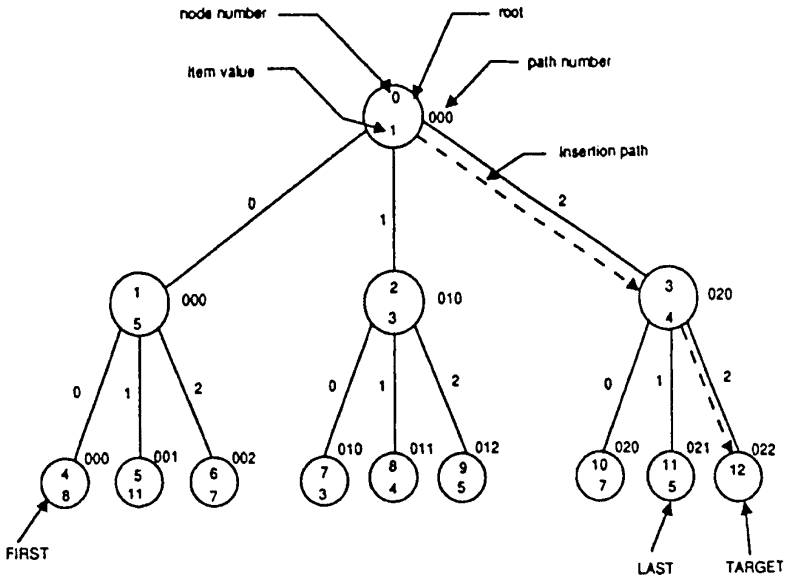


Figure 1: A 3-ary heap of 12 items

in the range $1 \leq p \leq n$ and achieves optimal speedup. The employed processors act both as the general and maintenance processors during different phases. For example, during the think phase, all p processors are treated as general processors to perform the thinking process, while during the insert-delete phase, they are considered as maintenance processors to maintain the heap property at each level. Each level of the heap is associated with $\lceil x/\log_k x \rceil$ processors for maintenance operations when $x \geq 2$. For the special case of $x = 1$, only one processor is associated with each level. Since the number of items in a node and the number of maintenance processors assigned to a level of a k-ary heap vary depending on the total number of processors available, we divide the algorithm into three distinct cases by applying appropriate variants of a single technique:

- Case I: $(nk - n)^{1/(1+\log k)} < p \leq n$, and $x = p$
- Case II: $\lceil \log_k(n(k-1)+1) \rceil \leq p \leq (nk - n)^{1/(1+\log k)}$, and $x = q$
- Case III: $1 \leq p < \lceil \log_k(n(k-1)+1) \rceil$ and $x = 1$,

where q is a function of n and p and $1 \leq q < p$. In Case III, since the number of processors is less than the number of levels, each of p levels of the heap is associated with only one maintenance processor at any pipeline cycle (described in Section 3.2).

3.1 Generic Parallel Heap Data Structures

The data structures maintained by a parallel heap consist of: (i) data structures for the heap itself and a heap-node, (ii) data structure for a level of the heap, and (iii) other

data fields. Figure 2 shows a layout of a parallel heap data structure with three levels. These data structures are elaborated in the following.

(i) Let each node of a parallel heap contains x items, where $1 \leq x \leq p$. Then the parallel heap of at most n items can be represented as an array PHEAP[0 .. $m-1$], where $m = \lceil \frac{n}{x} \rceil$ is the number of nodes, each of which is a record of the form:

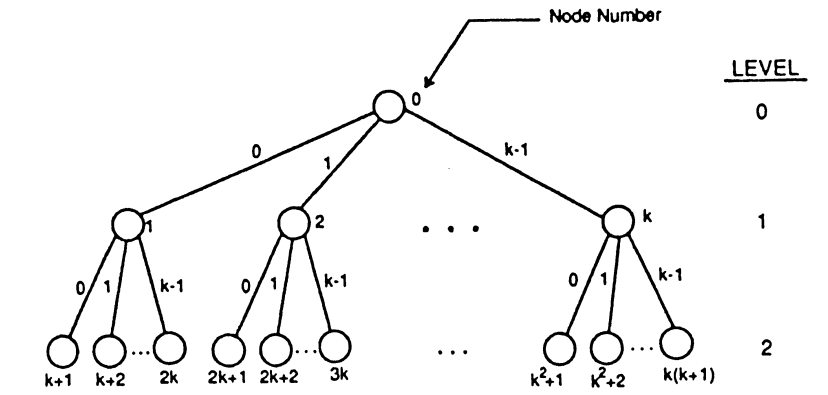
```
type NODE = record
          ITEM : array [0 .. x − 1] of item_type; /* items */
          #ITEMS : 0 .. x; /* currently */
          EXPECTED : 0 .. x; /* eventually */
     end_record;
var  PHEAP : array [0 .. m − 1] of NODE; /* parallel heap */
```

The array ITEM[0 .. $x-1$] stores items at a node, or the *substitute items* for an iteration of the delete-reheapification at that node. The field #ITEMS indicates how many items are currently in the node, and EXPECTED shows how many are eventually expected to be in it. An item which should be in the node but not yet available is called an *expected item*. In the following, we use #ITEMS(i) and EXPECTED(i) to denote the values of the fields #ITEMS and EXPECTED of node i, respectively, and use ITEM(j, i) to denote the value of the field ITEM[j] of node i.

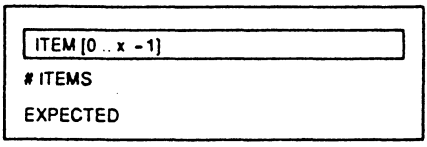
(ii) A data structure, called BLOCK, for each level of the heap is used for maintaining the parallel heap property. It is defined as follows.

```
type BLOCK = record
          WNODE : 0 .. m − 1; /* node to be processed */
          WTYPE : 0 .. α;
               /* 0 for delete-reheapification,
                  i for the ith insert-reheapification, where 1 ≤ i ≤ α */
          INSERT_ITEM : array [0 .. x − 1] of item_type; /* items to be inserted */
          LENGTH : 0 .. x − 1; /* number of inserted items */
          TARGET : 0 .. m − 1; /* index of the target node */
          INSERTION_PATH : 0 .. m − 1; /* insertion path */
     end_record;
var  LEVEL : array [0 .. d] of BLOCK; /* d is the depth of the parallel heap */
```

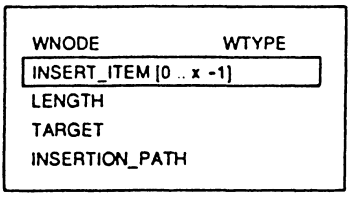
Like a *context block* in operating systems, the data structure BLOCK for each level of the heap stores all the required information for a node at that level which needs to perform an iteration of the insert- or delete-reheapification. The field WNODE is the index of the node to be processed at that level. If WNODE $= -1$, then no operations are performed at that level during that pipelined cycle. The field WTYPE indicates what kind of reheapification needs to be performed at node WNODE. If WTYPE $= 0$, then an iteration of delete-reheapification is performed. On the other hand, if WTYPE ≥ 1, an insert-reheapification process is performed. The array INSERT_ITEM[0 .. $x-1$] contains the items to be inserted during that insert-reheapification. LENGTH indicates how many items are to be inserted during that insert-reheapification, and TARGET is the index of the node where these items should be inserted eventually. INSERTION_PATH is a nonnegative integer representing the remaining path from WNODE to TARGET.



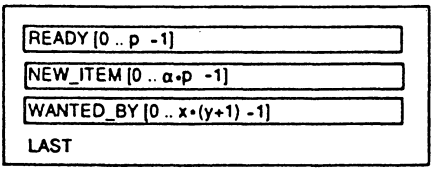
(a) A k-ary heap of three levels



(b) Data Structure for a node



(c) Data Structure for a level



(d) Other data fields

Figure 2: Data structures of a parallel heap

The array LEVEL contains $d+1$ data structures of type BLOCK such that level l of the heap is associated with LEVEL[l]. In the following, WNODE(l), WTYPE(l), LENGTH(l), TARGET(l), and INSERTION_PATH(l) denote the values of their corresponding data fields at level l, respectively. Also INSERT_ITEM(j, l) denotes the value of INSERTION_ITEM[j] at level l.

(iii) The remaining data fields associated with a parallel heap are three arrays READY, NEW_ITEM, and WANTED_BY, and an integer variable LAST. Assuming $y = \lceil \frac{p}{x} \rceil$, the declarations are:

```
var   READY : array [0 .. p − 1] of item_type;
      NEW_ITEM : array [0 .. α ∗ p − 1] of item_type;
      WANTED_BY : array [0 .. x ∗ (y + 1) − 1] of −1 .. p − 1;
      LAST : 0 .. m − 1; /* index of the last node */
```

The array READY contains p items (if the parallel heap is not empty) of the highest priority since the previous think phase is complete. The array NEW_ITEM is used to store at most αp new items produced by the processing items of the current think phase. Processors access the READY array simultaneously, each P_i reading the item READY[i], for $0 \leq i \leq p$, in constant time for the next think phase to process. After this think phase is complete, processor P_i puts its new generated items in consecutive locations $\alpha i, \alpha i+1, \cdots, \alpha(i+1)-1$ of the array NEW_ITEM. The field LAST is the index of the first node whose EXPECTED value is less than p. The array WANTED_BY is used by maintenance processors to check whether to move items from the bottom of the heap to the array NEW_ITEM. Initially, the field WANTED_BY[i], for $0 \leq i < x(y+1)$, is set to -1 to indicate that no movement of items is required. The array WANTED_BY is divided into $y+1$ subarrays such that the subarray WANTED_BY[$x(y-i)$ .. $x(y-i+1)-1$] is for node LAST$-i$, for $0 \leq i \leq y$. If the total number, say w, of new generated items after a think phase is less than p, then the additional $d_w = p - w$ items (called the *wanted items*) will be moved from the bottom of the heap to the subarray NEW_ITEM[w .. $p-1$] as follows. There are d_w processors allocated, one-to-one, to the last d_w items (including expected ones) in the parallel heap such that the last item (i.e., the item ITEM[EXPECTED-1] in node LAST) will be moved to the field NEW_ITEM[$p-1$], the last but one item to NEW_ITEM[$p-2$], and so on. If these wanted items are already in their target nodes, then all of them are moved to the subarray NEW_ITEM[w .. $p-1$] at the same time. If any one of these wanted items is still somewhere else along its insertion path, then its corresponding WANTED_BY fields will be set to the index of the subarray NEW_ITEM[w .. $p-1$] indicating where it should be moved to. The fields #ITEMS and EXPECTED of the nodes which move wanted items and/or is going to move expected items to NEW_ITEM array are updated appropriately. The data field LAST is updated accordingly. At the next iteration of the insert-reheapification, the maintenance processors at each level of the heap will first check the WANTED_BY array. If any element WANTED_BY[j] is set to value $g \neq -1$, then the corresponding wanted item at that level is moved to the NEW_ITEM[g], and WANTED_BY[j] is reset to -1.

3.2 Generic Parallel Heap Operations

The parallel heap algorithm presented in this paper is based on the framework due to Deo and Prasad [4]. In our approach, we also use the parallel prefix algorithm due to Ladner and Fischer [9] for packing the array NEW_ITEM, the optimal parallel mergesort due to Cole [3] for sorting new generated items, and the adaptive bitonic merging algorithm due to Bilardi and Nicolau [1] for maintaining the parallel heap property at each level of the heap. On an EREW PRAM model, each of the parallel prefix and bitonic merge algorithms on input of size $O(x)$ require $O(\log x)$ using $\lceil x/\log x \rceil$ processors; while Cole's mergesort has $O(\log x)$ time complexity for sorting $O(x)$ items employing x processors.

In order to maintain the parallel heap property, two generic operations — *delete-reheapification* and *insert-reheapification* — are performed. Since each iteration of the parallel reheapification is performed in a pipelined fashion, at most one node may violate the parallel heap property at each level. Such a state is called a *partial parallel heap*. In the following we briefly outline parallel reheapification procedures.

An *iteration* of the *parallel delete-reheapification* at level l is performed if BLOCK[l].WTYPE = 0. Let i be the value of BLOCK[l].WNODE. If node i satisfies the parallel heap property, the current reheapification terminates. Otherwise, merge the substitute items in PHEAP[i].ITEM and the items in its children, move the items in the merger to their appropriate places, and reset the data fields of the next lower level. A formal algorithm, called Parallel_Delete_Reheapification, for an iteration of the delete-reheapification is presented in [5], which uses a subprocedure Parallel_Bitonic_Merge.

An *iteration* of the *parallel insert-reheapification* at level l is performed if BLOCK[l].WTYPE $\geq$ 1. Let i be the value of BLOCK[l].WNODE. The procedure Parallel_Insert_Reheapification first moves the wanted items to the array NEW_ITEM, if necessary. Then, it merges these leftover inserted items and those in PHEAP[i].ITEM, moves items in the merger to their appropriate places, and resets the data fields of the next lower level. The details of this algorithm utilizing a subprocedure Parallel_Prefix for packing is presented in [5].

The maintenance processors at each level l of a partial parallel heap performs either an iteration of the insert-reheapification or delete-reheapification on node WNODE(l), according to the value of WTYPE(l). One such iteration is called a *pipeline cycle*, which is described as the following procedure.

```
procedure Pipeline_Cycle(l);
begin
    if (BLOCK[l].WTYPE = 0) then Parallel_Delete_Reheapification(l)
    else if (BLOCK[l].WTYPE ≥ 1) then Parallel_Insert_Reheapification(l)
end; /* Pipeline_Cycle */
```

Analyzing the subprocedures Parallel_Bitonic_Merge and Parallel_Prefix, an iteration of the parallel delete-reheapification (or insert-reheapification) as well as a pipeline cycle require $O(\log x)$ time. It is to be noted that a pipeline cycle defined in this paper is different from that in [4], where one iteration of the delete-reheapification and two iterations of the insert-reheapification are included in one pipeline cycle.

3.3 Case I: For p Processors, where $(nk-n)^{1/(1+\log k)} < p \leq n$

As mentioned earlier, we divide our algorithm into three distinct cases based on the number of processors available. Since we apply variants of the same technique for these three cases, for brevity, we present only Cases I and II in the following. Interested readers may find the third case in [5].

In Case I, $(nk-n)^{1/(1+\log k)} < p \leq n$, $x = p$, $m = \lceil \frac{n}{p} \rceil$, and $d = \lceil \log_k(m(k-1)+1) \rceil - 1$. Each level in the heap is associated with $\lceil p/\log p \rceil$ processors. The parallel heap algorithm divides a delete-think-insert cycle into two phases: *think phase* and *delete-insert phase* as discussed in the following.

THINK PHASE:

During the think phase, all the available p processors are switched to serve as general processors. Each processor P_i, for $0 \leq i < p$, performs the thinking process by accessing item READY[i]. After this think phase is complete, processor P_i generates at most α new items and places them in the subarray NEW_ITEM[αi .. $\alpha(i+1)-1$]. There are at most αp new items in the array NEW_ITEM.

INSERT-DELETE PHASE:

All p processors are switched back to serve as maintenance processors during this phase which includes the following five steps.

Step 1: *Packing new generated items*

Since some processors may generate less than α new items, the parallel prefix algorithm [9] is employed to pack these new items and to calculate the total number of such items produced by using the maintenance processors at level 0 (the root) of the heap. While packing new generated items, the maintenance processors at other levels perform one pipeline cycle at the same time.

Step 2: *Moving the wanted items to the array NEW_ITEM, if necessary*

After the new items are packed, the maintenance processors at level 0 check whether the total number of new generated items, say w, is less than p. If it is the case, all p processors are employed in moving $d_w = p - w$ items to the subarray NEW_ITEM[w .. $p-1$] if they are at the bottom of the heap, or set their WANTED_BY fields if they are still in their insertion paths. If any field of the array WANTED_BY is set, then one pipeline cycle is performed on the heap to move all the wanted items to the array NEW_ITEM, and w is updated to be the total number of items in NEW_ITEM.

Step 3: *Sorting the array NEW_ITEM*

If there are no items in NEW_ITEM, the algorithm terminates. Otherwise, the p available processors perform Cole's (optimal) parallel mergesort on NEW_ITEM.

Step 4: *Filling out the array READY*

After sorting the new items, all processors are switched back to serve as maintenance processors, and one pipeline cycle is issued at each level of the heap. Meanwhile, the maintenance processors at level 0 use bitonic merging algorithm to merge the items in the root and the first p items in NEW_ITEM if $w \geq p$ (or all the items in NEW_ITEM

if $w < p$). Then move the first p items of the merger to array READY and move the remaining ones to the root. Also both the fields WNODE and WTYPE of BLOCK[0] are set to 0 to indicate that heap-node 0 needs to perform an iteration of the delete-reheapification. After the current pipeline cycle is complete, one more pipeline cycle is issued on the heap so that, at this time the maintenance processors at level 0 perform an iteration of the delete-reheapification.

Step 5: *Inserting the remaining items in the array NEW_ITEM*

Let g be the number of remaining items in the array NEW_ITEM, where $0 \leq g \leq p(\alpha - 1)$. If $g = 0$, no further operation is needed since the heap has satisfied the partial parallel heap property at the root. Otherwise, let $d_p = p - \text{EXPECTED(LAST)}$. Then g can be expressed as $g = u + h \times p + v$, where $0 \leq h < \alpha - 1$, $0 \leq v < p$, and

$$u = \begin{cases} g & \text{if } g < d_p \\ d_p & \text{otherwise.} \end{cases}$$

This means that we need one (the first) iteration of insert-reheapification for inserting u new items, h (if $h \neq 0$) iterations for inserting p new items each, and one (the last, if $v \neq 0$) iteration for inserting v new items. Thus, at most α iterations of insert-reheapification process are required at the root. That is, at most α pipeline cycles will be issued on the heap. During each of these iterations, the maintenance processors at each level (other than level 0) perform their required operations, while the maintenance processors for level 0 will (i) set the field INSERTION_PATH according to the value LAST (as described in Section 2), (ii) increment the field LAST by one, (iii) update the fields WNODE and WTYPE to 0 and its corresponding iteration i, respectively, (iv) merge the items in the root and those in the array NEW_ITEM required by its corresponding iteration, and (v) follow the usual operation of an insert-reheapification process. After all such iterations are complete, the heap satisfies the partial parallel heap property and it starts the next think phase.

3.3.1 Time Complexity

Recall that one pipeline cycle requires $O(\log p)$ time in this case. Step 1 uses a parallel prefix algorithm to pack new items which requires $T_1 = O(\log p)$ time. In Step 2, the time required to move the wanted items from the bottom of the heap to the array NEW_ITEM and/or to set the fields WANTED_BY, and to perform one more pipeline cycle to move the wanted items to the array NEW_ITEM (if necessary) is given by $T_2 = O(1) + O(\log p)$. Step 3 performs parallel mergesort on the array NEW_ITEM in $T_3 = O(\log p)$ time. In Step 4, the time required by the root to merge the items in the subarray NEW_ITEM[0 .. $\beta - 1$] and those in the root, to move items to READY and the root, and to perform one pipeline cycle is $T_4 = 4 \times O(\log p)$. In Step 5, at most α pipeline cycles are issued for the root to perform an insert-reheapification process and the worst-case time is $T_5 = \alpha \times O(\log p)$ time. Therefore, the overall (parallel) time for an insert-delete phase to delete p items and to insert at most αp items is $T_{par} = T_1 + T_2 + T_3 + T_4 + T_5 = O(\log p)$, since α is a constant. Thus, the speedup in this case is $T_{seq}/T_{par} = O(p)$, where $T_{seq} = O(p \log n)$ is the time required for performing $O(p)$ sequential heap operations, and $\log p = O(\log n)$, for $(nk - n)^{1/(1+\log k)} < p \leq n$.

Assuming the time required for performing a think phase to be T_{think}, the total time

for a delete-think-insert cycle is $O(\log p) + T_{think}$. Obviously, the speedup for this case is also $O(p)$. Therefore, with or without incorporating the think time, our parallel heap algorithm for Case I achieves linear speedup (within a constant factor).

3.3.2 Space Complexity

The memory space required for the data structure PHEAP is $m(p+2) = \lceil \frac{n}{p} \rceil (p+2)$, that for LEVEL is $(d+1)(p+5) = \lceil \log_k(m(k-1)+1) \rceil (p+5)$, and that for other data fields is $(\alpha+2)p+1$. Thus, the additional memory space (other than that required for storing the heap) is approximately $2(n/p) + (p+5)\log_k(n/p) + (\alpha+3)p$. For the special case when $k = \alpha = 2$, the number p of processors can be represented as $p = n^\epsilon$, where $1/2 < \epsilon < 1$. Then the additional memory space in our algorithm is approximately $O(n^\epsilon \log n)$, whereas that required in [4] is $O(n)$. In general, since p is much less than n, it is clear that our algorithm can save much more additional memory space than that in [4].

3.4 Case II: For p Processors, where $\lceil \log_k(n(k-1)+1) \rceil \leq p \leq (nk-n)^{1/(1+\log k)}$

In this case, $x = q$, $m = \lceil \frac{n}{q} \rceil$, and $d = \lceil \log_k(m(k-1)+1) \rceil - 1$, where q is a function of n and p, and $1 \leq q < p$. Each level is associated with $\lceil q/\log q \rceil$ (or 1 if $q = 1$) processors. Thus the inequality $p \geq (d+1)\lceil q/\log q \rceil$ must hold, which yields $q \approx (p \log q)/(d+1)$. The algorithm for this case is similar to that in Case I except that there are at most $\lceil \frac{p}{q} \rceil$ iterations of the delete-reheapification for filling out the READY array and at most $\lceil (\alpha-1)p/q \rceil + 1$ iterations of the insert-reheapification for inserting the remaining items in the NEW_ITEM array to the heap. Since the think phase and Steps 1, 2, and 3 of the insert-delete phase are similar to those in Case I, in the following we describe only Steps 4 and 5 of the insert-delete phase.

INSERT-DELETE PHASE:

Step 4: *Filling out the array READY*

Let $m_w = \min(p, w)$ and $m_w = u \times q + v$, where $0 \leq u < \lceil m_w/q \rceil$ and $0 \leq v < q$. This indicates that there are u iterations of moving q items and one (the last, if $v \neq 0$) iteration of moving v items from NEW_ITEM to READY. Each of these iterations is performed as follows. At iteration i, for $1 \leq i \leq u$, the maintenance processors at the root use bitonic merging algorithm to merge the items in the root and those in the subarray NEW_ITEM$[(i-1)q \,..\, iq-1]$. At the same time, one pipeline cycle is performed at other levels of the heap. Then, move the first q items out of the merger to READY$[(i-1)q \,..\, iq-1]$, and move the remaining items to the root. Also, update WNODE and WTYPE of level 0 to zero to indicate that node 0 needs to perform an iteration of the delete-reheapification. After the current pipeline cycle is complete, one more such cycle is issued to the heap so that the root performs the delete-reheapification at this time. If $v \neq 0$, the $(u+1)$-st iteration is performed as follows. While the maintenance processors at other levels of the heap perform one pipeline cycle, the v items in NEW_ITEM$[uq \,..\, m_w-1]$ are merged with those in the root. Out of the merger, the first v items are placed

into the subarray READY[uq .. $m_w - 1$], the remaining items are moved to the root, and WNODE and WTYPE of level 0 are set to zero. After the current pipeline cycle is complete, one more pipeline cycle is issued to the heap as in the previous iteration.

Step 5: *Inserting the remaining items in the array NEW_ITEM*

If $w \leq p$, no insert-reheapification is needed since at this point, no items are left in NEW_ITEM. Otherwise, let $g = w - p$ be the number of remaining items in NEW_ITEM, where $0 \leq g \leq (\alpha - 1)p$. Denoting $u = q - \text{EXPECTED(LAST)}$, we get $g = u + h \times q + v$, where $0 < u \leq q$, $0 \leq h < \lceil (\alpha - 1)p/q \rceil$, and $0 \leq v < q$. Then follow the same method as in Case I to perform at most $h + 2$ iterations of the insert-reheapification and start the next think phase.

3.4.1 Time Complexity

In this case, one pipeline cycle requires $O(\log q)$ time. Therefore, the time required for each of Steps 1 and 3 is $O(\log p)$, and that required for Step 2 is $O(\log q)$. Step 4 requires $2(u+1) \times O(\log q) = O(p/q) \times O(\log q) = O(\log n)$ time. Step 5 performs at most $h + 2$ pipeline cycles and it requires $(h + 2) \times O(\log q) = O(p/q) \times O(\log q) = O(\log n)$ time. Therefore, the overall (parallel) time for an insert-delete phase to delete p items and to insert at most αp items is $O(\log n)$. Following the same approach as in Case I, with or without incorporating the think time, the speedup for this range of processors is still $O(p)$.

3.4.2 Space Complexity

The memory space required for data structure PHEAP is $m(q + 2) = \lceil \frac{n}{q} \rceil (q + 2)$, that for LEVEL is $(d + 1)(q + 5) = \lceil \log_k (m(k - 1) + 1) \rceil (q + 5)$, and that for other data fields is $(\alpha + 2)p + 1$. Therefore, the additional memory space is approximately $2(n/q) + (q + 5) \log_k (n/q) + q + (\alpha + 2)p$. For the special case when $k = \alpha = 2$, the number of processors p is in the range $\lceil \log(n + 1) \rceil \leq p \leq \sqrt{n}$. Then the additional memory space in our algorithm is $O((n/q) + q \log(n/q) + p)$. Clearly, on an average, our algorithm for this range of processors saves much more additional memory space than that required in [4], which is always $O(n)$.

3.4.3 Calculation of q

As mentioned earlier, each node of the heap in Case II contains q items, where q is a function of n and p. According to Section 3.4, the following inequality holds:

$$p \geq \lceil q / \log q \rceil \times \lceil \log_k (m(k - 1) + 1) \rceil,$$

from which an approximate equation can be derived:

$$q \approx p \log q / \lceil \log_k (m(k - 1) + 1) \rceil.$$

Now the approximate value of q can be calculated by the following algorithm.

Table 2: Performance of parallel binary heaps

Researchers	Das & Horng		Deo & Prasad	
Complexity	Space	Time	Space	Time
Case I	$O(n^\epsilon \log n)$, $\frac{1}{2} < \epsilon \leq 1$	$O(T + \log n)$	$O(n)$	$O(\frac{p}{r}(T + \log r))$
Case II	$O(p + \frac{n}{q} + q \log \frac{n}{q})$	$O(T + \log n)$	$O(n)$	$O(\frac{p}{r}(T + \log r))$
Case III	$O(\log n)$	$O(T + \log n)$	$O(n)$	$O(T + \log n)$

Algorithm: *Computing the value of* q
Input: n, p, and k.
Output: q.

1. $q' \leftarrow k$;
   repeat $q \leftarrow q'$; $q' \leftarrow p \log q / \lceil \log_k (m(k-1)+1) \rceil$ **until** $|q' - q| < 1$;
   $q \leftarrow \lceil q' \rceil$;
2. $m = \lceil n/q \rceil$; $l \leftarrow \lceil \log_k (m(k-1)+1) \rceil$; $s \leftarrow \lceil q / \log q \rceil$;
   if $s \times l \leq p$ **then return**(q);
3. $s \leftarrow \lceil p/l \rceil$;
   repeat $q \leftarrow q'$; $q' \leftarrow s \log q$ **until** $|q' - q| < 1$;
   $q \leftarrow \lceil q' - 1 \rceil$;
   goto Step 2.

3.5 Case III: For p Processors, where $1 \leq p < \lceil \log_k (n(k-1)+1) \rceil$

For the number of processors in this range, each node of the parallel heap contains only one item. Consequently, $m = n$, and $d = \lceil \log_k (n(k-1)+1) \rceil - 1$. Also, the data structures of the parallel heap and an iteration of the insert- or delete-reheapification can be simplified. Since $p \leq d$, instead of statically allocating a certain number of processors to each level as in the previous cases, each of p processors is dynamically assigned to a level of the parallel heap as the maintenance processor which performs the insert- or delete-reheapification. Due to space limitation, the details are omitted here and readers may refer to [5]. The performance of our algorithm for this case are summarized below. One pipeline cycle requires $O(1)$ time, and a linear speedup is achieved with or without incorporating the think time. The additional memory space required is approximately $5 \log_k n + (\alpha + 2)p$. For the special case when $k = \alpha = 2$, p is in the range $1 \leq p < \lceil \log(n+1) \rceil$ and the additional space required by our algorithm is $5 \log n + 4p = O(\log n)$ while that space in [4] or [14] is $2n + \log n = O(n)$.

Table 2 provides a comparison of a special case of our algorithm on binary heaps (i.e., for $k = \alpha = 2$) with the approach due to Deo and Prasad. In this table, T stands for T_{think}, the quantity q is the number of items in a node for Case II of our algorithm, and r is the number of items in a node in Cases I and II of [4].

4 Proofs of Correctness

It can be shown that our parallel heap algorithm is deadlock-free, starvation-free, and provides mutual exclusion on each heap-node [5]. The correctness of the proposed algorithm is presented in Theorem 1. Before proving it, we establish four lemmas which can be proved from the generic parallel heap operations. As mentioned, a partial parallel heap stands for a k-ary partial heap with each node containing x items, where $1 \leq x \leq p$.

Lemma 1: For a partial parallel heap, the items in each node of the heap and those in INSERT_ITEM at each level are in sorted order.

Lemma 2: For a partial parallel heap, after an iteration of the insert-reheapification on a node i at level l, node i still satisfies the parallel heap property.

Lemma 3: For a partial parallel heap, after an iteration of an iteration of the delete-reheapification on a node i is complete, the child which is not a leaf node and has the smaller last item among the children of node i may violate the parallel heap property, while nodes i and the other children still satisfy this property.

Lemma 4: For a partial parallel heap, only those nodes which need to perform the delete-reheapification may violate the parallel heap property.

Theorem 1: After each insert-delete phase in a partial parallel heap, the values of the items in the root are less than or equal to those of the items in its descendants, and the items in the array READY are in sorted order with the values of these items being less than or equal to those of the items in the root of the heap.
Proof (by induction): The induction parameter s in this theorem is the iteration number of delete-think-insert cycle applied so far.

Induction Basis: At the beginning of the first insert-delete phase (i.e., $s = 1$), only one item is in the heap. After this phase is complete, the item is moved to the array READY and the heap becomes empty. Clearly, the theorem holds for this case.

Induction Hypothesis: Assume that at the end of i-th iteration (i.e., $s = i$) of delete-think-insert cycle, the theorem holds true.

Induction Step: Now, let us consider the beginning of the $(i+1)$-th iteration of insert-delete phase. By definition, before an iteration of the delete-reheapification is introduced, the first x items of the smallest values from the merger of items in array NEW_ITEM and those in the root, are moved to array READY. Thus, after performing all delete-reheapification iterations in an insert-delete phase on the root, the items in READY are in sorted order; and their values are no more than those of the items in the heap and the remaining ones in NEW_ITEM. Essentially, we have a partial parallel heap with the root-node satisfying the parallel heap property, thereby implying that the items in the root contain the smallest values of the entire heap. □

Example: Let us illustrate the performance of our algorithm and compare it with Deo-and-Prasad's algorithm for $n = 2^{32}$ and $p = 2^{13}$. Note that the given values of n and p correspond to Case II of our approach. The results are given in Table 3.

We observe the following facts from Table 3: (i) the number of items in a node in our algorithm is almost 1.7 times larger than that in a node in the algorithm due to Deo and Prasad, and (ii) in our algorithm 8180 processors are allocated as the maintenance processors during the insert-delete phase and all of 8192 processors are switched back

Table 3: Comparison of two algorithms when $n = 2^{32}$ and $p = 2^{13}$

Algorithm	#items per node	#nodes in heap	#levels of heap	#processors per level	#maintenance processors	#general processors
Das and Horng	5029	854041	20	409	8180	8192
Deo and Prasad	2951	1455480	21	257	5397	2951

to serve as the general processors during the think phase. However, in the algorithm presented in [4], only 5397 processors are dedicated to serve as the maintenance processors and 2951 processors act as the general processors. Therefore, approximately 66% and 34% of the available processors are idle during the think phase and insert-delete phase, respectively.

5 Concluding Remarks

The contribution of this paper is to develop a cost-optimal algorithm for managing a parallel heap, which is also an improvement over the parallel (binary) heap proposed in [4], for an EREW PRAM model. Our algorithm efficiently employs p processors in the range $1 \leq p \leq n$. We represent a parallel heap as a complete k-ary tree with each node containing x items (for $1 \leq x \leq p$) depending on the number of processors available. A delete-think-insert cycle is repeatedly performed, assuming that the think time is identical for each processor. Also, it is assumed that each processing item generates at most α (a constant) new items. The proposed algorithm is divided into three distinct cases, each of which utilizes an appropriate variant of a single strategy. The p processors are efficiently utilized such that the time required for deleting p items with the highest priority from the heap is $O(1)$, while that for inserting at most αp new items is $O(\log n)$. Though our algorithm originates from the idea developed by Deo and Prasad [4], the design strategy and data structures differ in several ways. Some of the salient features and major advantages of our approach are highlighted below.

1. At each delete-think-insert cycle, p items are deleted for the think phase to start.
2. Rather than partitioning p processors into two disjoint sets for distinct usages, namely *general* and *maintenance* processors for the think and insert-delete phases, we use all of them efficiently for both purposes.
3. Unlike in [4], where all the working data structures are stored in each heap-node, we propose a global data structure associated with each level of the heap. This leads to a reduction of extra memory space for implementing our algorithm.
4. Variants of the same algorithmic strategy are applied for three different ranges of available processors between 1 and n, whereas the authors in [4] apply the mechanism proposed in [14] for the number of processors in the range $1 \leq p \leq \log n$.
5. With or without incorporating arbitrary think time, our algorithm is proved to achieve optimal speedup.

6. The number of items stored in a heap-node is greater than or equal to that in [4], and it depends on the available range of processors.

As part of future work, we intend to implement the parallel heap algorithm on commercial shared memory parallel computers, in order to verify how closely its actual performance follows the theoretical analyses developed here. Since this algorithm allows simultaneous deletion of p items from the heap, we are currently investigating how to apply it for estimating the speedup of parallel algorithms based on the branch-and-bound strategy. Our preliminary observations are encouraging, which will be reported in a forthcoming paper.

References

[1] G. Bilardi and A. Nicolau, "Adaptive Bitonic Sorting: An Optimal Algorithm for Shared-Memory Machines," *SIAM J. Comput.*, vol. 18, no. 2, pp. 216-228, Apr. 1989.

[2] J. Biswas and J. C. Browne, "Simultaneous Update of Priority Structures," *Proc. Int. Conf. Parallel Process.*, 1987, pp. 124-131.

[3] R. Cole, "Parallel Merge Sort," *SIAM J. Comput.*, vol. 17, no. 4, pp. 770-785, Aug. 1988.

[4] N. Deo and S. Prasad, "Parallel Heap," *Proc. Int. Conf. Parallel Process.*, vol. III, 1990, pp. 169-172.

[5] W.-B. Horng and S. K. Das, *Heaps — Concurrency and Parallelism*, Tech. Rep. #N-90-003, Dept. Comput. Sci., Univ. North Texas, Denton, Mar. 1990.

[6] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Rockville, Maryland, 1978.

[7] D. W. Jones, "Concurrent Operations on Priority Queues," *Commun. ACM*, vol. 32, no. 1, pp. 132-137, Jan. 1989.

[8] R. M. Karp and V. Ramachandran, "Parallel Algorithms for Shared-Memory Machines," in *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity* (J. van Leeuwen, Ed.), MIT Press, Cambridge, MA, 1990, pp. 869-941.

[9] R. E. Ladner and M. J. Fischer, "Parallel Prefix Computation," *J. ACM*, vol. 27, no. 4, pp. 831-838, Oct. 1980.

[10] T.-W. Lai and S. Sahni, "Anomalies in Parallel Branch-and-Bound Algorithms," *Commun. ACM*, vol. 27, no. 6, pp. 594-602, June 1984.

[11] T.-W. Lai and A. Sprague, "Performance of Parallel Branch-and-Bound Algorithms," *IEEE Trans. Comput.*, vol. C-34, no. 10, pp. 962-964, Oct. 1985.

[12] G.-J. Li and B. W. Wah, "Coping with Anomalies in Parallel Branch-and-Bound Algorithms," *IEEE Trans. Comput.*, vol. C-35, no. 6, pp. 568-573, June 1986.

[13] M. J. Quinn and Y. B. Yoo, "Data Structures for the Efficient Solution of Graph Theoretic Problems on Tightly-Coupled MIMD Computers," *Proc. Int. Conf. Parallel Process.*, 1984, pp. 431-438.

[14] V. N. Rao and V. Kumar, "Concurrent Access of Priority Queues," *IEEE Trans. Comput.*, vol. 37, no. 12, pp. 1657-1665, Dec. 1988.

[15] B. Wah and Y. Ma, "MANIP — A Parallel Computer System for Implementing Branch-and-Bound Algorithm," *Proc. 8th Annu. Symp. Comput. Archi.*, 1982, pp. 239-262.

Parallel complexity in the design and analysis of concurrent systems

Carme Àlvarez* José L. Balcázar* Joaquim Gabarró* Miklós Sántha+

Dep. de Llenguatges i Sistemes Informatics
Universitat Politècnica de Catalunya
Pau Gargallo 5, 08028 Barcelona
Spain

CNRS - LRI
Université Paris-Sud
91405 Orsay
France

Keywords: Petri nets; partially commutative monoids; CCS; PRAM algorithms; boolean circuits; P-completeness.

Abstract: We study the parallel complexity of three problems on concurrency: decision of firing sequences for Petri nets, trace equivalence for partially commutative monoids, and strong bisimilarity in finite transition systems. We show that the first two problems can be efficiently parallelized, allowing logarithmic time Parallel RAM algorithms and even constant time unbounded fan-in circuits with threshold gates. However, lower bounds imply that they cannot be solved in constant time by a PRAM algorithm. On the other hand, strong bisimilarity in finite labelled transition systems can be classified as P-complete; as a consequence, algorithms for automated analysis of finite state systems based on bisimulation seem to be inherently sequential in the following sense: the design of an efficient parallel algorithm to solve any of these problems will require an exceedingly hard algorithmic breakthrough.

1. Introduction

Given the intrinsic difficulty of designing large software systems, it is natural that software tools would be designed to help in performing this task. The possibility of formalizing both specifications and implementations in the same, or in a closely related, formal language yields the potential of automated analysis, allowing for early checking of correctness and provably correct prototypes.

The design of correct concurrent programs is even more difficult than in the sequential case, and their verification using formal systems may give rise to formidable computational problems. For instance, the study of the correctness and liveness properties of mutual exclusion algorithms for just two processes already requires resorting to computerized analysis [27]; if more processes are considered, the state space soon becomes intractable.

One reason to develop concurrent programs stems from the fact that important advantages can be gained from the use of massive parallelism. In view of the large number of

* Research supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 3075 (project ALCOM).

+ Research supported by the Programme MERCURE of the DCSTD of the Ministère Français des Affaires Etrangères and the DGICYT of the Ministerio de Educación y Ciencia de España. This research was performed while visiting the Dep. de Llenguatges i Sistemes Informàtics of the Universitat Politècnica de Catalunya.

parallel algorithms discovered in recent years (see [10] and [16]), it might be hoped that one such application would be the study of concurrent systems, and that algorithms running on highly parallel machines could perform automated analysis of large concurrent programs substantially faster than sequential algorithms. Such a behaviour corresponds to a running time roughly logarithmic in the size of the state space (assumed finite); and being able to tackle problems of relevant size corresponds to algorithms that use a large but feasible number of processors (cf. the definition of the class NC below).

One of the first models issued to study concurrent systems was the Petri net model. A Petri net consists of two different kinds of objects: places and transitions. Places serve to model pre and post conditions and transitions model events. A transition needs to satisfy some conditions to be fired, and its firing changes the valuations on the places (see below for exact definitions). The net evolves firing transitions sequentially and the behaviour of the whole system is described by the set of all possible firing sequences.

For each fixed Petri net, we exhibit an NC algorithm to decide very efficiently in parallel whether a given sequence of transitions is a firing sequence. We also discuss some lower bounds on the parallel time necessary to solve this problem.

In a monoprocessor environment, a concurrent system is fully described by the set of all sequential evolutions. A possible evolution of the system is described by a finite word called trace. Let us consider with more attention the sequencing of two events x and y in a trace $w = \cdots xy \cdots$. Let $w' = \cdots yx \cdots$ be the trace obtained from w by commuting the order of the events x and y. We have two different possibilities:

- The events x, y are independent from each other. In this case the order of execution is irrelevant and x, y commute. Then the traces w and w' correspond to the same parallel behaviour.
- The execution of x modifies the environment of y. Then these events are in conflict. The trace w' represents a behaviour different from w.

A basic question is: given two traces, do they model the same concurrent behaviour? A way to deal with this approach is to consider partially commutative monoids. This framework has been fully developed by Mazurkiewicz [18]. A mathematical characterization of trace equivalence was found in [8], and it can be used to find a fast sequential algorithm. Here we prove the existence of an NC algorithm, and discuss also some lower bounds.

A capability that seems natural to expect from software tools for aiding the design of concurrent systems is to be able to decide some form of equivalence of finite state systems. Indeed, this problem plays a fundamental role in the study of concurrent systems, and has been widely studied both from theoretical and practical points of view. Milner specifies in [21] a complete set of axioms for proving equivalence of finite state agents. Kanellakis and Smolka consider in [15] efficient sequential algorithms to solve this problem. On the more practical side, the prototype named Concurrency Workbench, implemented in Standard ML, has been used by Walker [27] for undertaking the automated analysis of mutual exclusion algorithms via finite state systems, using the fact that the state space of all these algorithms is finite.

Until now, the analysis of concurrent systems by means of bisimulation techniques has been based on sequential algorithms. A natural question to ask is: do the automatic bisimulation techniques admit fast parallel algorithms?

In this paper we give a strong evidence that unfortunately the answer to the above question is negative. More precisely, we prove that deciding bisimulation in finite transition systems is a P-complete problem. P-complete problems have efficient sequential algorithms but it is widely believed that they do not admit fast parallel ones.

In fact, this concept plays a role analogous to the notion of NP-complete problems. These are problems that can be solved by an exponentially slow exhaustive search, and they inherently seem to require superpolynomial time algorithms. The NP-completeness of a problem implies that success in designing a polynomial time sequential algorithm for it is highly unlikely.

Analogously, P-complete problems are identified as inherently sequential problems: if there are any problems in P that do not admit efficient parallel algorithms, then all P-complete problems are among them. Conversely stated, if a parallel algorithm is found for a P-complete problem which uses a feasible (e.g. polynomial) amount of hardware and runs in polylogarithmic time, then all problems solvable in polynomial time have also such feasible and very fast parallel algorithms. However, strong research in the area during several years has failed to produce such an algorithm for any of the well studied P-complete problems. Thus, the design of a parallel algorithm with these characteristics for a P-complete problem would require a breakthrough in Algorithmics. Actually the conjecture of many researchers in the field is that such an algorithm does not exist at all. Surveys of P-complete problems have appeared in [13] and [22].

2. Preliminaries

2.1 Sequential and parallel complexity classes. For the formal study of the possible existence of parallel algorithms we will consider two main complexity classes: P and NC. We will mention also some interesting subclasses of NC. The class P models problems with *efficient sequential algorithms*; the class NC models problems with *fast parallel algorithms*, using a feasible number of processors. Each of these classes has many characterizations that support this description.

- By definition, the class P contains the problems for which a polynomial time sequential algorithm exists. This can be formalized by considering an abstract model of sequential computation for which "time" is a well-defined notion. Polynomial time RAM algorithms (a model quite close to a real computer [1]), polynomial time Turing machines ([1], [4]), or even polynomial size uniform circuits (see below) are all suitable for this purpose, and give equivalent definitions of the class P.

- The class NC formalizes the concept of efficiently parallelizable problems: it contains those problems for which a parallel algorithm can be designed that runs in polylogarithmic parallel time and uses a feasible (i.e. polynomial) amount of hardware. There are many characterizations of this class. Consider for instance Parallel RAM (PRAM) machines, which are one of the basic abstract models of parallel computers [26]. NC can be defined as the class of all the problems that can be solved in a PRAM within $O(\log^k n)$ time for constant k and using polynomially many processors.

For theoretical analysis sometimes *unbounded fan-in boolean circuits* are preferable [7]. A boolean circuit is a directed, acyclic, labelled graph in which the nodes of indegree zero are the inputs, the nodes of indegree 1 compute boolean negation, and the nodes of indegree 2 or more compute either boolean conjunction or disjunction of all their inputs, according to their respective label. The nodes of outdegree zero are the output nodes. The *size* of a circuit is the number of its nodes; the *depth* is the length of the longest path from an input to an output. The nodes in a circuit are called also *gates*. A boolean circuit computes a boolean function by substituting values for the inputs, evaluating all the nodes, and collecting values at the output nodes. Binary inputs and outputs might be binary encodings of other objects assuming some simple coding scheme.

To use boolean circuits to solve problems, we have to select a different circuit for each input length; but such a selection might be very hard to compute. Here we will explicitly rule out those families of circuits for which this selection is indeed hard, and will restrict ourselves to uniform families. A family of circuits is *uniform* if basic facts about the connection of the gates can be answered in deterministic logarithmic time, or equivalently can be expressed in an extended version of first order logic (see [6] for precise definitions).

It is well known that in many aspects PRAMs and uniform unbounded fan-in circuits are equivalent [26], with bounds on number of processors corresponding to bounds on the size of the circuit, and bounds on the PRAM time corresponding to bounds on the depth of the circuit. Thus NC is formed by the problems solvable by polylogarithmic depth, polynomial size uniform circuits.

NC has some interesting subclasses. In particular, AC^0 contains the problems solvable by unbounded fan-in uniform circuits of constant depth and polynomial size, or equivalently solvable by a PRAM in constant time with a feasible (i.e. polynomial) number of processors; and AC^1 contains the problems solved by unbounded fan-in uniform circuits of logarithmic depth and polynomial size, which corresponds to logarithmic time in a PRAM with again a feasible number of processors. AC^0 contains some problems with long history, for instance the addition of two integer numbers.

Lying between AC^0 and AC^1 is the class TC^0, defined by uniform constant depth polynomial size circuits that are allowed to use threshold gates. This class can be motivated by the growing of a complexity theory of neural networks [23], and is important for tight analysis of the complexity of certain problems; it also contains very natural and interesting problems such as the multiplication of two integer numbers [7].

Since threshold gates can simulate AND and OR gates we have that $AC^0 \subset TC^0$, but these two classes do not coincide: Ajtai [2] and Furst, Saxe and Sipser [9] proved that the inclusion was strict. This was shown by proving that the majority problem, coded as the set $MAJ = \{w \in (0+1)^* \mid |w|_1 \geq |w|_0\}$, cannot be solved by a constant depth polynomial size circuit having only AND and OR gates. The proof does not require any uniformity condition on the AC^0 circuits.

To compare and classify problems in P we use the *constant depth reducibility* [7]. A function f is constant depth reducible to g, denoted here as $f \leq_{cd} g$, if there is a family of circuits which compute f with polynomial size, constant depth, and oracle gates for g. The cost and depth of an oracle gate is 1. It can be easily shown that AC^0, TC^0,

AC^1, and AC are closed under this reducibility; e.g., if $g \in TC^0$ and $f \leq_{cd} g$, then $f \in TC^0$.

A problem S is *P-complete* under $\leq_{cd}$ reductions if $S \in P$ and every problem in P is $\leq_{cd}$-reducible to S. It can be shown that this reducibility is transitive, and therefore to prove that a problem in P is P-complete, it is enough to prove that some other complete problem in P is reducible to it. There are several standard P-complete problems which are natural candidates for the reduction. One of these is the Circuit Value Problem CVP. The input to this problem is pair formed by a circuit and an input to the circuit. The problem consists of computing the output of the circuit on the given input. When suitable additional hypotheses are assumed on the given circuit, we obtain variants of this problem that still are P-complete. In order to prove our results we consider one of these variants: the evaluation problem for *monotone alternating circuits*. Such a circuit is divided into levels, so that the inputs to a gate at a given level are all outputs of gates from the immediately preceeding level. The circuit does not contain negation gates; instead it receives each input together with its negation. All gates in the same level are of the same type, and the levels alternate between AND and OR levels. Figure 1 gives us an example of a monotone alternating circuit.

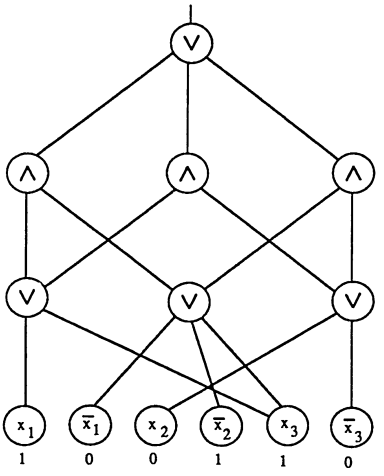


Fig. 1 A monotone alternating boolean circuit C

The following is known [13]:

Theorem 1: The Monotone Alternating Circuit Value Problem $MACVP$ is P-complete.

Input: An encoding of a monotone alternating circuit c with one output, together with boolean input values $x_1, \overline{x}_1, \ldots, x_n, \overline{x}_n$.

Output: The value of c on these input values.

Let us end here our complexity-theoretic notions and go on to introduce the problems whose complexity will be classified. The notations introduced here will be necessary for later description of parallel algorithms.

2.2 Petri nets. The Petri net model was one of the first models introduced to describe concurrent processes with distributed control [25]. Formally a Petri net is a tuple $N = \langle P, T, F, M_0 \rangle$ where:

1. The set $P = \{p_1, \ldots, p_r\}$ is called the *set of places*. During the evolution of the net, a place p contains a number of tokens denoted as $M(p)$ and called its marking. Such a marking models some local aspect of the system with global state $M = (M(p_1), \ldots M(p_r))$.
2. The *set of transitions* is $T = \{t_1, \ldots, t_s\}$. Transitions model the events of N and every sequential behaviour is represented by a word $w \in T^*$.
3. The *flow function* $F : \{(P \times T) \cup (T \times P)\} \rightarrow \mathbb{N}$ connects between them places and transitions. The value of F fixes the precondition to be fulfilled in order to fire a transition t in a marking M. The *firing rule* is:

   $$\forall p \in P : M(p) \geq F(p,t).$$

   Additionally, F gives us the new marking M' reached after the firing of t in M, denoted as $M[t\rangle M'$, and defined by:

   $$\forall p \in P : M'(p) = M(p) - F(p,t) + F(t,p).$$

4. $M_0 : P \rightarrow \mathbb{N}$ is the initial marking.

We denote by $\Delta(p,t)$ the variation on the number of tokens in a place p when t is fired, $\Delta(p,t) = F(t,p) - F(p,t)$. Then $M'(p) = M(p) + \Delta(p,t)$. The firing rule can be extended from transitions to words $w \in T^*$ as usual and the whole sequential behaviour of the net N is described by the set of *firing sequences* which is:

$$S_N(M_0) = \{w \in T^* \mid \exists M : M_0[w\rangle M\}.$$

Our first main result in the next section will classify the problem for Petri nets defined as follows:

Problem 2: Fixed a Petri net $N = \langle P, T, F, M_0 \rangle$, the membership problem for firing sequences on this net N, denoted as *N-PETRI-FIRING* is:

Input: $w \in T^*$.

Question: $w \in S_N(M_0)$?.

2.3 Partially commutative monoids. Another way to model concurrent systems is with concurrent alphabets and partially commutative monoids [18]. We call *concurrent alphabet* a pair $(\Sigma, \sim)$ where $\Sigma = \{x_1, \ldots x_s\}$ is a finite alphabet denoting the set of events and $\sim$ is a symmetric and irreflexive binary relation on Σ called the *commutation relation.* The complementary notion is also useful: the *conflict relation* is defined as $\Sigma \times \Sigma \setminus \sim$. To describe equivalent behaviours in Σ^* we introduce the congruence generated by the commutation relations (i.e. if x and y commute we consider the relation $xy \sim yx$) and we denote as $w \overset{*}{\sim} w'$ the equivalence given by this congruence. The quotient monoid $\Sigma^*/\sim$ is called *partially commutative monoid* and its elements are called *traces.* If w and w' are equivalent then they model two sequential evolutions corresponding to a unique parallel behaviour. To study this equivalence we need the projection function over a subset Δ of Σ denoted as $\pi_\Delta : \Sigma^* \longrightarrow \Delta^*$. This function is defined as $\pi_\Delta(x) = x$ if $x \in \Delta$ and $\pi_\Delta(x) = \lambda$ otherwise. The trace equivalence $w \overset{*}{\sim} w'$ has been characterized in [8] in the following way:

1. for every event x in Σ we have $\pi_{\{x\}}(w) = \pi_{\{x\}}(w')$ and
2. for every pair (x, y) of different events in conflict we have $\pi_{\{x,y\}}(w) = \pi_{\{x,y\}}(w')$.

In our main results we will consider the following problem:

Problem 3: Fixed a concurrent alphabet $(\Sigma, \sim)$, the trace equivalence problem, denoted as $(\Sigma, \sim)$-*TRACE-EQUIVALENCE*, is

Input: A string $w\$w'$ where $w, w' \in \Sigma^*$ and $\$ \notin \Sigma$.

Question: It is true that $w \overset{*}{\sim} w'$?.

2.4 Finite transition systems. Concurrent systems can be analyzed also by means of transition systems [17]. Recall that a *finite labelled transition system* (FLTS for short) is a triple $M = \langle Q, \Sigma, T \rangle$, where Q is a finite set of states (or processes), Σ is a finite alphabet of actions and $T \subseteq Q \times \Sigma \times Q$ is the set of transitions. A transition $(q, x, q') \in T$ has label x and is denoted by $q \xrightarrow{x} q'$. Given two states p and q, the idea of having the same behaviour is formalized by the notion of strong bisimulation [24] (see also [20]).

A relation $S \subseteq Q \times Q$ is a *strong bisimulation* if $(p, q) \in S$ implies, for all $x \in \Sigma$, the following bisimilarity conditions:

(i) whenever $p \xrightarrow{x} p'$, then for some q', $q \xrightarrow{x} q'$ and $(p', q') \in S$,

(ii) whenever $q \xrightarrow{x} q'$, then for some p', $p \xrightarrow{x} p'$ and $(p', q') \in S$.

The *strong bisimilarity* relation $\sim$ is defined as the union of all strong bisimulations, that is

$$\sim \; = \bigcup \{S \mid S \text{ is a strong bisimulation } \}$$

Notice that the strong bisimilarity relation is also a strong bisimulation.

Other relationships such as bisimulation and observational equivalence can be defined in similar ways, using "invisible actions" [20]. It is not difficult to see that the decisional problems for these notions are equivalent to the decision of strong bisimulations.

We will prove the P-completeness of the the following problem:

Problem 4: The problem *STRONG-BISIMILARITY* is

Input: An encoding of a finite transition system with two selected states p^* and q^*.

Question: Are p^* and q^* strongly bisimilar?.

3. Main results

3.1 Petri net firing. Fixed a Petri net $N = \langle P, T, F, M_0 \rangle$ we would like to study the complexity of *N-PETRI-FIRING* problem. We start with an intuitive massivelly parallel algorithm able to solve this problem. After, we will consider some tight bounds.

Proposition 5: Given a Petri net N, the decision problem *N-PETRI-FIRING* belongs to NC.

Proof. Given a Petri net N and a sequence of transitions $w = x_1 \dots x_i \dots x_n$ it is easy to prove that w is a firing sequence iff the following holds:

- To fire the transition x_1 the following property has to be satisfied:

$$\forall p \in P : M_0(p) \geq F(p, x_1).$$

- To fire the transition x_i $(1 < i \leq n)$ it is necessary to fulfil two conditions. First the prefix $x_1 \dots x_{i-1}$ is a firing sequence. And second, all the places have to contain enough tokens to enable x_i. Both conditions can be expressed together as

$$\forall 1 < i \leq n \quad \forall p \in P : M_0(p) + \sum_{t \in T} \Delta(p,t) \cdot |x_1 \dots x_{i-1}|_t \geq F(p, x_i).$$

These conditions can be easily verified in parallel. To do this we associate a processor to every transition x_i of the input string. The processor i will operate fundamentally with transition x_i. The NC program solving this problem is given in the program "N-Petri-Firing". ∎

To obtain a tight upper bound we can express the *N-PETRI-FIRING* problem in terms of first order logic enlarged with majority quantifiers [3]. Considering Immerman's work [14], this formalism can be transformed into parallel programs running over PRAM machines enlarged with threshold operations. In our case these programs have constant time. We also give a lower bound by showing that *N-PETRI-FIRING* problem is equivalent to the *MAJ* problem under constant depth reductions. Hence this problem cannot be solved in constant time by a standard PRAM with a polynomial number of processors.

for $1 \leq i \leq n$ **do in parallel**
  for $1 \leq j \leq s$ **do**
    (* by prefix sum techniques processor i compute $count_i[j]$ *)
    $count_i[j] := |x_1 \ldots x_i|_{t_j}$
  end for;
  for $1 \leq k \leq r$ **do**
    $delta_i[k] := count_i[1] \cdot \Delta(p_k, t_1) + \ldots + count_i[s] \cdot \Delta(p_k, t_s)$;
  end for;
  if $i = 0$
    then $enabled_i := \bigwedge_{1 \leq k \leq r} (M_0[k] \geq F(p_k, x_i))$
    else $enabled_i := \bigwedge_{1 \leq k \leq r} (M_0[k] + delta_{i-1}[k] \geq F(p_k, x_i))$
  end if;
end parallel for;
(* by recursive folding all the processors help to compute the result *)
$N\text{-}Petri\text{-}Firing := \bigwedge_{1 \leq i \leq n} enabled_i$

Program. N-Petri-Firing

Proposition 6: The *N-PETRI-FIRING* problem belongs to TC^0. Moreover, a lower bound complexity is fixed by the following two assertions:

1. Fixed a Petri net N we have *N-PETRI-FIRING*$\leq_{cd}$ *MAJ*.
2. There exists a Petri net N such that *MAJ* $\leq_{cd}$ *N-PETRI-FIRING*.

From 1 and 2 we conclude that fixed a Petri net N the firing cannot solved in constant time by a PRAM with a polynomial number of processors.

For a detailed proof of this proposition see [3].

3.2 Trace equivalence. Fixed a concurrent alphabet $(\Sigma, \sim)$ we would like to study the complexity of the $(\Sigma, \sim)$-*TRACE-EQUIVALENCE* problem. As we have done above, first we will propose an intuitive massively parallel algorithm to solve this problem, and second we will consider some tight bounds.

Proposition 7: Given a concurrent alphabet $(\Sigma, \sim)$, the decision problem $(\Sigma, \sim)$-*TRACE-EQUIVALENCE* belongs to *NC*.

Proof. Given w and w' it is easy to prove that $w \overset{*}{\sim} w'$ iff the two conditions given by [8] are satisfied. These conditions can be verified in parallel.

For the first condition, i.e., every letter x of the alphabet Σ appears in w and in w' the same number of times, we use masking and prefix sum techniques as we propose in the program "Equal-Length". And for the second one, the letters of every pair in conflict appear in w and in w' following the same order, is verified also using masking and prefix sum techniques and as many processors as $\max\{|w|, |w'|\}$. For every pair (x, y) in conflict, the processor i verifies that the i^{th} letter of $\Pi_{\{x,y\}}(w)$ is equal to $\Pi_{\{x,y\}}(w')$. The program "Equal-Conflicts" verifies this condition. ∎

---

equal-length := TRUE;
for $x \in \Sigma$ **do**
  (* by masking and prefix sum techniques compute *)
  $l_1 := |\Pi_x(w)|$;
  $l_2 := |\Pi_x(w')|$;
  equal-length := *equal-length* $\wedge (l_1 = l_2)$;
end do

Program Equal-Length

We can obtain a tight upper bound on the complexity of $(\Sigma, \sim)$-*TRACE-EQUIVALENCE* by expressing it in terms of first order logic enlarged with majority quantifiers. We also have a lower bound of this problem. It can be seen that $(\Sigma, \sim)$-*TRACE-EQUIVALENCE* cannot be solved by a PRAM in constant time because it is equivalent to the MAJ problem under a constant depth reduction which increases the computation time only with a constant.

Proposition 8: The $(\Sigma, \sim)$-*TRACE-EQUIVALENCE* problem belongs to TC^0. Moreover, a lower bound complexity is fixed by the following two assertions:

1. Fixed a concurrent alphabet $(\Sigma, \sim)$ we have

$$(\Sigma, \sim)\text{-}TRACE\text{-}EQUIVALENCE \leq MAJ.$$

2. There exists a concurrent alphabet $(\Sigma, \sim)$ such that

$$MAJ \leq (\Sigma, \sim)\text{-}TRACE\text{-}EQUIVALENCE.$$

From 1 and 2 we conclude that the trace equivalence on the partially commutative monoid generated by $(\Sigma, \sim)$ cannot be solved in constant time by a PRAM with a polynomial number of processors.

equal-conflicts := TRUE;
for every pair (x,y) in conflict **do**
  (* by masking and prefix sum techniques compute *)
  $k_1 := |\Pi_{\{x,y\}}(w)|; k_2 := |\Pi_{\{x,y\}}(w')|;$
  if $k_1 = k_2$
    then
      for $1 \leq i \leq k_1$ **do in parallel**
        $u :=$ letter i^{th} of $\Pi_{\{x,y\}}(w); v :=$ letter i^{th} of $\Pi_{\{x,y\}}(w');$
        $test_i := (u = v)$
      end parallel for;
      (* by prefix sum techniques *)
      $equal\text{-}conflicts := equal\text{-}conflicts \wedge \bigwedge_{1 \leq i \leq k_1} test_i$
    else
      equal-conflicts := FALSE
    end if
end for

Program Equal-Conflicts

For a detailed proof of this proposition see [3].

3.3 Bisimulations. In contrast with these problems allowing very fast and feasible parallel algorithms we prove next that the *STRONG-BISIMILARITY* problem is *P*-complete

It is well known that strong bisimilarity in a LFTS is a polynomial time decidable property [20]. To see this, it suffices to construct $\sim$ as intersection of the sequence of relations $\equiv_0, \equiv_1, \ldots$, which are defined by induction as follows:

(i) For every $(p,q) \in Q \times Q$, $p \equiv_0 q$,

(ii) $p \equiv_{i+1} q$ if for every $x \in \Sigma$,

whenever $p \xrightarrow{x} p'$, then for some q', $q \xrightarrow{x} q'$ and $p' \equiv_i q'$;

whenever $q \xrightarrow{x} q'$, then for some p', $p \xrightarrow{x} p'$ and $p' \equiv_i q'$.

It is easy to see that these relations can be constructed in polynomial time. This is because $\sim$ coincides with $\equiv_k$ where k is the number of states in the finite transition system. More efficient algorithms to solve this problem have been considered in [15]. The *P* completeness of the *STRONG-BISIMILARITY* problem will follow from the following lemma. A more detailed proof appears in [5].

Lemma 9: *MACVP* can be reduced to *STRONG-BISIMILARITY*.

Proof. We will transform an arbitrary instance of the circuit value problem for monotone alternating circuits *MACVP* into an instance of *STRONG-BISIMILARITY* in three steps.

• We define the k-alternating pattern A_k. Figure 2 shows A_4. This is a circuit of height k, where every level has two gates, one valuated to 0 and other valuated to 1. It is easy to check then that in A_k the following two conditions are satisfied:

- every OR gate has an input valuated to 0,
- every AND gate has an input valuated to 1.

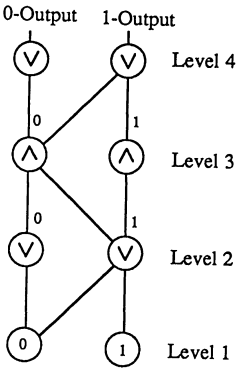


Fig. 2 The 4-alternating pattern A_4

• We couple the k-alternating pattern A_k with the circuit C to get a new circuit C'. Figure 3 shows the circuit C' constructed from the example of Figure 1. The circuit C' satisfies the following three properties:

- every OR gate has at least an input valuated to 0,
- every AND gate has at least an input valuated to 1,
- every gate of C' evaluates to the same value as the corresponding gate in A_k or C.

• We now transform the circuit C' into a FLTS M over a one letter alphabet. M contains a state corresponding to each gate of C'. These states are called *ordinary states*. In addition M contains $n+1$ *auxiliary states*, associated with the $n+1$ inputs of C' which evaluate to 1 (the n inputs of C of value 1, and the constant 1 input of A_k). We say that these auxiliary states are on level 0. Figure 4 shows M in our example.

By induction it can be shown that the circuit C evaluates to 1 with the given input values if and only if states p^* and q^* in M are strongly bisimilar.

As a consequence of the precedent lemma we obtain our announced result.

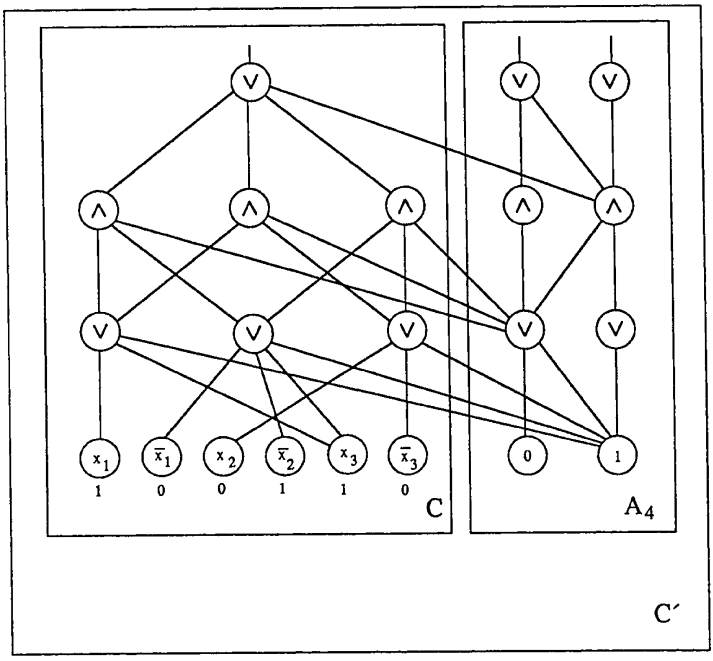


Fig. 3 The coupling of C and A_4 into C'

Theorem 10: The *STRONG-BISIMILARITY* problem is P-complete.

Other results related to this one can be obtained by the same proof idea. The properties named Observation Equivalence and Observation Congruence are defined in [19]. We can state:

Theorem 11: The problem of deciding Observation Equivalence and the problem of deciding Observation Congruence of two states in a LFTS are both P-complete.

4. Extensions

We have presented a quite precise classification of three problems on concurrency. These were the decision of firing sequences for Petri nets, the trace equivalence for partially commutative monoids, and the strong bisimulation decision problem for finite transition systems. These classifications give us hints about the complexity of massively parallel algorithms to solve them: the first two have such algorithms but however the third one cannot have such an algorithm unless all problems in P do.

Now we want to complete the discussion by raising some questions. For the bisimulation problem, our version of the statement requires the system to be part of the input. This

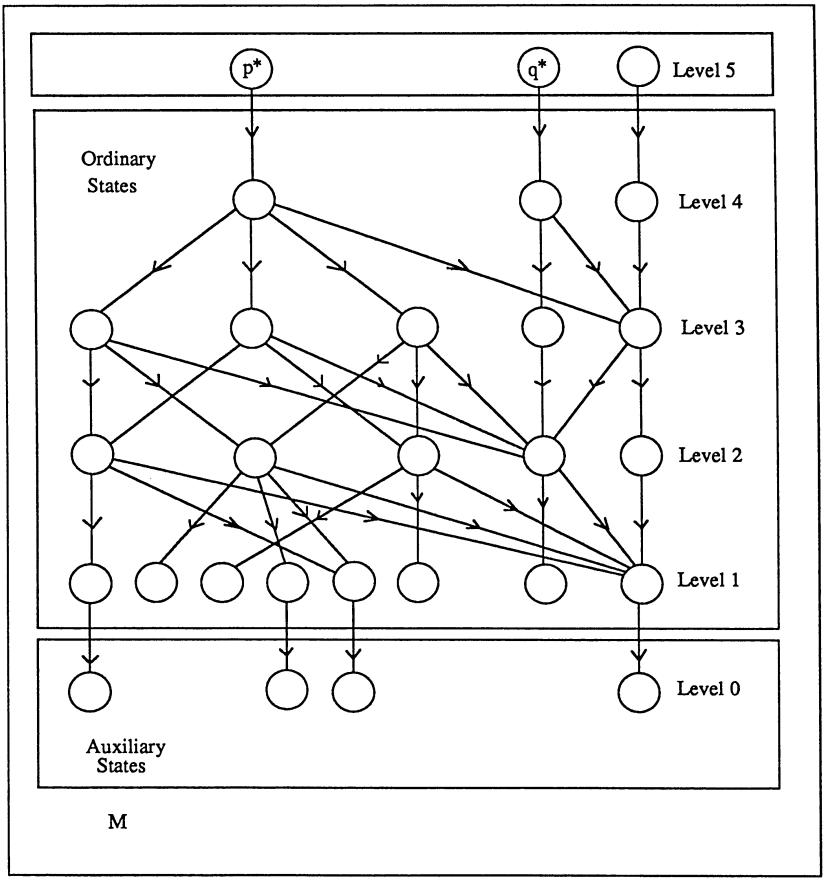


Fig. 4 The transition system M corresponding to C'

is necessary since, the system being finite, if we fix it then we obtain only a finite number of possible pairs of states, and therefore the problem can be solved in constant time.

On the contrary, in our first two problems the devices (i.e. Petri nets and concurrent alphabets) are independent of the input, and the proofs rely strongly of this fact. It is interesting to see what happens when the description of the device is added as a part of the input. For comparison, recall that context-free parsing can be done efficiently in parallel [16]; however when a coding of the grammar is added as a part of the input the complexity grows up substantially, becoming P-complete [11]. Let us briefly describe the properties of our first two problems assuming that the devices are part of the input. No proofs will be provided here.

Let us consider first the problem of general trace equivalence for partially commutative monoids. Inputs are a concurrent alphabet $(\Sigma, \sim)$, and words w and w' in Σ^*; the problem is to decide whether w and w' are equivalent. Using more complex arguments, we can prove that this problem is in uniform TC^0, and thus has the same complexity as the problem for fixed monoid; it therefore can be solved by fast parallel algorithms.

In the same way we can consider a more general version of membership for Petri net firing sequences, where the net is a part of the input. Our way of computing the variation of tokens in each place due to a prefix of the trace relies on the fact that the net is fixed, and therefore the numbers of places and transitions are constants. But if the net is part of the input, these expressions are sums of nonconstant numbers, and we must resort to a multiple addition. It is known that this problem belongs to a uniform version of TC^0 [6]. Thus we obtain that the problem belongs to uniform TC^0.

Finally, let us present some additional considerations regarding the bisimulation problem. Since it is so relevant to the design of concurrent systems, our negative P-completeness result calls for new concepts of equivalence that might be of practical value, yet testable by fast parallel algorithms. On the other hand, from the standpoint of a developer of a concurrent system, another relevant issue is whether interaction with a software tool might be more efficient than completely automatic equivalence testing.

It is well known from the study of NP problems that in many cases "verifying" is easier than "computing". This is also true in our case; indeed, the problem of whether a given relation is a bisimulation is in NC. This opens a possible way to partially overcome the P-completeness obstacle. The idea would be to design concurrent systems in an interactive way through a sequence of stepwise refinements, e.g. in the line of [12], in such a way that at every step the designer keeps direct intuition of how to transform the precedent bisimulation to obtain a new one. He then can guess the result and verify it. Perhaps only in some rare cases the designer will need to compute the whole bisimulation, and if this case is infrequent enough he would accept such a long computational process.

References

[1] Aho, A., Hopcroft, J., Ullman, J.: *The Design and Analysis of Computer Algorithms.* Addison-Wesley (1975).

[2] Ajtai, M.: Σ^1_1-formulae on finite structures. *Ann. Pure Appl. Logic* **24**, 1–48 (1983).

[3] Àlvarez, C., Gabarró, J.: The parallel complexity of two problems on concurrency. To appear at IPL.

[4] Balcázar, J.L., Díaz, J., Gabarró, J.: *Structural Complexity I.* Springer Verlag EATCS Monographs in Theoretical Computer Science, v. 11 (1988).

[5] Balcázar, J., Gabarró, J., Sántha, M.: Deciding bisimilarity is P-complete. Report LSI-90-25, Universitat Politècnica de Catalunya. Submitted for publication.

[6] Barrington, D.M., Immerman, N., Straubing, H.: On uniformity within NC^1. *J. Comp. Syst. Sci.* **41**,274–306 (1990).

[7] Chandra, A.K., Stockmeyer, L., Vishkin, U.: Constant depth reducibility. *SIAM J. Comput.* **13**, 2, 423–439 (1984).

[8] Cori, R., Perrin, D.: Automates et commutations partielles. *RAIRO Inf. Theor.* **19**, 21–31 (1985).

[9] Furst, M., Saxe, J.B., Sipser, M.: Parity, circuits and the polynomial time hierarchy. *Math. Syst. Theory* **17**, 13–27 (1984).

[10] Gibbons, A., Rytter, W.: *Efficient Parallel Algorithms.* Cambridge University Press (1988).

[11] Goldschlager, L.: ϵ-Productions in context-free grammars. *Acta Informatica* **16**, 303–308 (1981).

[12] He Jifeng: Process Simulation and Refinement. *Formal Aspects of Computing* **1**, 229–241 (1989).

[13] Hoover, H.J., Ruzzo, W.L.: A Compendium of Problems Complete for P. Manuscript (1984).

[14] Immerman, N.: Expressibility and parallel complexity. *SIAM J. Comput.* **18**, 3, 625–638 (1989).

[15] Kanellakis, P.C., Smolka, S. A.: CCS expressions, finite state processes, and three problems of equivalence. *Information and Computation* **86**, 202–241 (1990).

[16] Karp, R., Ramachandran, V.: Parallel Algorithms for Shared Memory Machines. In: *Handbook of Theoretical Computer Science*, (vol A), 869–941, editor Jan Van Leeuwen, Elsevier (1990).

[17] Keller, R.M.: Formal Verification of Parallel Programs. *Comm. ACM*, **19**, 7, 371–384 (1976).

[18] Mazurkiewicz, A: Basic notions of trace theory. Springer Verlag Lecture Notes in Computer Science 354, 285–363 (1989).

[19] Milner, R.: *A Calculus of Communicating Systems.* Springer Verlag Lecture Notes in Computer Science 92 (1980).

[20] Milner, R.: *Communication and Concurrency.* Prentice Hall (1989).

[21] Milner, R.: A Complete Axiomatization for Observation Congruence of Finite-State Behaviours. *Information and Computation.*

[22] Miyano, S., Shiraishi, S., Shoudai, T.: A list of P-complete problems. Technical Report RIFIS-TR-CS-17, Kyushu University 33, 1989.

[23] Parberry, I.: A primer on the complexity theory of neural networks. In: *Formal techniques in artificial intelligence*, R.B. Banerji (editor), North-Holland (1990).

[24] Park, D.: Concurrency and Automata on Infinite Sequences. Springer Verlag Lecture Notes in Computer Science 104, 168–183 (1981).

[25] Peterson, J.L.: *Petri net theory and the modeling of systems.* Prentice-Hall (1981).

[26] Stockmeyer, L., Vishkin, U.: Simulation of parallel random access machines by circuits. *SIAM J. Comput.* **13**, 2, 409–422 (1984).

[27] Walker, D.J.: Automated Analysis of Mutual Exclusion Algorithms using CCS. *Formal Aspects of Computing*, 1, 273–292 (1989).

FORK
A High–Level Language for *PRAM*s

T. Hagerup[1], A. Schmitt[2], H. Seidl[2]

Fachbereich Informatik, Universität des Saarlandes
Im Stadtwald, 6600 Saarbrücken, Germany

Abstract

We present a new programming language designed to allow the convenient expression of algorithms for a parallel random access machine (*PRAM*). The language attempts to satisfy two potentially conflicting goals: On the one hand, it should be simple and clear enough to serve as a vehicle for human-to-human communication of algorithmic ideas. On the other hand, it should be automatically translatable to efficient machine (i.e., *PRAM*) code, and it should allow precise statements to be made about the amount of resources (primarily time) consumed by a given program. In the sequential setting, both objectives are reasonably well met by the Algol-like languages, e.g., with the *RAM* as the underlying machine model, but we are not aware of any language that allows a satisfactory expression of typical *PRAM* algorithms. Our contribution should be seen as a modest attempt to fill this gap.

1 Introduction

A *PRAM* is a parallel machine whose main components are a set of processors and a global memory. Although every real machine is finite, we consider an ideal *PRAM* to have a countably infinite number of both processors and global memory cells, of which only a finite number is used in any finite computation. Both the processors and the global memory cells are numbered consecutively starting at 0; the number of a processor is called its *processor number* or its *index*, and the number of a memory cell is, as usual, also known as its address. Each processor has an infinite local memory and a local program counter. All processors are controlled by the same global clock and execute precisely one instruction in each clock cycle. A *PRAM* may hence also be characterized as a synchronous shared–memory MIMD (multiple-instruction multiple-data) machine.

The set of instructions available to each processor is a superset of those found in a standard *RAM* (see, e.g., [2]). The additional instructions not present in a *RAM* are an instruction LOADINDEX to load the index of the executing processor into a cell in its local memory and instructions READ and WRITE to copy the contents of a given global memory cell to a given cell in the local memory of the executing processor, and vice versa. All processors can access a global memory cell in the same step, with some restrictions concerning concurrent access by several processors to the same cell (see Section 2.5).

Among researchers working on the development of concrete algorithms, the *PRAM* is one of the most popular models of parallel computation, and the number of published *PRAM* algorithms is large and steadily growing. This is due mainly to the convenient and very powerful mechanism for inter-processor communication provided by the global memory. Curiously, there is no standard *PRAM* programming language, and each researcher, in so far as he wants to provide a formal description of his algorithms, develops his own notation from scratch. The disadvantages of this are evident:

---

[1] supported by Deutsche Forschungsgemeinschaft, SFB 124, TP B2

[2] supported by Deutsche Forschungsgemeinschaft, SFB 124, TP C1

1. At least potentially, difficulties of communication are aggravated by the lack of a common language;

2. The same or very similar definitions are repeated again and again, resulting in a waste of human time and journal space;

3. Since the designer of an algorithm is more interested in the algorithm than in the notation used to describe it, any language fragments that he may introduce are not likely to be up to current standards in programming language design.

In the wider area of parallel computing in general, much effort has gone into the development of adequate programming languages. Most of these languages, however, are intended to be used with loosely coupled multiprocessor systems consisting of autonomous computers, each with its own clock, that run mainly independently, but occasionally exchange messages. The facilities provided for interprocessor communication and synchronization are therefore based on concepts such as message exchange (Ada [16]; OCCAM [15]; Concurrent C [10]) or protected shared variables (Concurrent Pascal [13]). In particular, a global memory simultaneously accessible to all processors is not supported by such languages, and it can be simulated only with an unacceptably high overhead. While such languages may be excellent tools in the area of distributed computing, they are not suited to the description of *PRAM* algorithms.

Before we go on to discuss other languages more similar in spirit to ours, we describe what we consider to be important features of such languages. Most obviously, they must offer a way to state that certain operations can be executed in parallel. Secondly, we want to write programs for a shared–memory machine. Therefore, the language should distinguish between shared variables, which exist only once and can be accessed by a certain group of processors, and private variables, of which there might be several instances, each residing in a different processor's private memory and possibly having a different value.

Also, the machine facilities of synchronous access to shared data, should be reflected in the language. Finally, program constructs like recursion, which are well suited for writing clear and well structured sequential programs, should be allowed to be freely combined with parallelism. Recursion is characterized by a subdivision of a given problem into a set of subproblems that can be solved independently and possibly in parallel. Each subproblem may again be worked on by several processors. Therefore, the programming language should provide the programmer with a means of generating independently working subgroups of synchronously running processors. Since the efficiency of many algorithms relies on a subtle distribution of processors over tasks, an explicit method should be available to assign processors to newly created subgroups.

A frequently used tool for indicating parallelly executable program sections is a *for* loop where all loop iterations are supposed to be executed in parallel. Such a construct is, e.g., used in extensions of sequential imperative languages like FORCE [14] and *ParC* [4]. Also textbooks about *PRAM* algorithms, e.g. [3, 11], usually employ some Algol–style notation together with a statement like **for i:=1 to n pardo ... endpardo .**

A different approach is taken in the language GATT [7]. In GATT all processors are started simultaneously at the beginning. During procedure calls subgroups can be formed to solve designated subproblems. However, since GATT is designed for describing efficient algorithms on processor networks, GATT lacks the concept of shared variables. Instead, every variable has to reside in the private memory of one of the processors.

For *PRAMs*, there are various examples of descriptions of recursive algorithms using an informal group concept, e.g., see [3, sect. 4.6.3, p. 101], [8, 6]. An attempt to formulate a recursive *PRAM* algorithm more precisely is made in [5]. Corresponding to the machine-level *fork* instruction of [9], a *fork* statement is introduced, which allows a given group of synchronously working processors to be divided into subgroups. This *fork* statement gave the name to our language.

The present paper embeds the *fork* statement suggested in [5] into a complete programming language. In detail, the contributions of *FORK* are the following:

- It adds a *start* construct, which allows a set of new processors with indices in a specified range to be started.

- It makes precise the extent to which the semantics guarantees synchronous program execution (and hence synchronous data access).
- Besides the implicit synchronization at the beginning of every statement, as proposed in [5], it introduces implicit splitting into subgroups at every branching point of the program where the branch taken by a processor depends on its private variables.

It is argued that the available program constructs can be freely nested. In particular, iteration and recursion are compatible both with the starting of new processors and the forking of subprocesses.

The paper is organized as follows. In Section 2 we explain the mechanism of synchronism of *FORK* together with the new constructs in *FORK* for maintaining parallelism. Moreover, we introduce the three basic concepts of *FORK*, namely the concepts of a "logical processor", of a "group" of logical processors and of "synchronous program execution". These basic concepts are used in Section 3, which gives the semantics of the main constructs of *FORK* in an informal way. A complete description of *FORK* can be found in [12]. Section 4 concludes with some hints on how programs of the proposed language can be compiled to efficiently running *PRAM* machine code.

It should be emphasized that although our language design aims to satisfy the needs of theoreticians, we want to provide a practical language. The language *FORK* was developed in close connection with a research project at the Saarbrücken Computer Science Department that in detail explores the possibilities of constructing a *PRAM* [1] and is going to build a prototype. We plan to write a compiler for our language that produces code for this physical machine.

Both a formal semantics of *FORK* and a more precise description of a compiler for *FORK* are in preparation.

2 An overview on the programming language *FORK*

Parallelism in *FORK* is controlled by two orthogonal instructions, namely **start** [<*expr*>..<*expr*>] and **fork** [<*expr*>..<*expr*>]. The *start* instruction can be used to readjust the group of processors available to the *PRAM* for program execution, whereas the *fork* instruction leaves the number of processors unchanged, but creates independently operating subgroups for distinct subtasks and allows for a precise distribution of the available processors among them. The effect of these instructions together with *FORK*'s concept of synchronous program execution will be explained by the examples below.

2.1 Creating new processors: The *start* statement

A basic concept of *FORK* is a *logical processor*. Logical processors are meant to be mapped onto the physical processors provided by the hardware architecture. However, the number of actually working logical processors may vary during program execution; also, the number of logical processors may exceed the number of physically available processors. Therefore, these two kinds of processors should not be confused. In the sequel, if we loosely speak of "processors" we always mean "logical processors". If we mean physical processors we will state so explicitly.

Every (logical) processor p owns a distinguished integer constant ♯ whose value is referred to as the *processor number* of p. Also, it may have other private constants, private types, and private variables which are only accessible by itself. Objects declared as *shared* by a group of processors can be accessed by all processors of the given group.

As a first example consider the following problem. Assume that we are given a forest F with nodes $1,\ldots,N$. F is described by an array A of N integers, where $A[i] = i$ if i is a root of F, and $A[i]$ contains the father of i otherwise. For an integer constant N, the following program computes an array R of N integers such that $R[i]$ contains the root of the tree to which i belongs in F.

As in PASCAL, the integer constant N, the loop variable t, and the arrays A and R must be declared in the surrounding context; in *FORK* this declaration indicates whether variables are *shared* (as in the example) or *private* and hence only accessible to the individual processor itself.

```
...                                                              (1)
shared const N = ...;                                            (2)
shared var t : integer;                                          (3)
shared var A : array [1 .. N] of integer;                        (4)
shared var R : array [1 .. N] of integer;                        (5)
...                                                              (6)
start [1..N]                                                     (7)
    R[ǂ] := A[ǂ] ;                                               (8)
    for t := 1 to log(N) do                                      (9)
    /* log(N) denotes ⌈log2(N)⌉ */                              (10)
        R[ǂ] := R[R[ǂ]]                                          (11)
    enddo                                                        (12)
endstart                                                         (13)
...                                                              (14)
```

Initially there is just one processor with processor number 0. The instruction *start* [1..N] in line (7) starts processors with processor numbers $1, \ldots, N$. The corresponding instruction *endstart* stops these processors again and reestablishes the former processors. Hence the sequence of an instruction *start* [1..N] immediately followed by an instruction *start* [1..M] does not start NM processors but only M processors. An occurrence of *endstart* finishes the phase where M processors were running and again there are N processors with numbers $1, \ldots, N$.

At the machine level every instruction consumes exactly one time unit. However, the semantics of a high-level program should be independent of the special features of the translation schemes. Therefore, it should be left unspecified how many time units are precisely consumed by, e.g., an assignment statement of *FORK*.

For this reason a notion of synchronous program execution is needed which only depends on the program text itself. Again, the semantic notion of "synchronous program execution" should not be confused with the notion of a global clock of a physical *PRAM*. For example, the underlying hardware may allow different processors to execute different instructions within the same clock cycle, whereas our notion of synchronism does *not* allow for a synchronous execution of different statements. Being synchronous is a property of a set of processors. It implies that all processors within this set are at the same program point. This means that they not only execute the same statement within the same loop within the same procedure. It also means that the "history" of the recursive call to that procedure and the number of iterations within the same loop agree. There is no explicit synchronization mechanism in *FORK*. Implicit synchronization in *FORK* is done statement by statement. At the end of each statement there is an (implicit) *synchronization point*. This means that if a set of processors synchronously executes a statement sequence

$$<statement>_1 ; <statement>_2$$

the processors of this set first synchronously execute $<statement>_1$. When all processors of this set have finished the execution of $<statement>_1$ they synchronously execute $<statement>_2$. Note that within the execution of $<statement>_1$ different processors may reach different program points. Thus they may become asynchronous in between.

FORK is well structured; there are no *gotos*. Hence implicit synchronization points cannot be circumvened. Nontermination caused by infinite waiting for deviating processors is therefore not possible.

In the given example all the processors execute the same code. According to our convention they execute statement by statement synchronously. Hence, first every processor copies the value of $A[ǂ]$ to $R[ǂ]$. Recall that the constant ǂ is distinct for every processor. Then all processors assign 1 to the variable t, followed by the execution of line (11). Then they assign 2 to t, and so forth. Since the upper bound for t depends on shared data only (namely on N), all processors finish the *for* loop at the same time.

Observe here that the synchronous execution of an assignment statement is subdivided into three synchronously executed steps: first, the right-hand side is evaluated; secondly, the variable corresponding to the left-hand side is determined; finally, the value of the right hand side is assigned to the variable described by the left-hand side.

In our example, in line (11), first the value of $R[R[\sharp]]$ is computed in parallel for every processor, secondly, the variable $R[\sharp]$ is determined, which receives its new value in step three.

2.2 Forming groups of processors: The *fork* statement

FORK allows free combination of parallelism and recursion. This gives rise to the second basic concept of *FORK*: a *group*. Groups are formed by a (possibly empty) set of processors. Shared variables are always shared relative to a group of processors, meaning that they can be accessed by processors within this group but not by processors from the outside.

Groups can be divided into subgroups. This is done by the *fork* construct of *FORK*. The most recently established groups are called *leaf groups*. Leaf groups play a special role in *FORK*. As a minimum, the processors within one leaf group work synchronously. Also, if new shared objects are declared, they are established as shared relative to the leaf group executing this declaration.

Every group has a group number. The group number of the most recently created group can be accessed by its members through the distinguished private integer constant @. Clearly, the values of @ are equal throughout that group. Initially, there is just one group with group number 0 which consists of the processor with processor number 0.

As an example, consider the following generic divide-and-conquer algorithm *DC*. *DC* has a recursion parameter N describing the maximal number of processors available to solve the given problem and additional parameters containing the necessary data, which for simplicity are indicated by .... Assuming that the problem size is reduced to its square root at every recursion step, *DC* may be programmed as follows:

```
procedure DC(shared const N: integer; ... );                    (1)
...                                                             (2)
if trivial(N)                                                   (3)
    then conquer( ... )                                         (4)
    else                                                        (5)
        fork [0 .. sqrt(N)−1]                                   (6)
        @ = ♯ div sqrt(N) ;                                     (7)
        ♯ = ♯ mod sqrt(N) ;                                     (8)
            DC(sqrt(N), ... ) /* sqrt(N) denotes ⌈√N⌉ */       (9)
        endfork;                                                (10)
        combine( ... )                                          (11)
endif;                                                          (12)
...                                                             (13)
```

When a leaf group reaches the *fork* instruction in line (6), a set of subgroups with group numbers $0, \ldots, sqrt(N) - 1$ is created. These newly created groups are leaf groups during the execution of the rest of the *fork* statement, which, in the example, consists of line (9). Observe that procedure calls inside a *fork* may allocate distinct instances of the same shared variable for each of the new leaf groups.

Executing the right–hand side of line (7), every processor determines the leaf group to which it will belong.

In order to make the call to a recursive procedure simpler it may be reasonable for a processor to receive a new processor number w.r.t. the chosen leaf group. In the example this new number is computed in line (8).

When the new leaf groups have been formed, the existing processors have been distributed among these groups, and the processor numbers have been redefined, the leaf groups independently execute the statement list inside the *fork* construct. In the example this consists just of a recursive call to *DC*. Clearly, the parameters of this recursive call which contain the specification of the subproblem in general depend on the value of the constant @ of its associated leaf group.

When the statements inside a *fork* statement are finished the leaf groups disappear — in the example at line (10). The original group is reestablished as a leaf group, and all the processors continue to synchronously execute the next statement (11).

2.3 Why no *pardo* statement?

There is no *pardo* statement in *FORK*. This choice was motivated by the observation that in general *pardo* is used simply in the sense of our *start*. A difference occurs for nested *pardos*. Consider the program segment

```
for i := 1 to n pardo                                    (1)
    for j := 1 to m pardo                                (2)
        op(i,j)                                          (3)
    endpardo                                             (4)
endpardo                                                 (5)
```

Using a similar semantics as for the *start* instruction in *FORK*, the second *pardo* simply would overwrite the first one, which means that on the whole only m processors execute line (3); moreover, the value of i in line (3) would no longer be defined. This is not the intended meaning.

Instead, two nested *pardos* as in lines (1) and (2) are meant to start nm processors indexed by pairs (i, j). Precisely, a *pardo* statement of the form

for i := <*expr*>$_1$ **to** <*expr*>$_2$ **pardo** <*statement*> **endpardo**

where the expressions <*expr*>$_1$ and <*expr*>$_2$ and the statement <*statement*> do not use any private objects, can be simulated as follows:

```
begin                                                    (1)
  /* declare two new auxiliary constants in order        (2)
     to avoid double evaluation of the                   (3)
     expressions <expr>_1 and <expr>_2                   (4)
  */                                                     (5)
  shared const a1 = <expr>_1;                            (6)
  shared const a2 = <expr>_2;                            (7)

  /* start a2-a1+1 new processors ... */                 (8)
  start[a1 .. a2]                                        (9)
    /*... and distribute them among a2-a1+1 new groups */(10)
    fork[a1 .. a2]                                       (11)
      @ = ♯;                                             (12)
      ♯ = 0;                                             (13)
      /* each leaf group creates a new variable i and    (14)
         initilizes it with the group number             (15)
      */                                                 (16)
      begin                                              (17)
        shared var i : integer;                          (18)

        i := @;                                          (19)
        <statement>                                      (20)
      end                                                (21)
    endfork                                              (22)
  endstart                                               (23)
end /* of the pardo simulation */                        (24)
```

In order to avoid redundancies we decided not to include the *pardo* construct in *FORK*.

On the other hand one may argue that the *fork* construct as provided by *FORK* is overly complicated. Using the very simple *pardo* would suffice in every relevant situation. Using *pardo* a generic divide-and-conquer algorithm may look as follows:

```
procedure DC(shared const N: integer; ... );                    (1)
...                                                             (2)
if trivial(N)                                                   (3)
    then conquer( ... )                                         (4)
    else                                                        (5)
        for i := 1 to sqrt(N) pardo                             (6)
            DC(sqrt(N), ... )                                   (7)
        endpardo;                                               (8)
        combine( ... )                                          (9)
endif;                                                          (10)
...                                                             (11)
```

In the *pardo* version of DC beginning with one processor, successively more and more processors are started. In particular, every subtask is always supplied with one processor to solve it. Opposed to that, in the *fork* version the leaf group of processors is successively subdivided and distributed among the subtasks. The leaf group calling *DC* does not necessarily form a contiguous interval. Hence there might be subtasks which receive an empty set of processors and thus are not executed at all. In fact, this capability is essentially exploited in the order-chaining algorithm of [5]. This algorithm is not easily expressible using *pardos*. This was one of the reasons for introducing the *fork* construct.

2.4 Forming subgroups implicitly: The *if* statement

So far we have not explained what happens if the processors of a given leaf group synchronously arrive at a conditional branching point within the program. As an example, assume that for some algorithm the processors $1,\ldots,N$ are conceptually organized in the form of a tree of height $\log(N)$. At time t, a processor should execute a procedure *op1*(♯) if its height in the tree is at most t, and another procedure *op2*(♯) otherwise. The corresponding piece of program may look like:

```
shared var t : integer;                                         (1)
...                                                             (2)
for t := 1 to log(N) do                                         (3)
    if height(♯) <= t                                           (4)
      then op1(♯)                                               (5)
      else op2(♯)                                               (6)
    endif                                                       (7)
enddo                                                           (8)
...                                                             (9)
```

For every t the condition of line (4) may evaluate to *true* for some processors, and to *false* for others. Moreover, the evaluation of both *op1*(♯) and *op2*(♯) may introduce local shared variables, which are distinct even if they have the same names. Therefore, every *if–then–else* statement whose condition depends on private variables implicitly introduces two new leaf groups of processors, namely those that evaluate the condition to *true* and those that evaluate it to *false*. Both groups receive the group number of their father group, i.e. the private constants @ are not redefined.

Clearly, within each new leaf group every processor is at the same program point. Hence, they in fact can work synchronously as demanded by *FORK*'s group concept. As soon as the two leaf groups have finished the *then* and the *else* parts, respectively, (i.e., at the instruction *endif*) the original leaf group is reestablished and the synchronous execution proceeds with the next statement. *Case* statements and loops are treated analogously.

In the above example the condition of the *for* loop in line (3) depends only on the shared variable t. Therefore, the present leaf group is not subdivided into subgroups after line (3). However, this subdivision occurs after line (4). The two groups for the *then* and the *else* parts execute lines (5) and (6) in parallel, each group internally synchronously but asynchronously w.r.t. the processors of the other group. Line (7) reestablishes the original leaf group, which in return synchronously executes the next round of the loop, and so on.

The fact that we implicitly form subgroups at branching points whose conditions depend on private data allows for an unrestricted nesting of *if*s, loops, procedure calls and *fork*s.

Observe that at every program point the system of groups and subgroups containing a given processor forms a hierarchy. Corresponding to that hierarchy, the shared variables can be organized in a tree–like fashion. Each node corresponds to a group in the hierarchy and contains the shared variables relative to that group. For a processor of a leaf group all those variables are relevant that are situated on the path from this leaf group to the root. Along this path, the ordinary scoping rules hold.

For returning results at the end of a *fork* or for exchanging data between different leaf groups of processors it is necessary also to have at least in some cases a synchronous access to data shared between different subgroups of processors.

Consider the following example:

```
...                                                          (1)
shared var A : array [0..N−1] of integer;                    (2)
shared var i : integer;                                      (3)
...                                                          (4)
fork [0..N−1]                                                (5)
@ = ... ;                                                    (6)
♯ = ... ;                                                    (7)
    for i := 0 to N−1 do                                     (8)
        A[ (@+1) mod N ] := result(i,@,A)                    (9)
    enddo                                                   (10)
endfork                                                     (11)
...                                                         (12)
```

In this example the array A is used as a mail box for communication between the groups $1,\ldots,N$. The loop index i and the limits of the *for* loop of lines (8) and (9) are shared not only by the processors within every leaf group, but also between all groups generated in line (5). If (as in the example) the loop condition depends only on variables shared by all the existing groups, the semantics of *FORK* guarantees that the loop is executed synchronously throughout those groups.

Hence, the results computed in round i are available to all groups in round $i+1$. The general rule by which every processor (and hence also the programmer) can determine the largest surrounding group within which it runs synchronously is described in detail in Section 3.2.2.

2.5 How to solve read and write conflicts

So far we have explained the activation of processors and the generation of subgroups. We left open what happens when several processors access the same shared variable synchronously. In this case, failure or success and the effect of the succeeding access is determined according to an initially fixed regime for solving access conflicts. Most *PRAM* models allow common read operations. However, we do not restrict ourselves to such a model. The semantics of a *FORK* program also may be determined, e.g., w.r.t. an exclusive read regime where synchronous read accesses of more than one processor to the same shared variable leads to program abortion. Also, several regimes for solving write conflicts are possible. For example, we may fix a regime where common writes are allowed provided all processors assign the same value to the shared variable. In this case, the *for* loop in the example above is executed successfully, whereas if we fix a regime where common writes are forbidden, a *for* loop with a shared loop parameter causes a failure of program execution.

As another example consider a regime where common writes are allowed, and the result is determined according to the processor numbers of the involved processors, e.g., the processor with the smallest number wins. This regime works fine if the processors only access shared data within the present leaf group. If processors synchronously write to variables declared in a larger group g they may solve the write conflict according to their processor numbers relative to that group g. Consider the following example:

```
...                                                          (1)
start[0 .. 2*N-1]                                           (2)
    ...                                                      (3)
    shared var A : array[0 .. N-1] of integer;               (4)
    ...                                                      (5)
    fork[0 .. 1]                                             (6)
    @ = ♯ div N;                                             (7)
    ♯ = ♯ mod N;                                             (8)
        A[♯] := result(A,@,♯)                                (9)
    endfork                                                 (10)
endstart                                                    (11)
...                                                         (12)
```

The shared variable A is declared for some group g having according to line (2) processors numbered $0,\ldots,2*N-1$. For $n \in \{0,\ldots,N-1\}$ there are processors $p_{n,0}$ and $p_{n,1}$ of processor number n in the first and the second leaf group, respectively, that want to assign a value to $A[n]$ in line (9). This conflict is solved according to the processor numbers of $p_{n,0}$ and $p_{n,1}$ relative to group g, i.e. n and $N+n$, respectively. Hence, in line (9), the result of processor $p_{n,0}$ is stored in $A[n]$.

Observe that this scheme fails if another *start* occurs inside the *fork* statement because the original processor numbers can no longer be determined. In this case program execution fails.

In any case, the semantics of a *FORK* program is determined by the regime for solving read and write conflicts. Hence, *FORK* is intended to give the syntax and a *scheme* for the semantics of *FORK* programs.

2.6 Input/Output in *FORK*

There are at least two natural choices for designing input and output facilities for *FORK*. First, one may provide shared input/output facilities. These can be realized by (one–way infinite) shared arrays. Within this framework, synchronous I/O is realized by synchronous access to the corresponding array where conflicts are solved according to the same regime as with other accesses to shared variables (see Section 2.5).

As a second possibility one may provide private input/output facilities. These can be realized by equipping every logical processor with private input streams from which it can read, and private output streams to which it can write. The latter is the straightforward extension of PASCAL's I/O mechanism to the case of several logical processors.

Which of these choices is more suitable depends on the computing environment, e.g., the available hardware, the capabilities of an operating system and the applications. Therefore, input functions and output procedures are not described in this first draft on *FORK*.

3 The semantics of *FORK*

In this section we give a description of the constructs of the *PRAM* language *FORK*. As usual, nonterminals are enclosed in angled brackets, and we use them also for denoting arbitrary elements of the corresponding syntactical category, e.g., an arbitrary statement may be addressed by <*statement*>. The empty word is denoted by ε.

Programs are executed by processors which are organized in a group hierarchy. For each construct of *FORK* we have to explain how the execution of this construct affects the group hierarchy, the synchronism among the processors, and the scopes of objects.

We use the following terminology. A *group hierarchy* H is a finite rooted tree whose nodes are labeled by sets of processors. A node of H is called a *group (in H)*. Assume G is a group. A *subgroup of G* is a node in the subtree with root G. A *leaf group of G* is a leaf of this tree. A processor p *is contained in* a *group* G if it is contained in (the label of) a leaf group of G.

In the sequel we define inductively the actual group hierarchy and the notion of a *maximally synchronous* group w.r.t. this hierarchy. According to the inductive definition all processors in a maximally synchronous group are at the same program point. Also, each leaf group is a subgroup of a maximally

synchronous group. A group G is called *synchronous* if it is a subgroup of a maximally synchronous group. If we loosely speak of a synchronous group G executing, e.g., a statement, we always mean that all the processors within G synchronously execute this statement.

Before program execution, the group hierarchy H consists just of one group numbered 0 which contains a single processor also numbered 0. This group is maximally synchronous. At the end of program execution we again have this hierarchy H.

Declaration sequences and statement sequences are executed by maximally synchronous groups (w.r.t. the actual group hierarchy). In the sequel G always denotes such a maximally synchronous group w.r.t. the actual group hierarchy and H the subtree with root G. The execution of declarations and statements may change the group hierarchy and the synchronism — but only within the subtree H. Therefore, we only describe these changes.

3.1 *Declarations*

In order to have e.g. dynamic arrays, the values of constants may not be computable at compile time and hence have to be determined at runtime.

Assume G executes a declaration sequence

<*declaration*> ; <*decls*> .

Then G first executes the declaration <*declaration*>. During the execution of <*declaration*> the synchronism among the executing processors and the group hierarchy with root G may change. However, at the end of <*declaration*>, H is reestablished, and G is maximally synchronous again w.r.t. the actual group hierarchy. Now G starts executing <*decls*>. As in PASCAL, there are constant, type, variable, function and procedure declarations. Constant and variable declarations additionally have to determine the *access type* of the newly created data objects, i.e., whether they are *private* or *shared*. If a data object is declared private, a distinct instance of the object is created for every processor executing the declaration. If a data object is declared shared, each leaf group executing this declaration receives a distinct instance of this object, which is accessible by all the processors of this leaf group.

Also, the return values of functions and the formal parameters of procedures and functions are treated differently to PASCAL:

- In a function declaration we have to specify whether the return value should be treated as a private or as a shared value. If it is declared shared then it is shared relative to the leaf group which called the function.
- Our language uses *const* parameters instead of PASCAL's *value* parameters.
- Every formal parameter has to be declared as private or as shared.

3.2 *Statements*

Assume G is a maximally synchronous group executing a statement sequence

<*statement*> ; <*stats*> .

Then G first executes the statement <*statement*>. During the execution of <*statement*> the synchronism among the executing processors and the group hierarchy H with root G may change. However, at the end of <*statement*>, H is reestablished, and G is maximally synchronous again w.r.t. the actual group hierarchy. Then G starts executing <*stats*>.

Our language supports 6 kinds of statements:

1. assignments
2. branching statements (*if* and *case* statement)
3. loop statements (*while*, *repeat* and *for* statement)
4. procedure calls
5. activation of new processors (*start* statement)
6. splitting of groups into subgroups (*fork* statement)

The statements of types 1—4 are similar to their counterparts in PASCAL. But due to the fact that there are usually several processors executing such a statement synchronously there are some differences in

the semantics.

The *start* statement allows the activation of new processors by need. If an algorithm needs a certain number of processors these processors are activated via *start*. When the algorithm has terminated, the new processors are deactivated and the computation continues with the processors that were active before the *start*.

The *fork* statement does not change the number of active processors, but refines their division into subgroups.

3.2.1 *The assignment statement*

Assume an assignment statement of the form

$$<expr>_1 := <expr>_2$$

is executed. First all processors of G synchronously evaluate the two expressions (see Section 3.3). Each processor of G evaluates the left–hand side expression to a private or shared variable. Then the processors of G synchronously bind the value of the right–hand side expression to this variable. The effect of this is defined as follows:

1. if the variable is private, then after the assignment it contains the value written by the processor.
2. if the variable is shared, then the regime for solving write conflicts (see Section 2.5) determines the success or failure of the assignment, and, in case of success, the value to which the variable is bound.

Example Assume that four processors numbered 0 — 3 synchronously execute the assignment

$$x := @ + \sharp$$

where x is a variable of type integer. The value of x after the assignment depends on whether x is declared as private or as shared. We list some cases below.

1. x is a private variable. In this case there are four distinct incarnations of x. Then after the assignment x contains for each processor the value of the expression $@ + \sharp$.
2. the four processors form a leaf group and the variable x is shared relative to that leaf group. In this case the four processors write to the same variable and this write conflict is solved according to the regime for solving write conflicts (see Section 2.5).
3. the four processors form two different leaf groups (i.e. processors 0 and 1 are in the first one, 2 and 3 in the second one). Each leaf group has a distinct instance of the variable x. Then processors 0 and 1 write to the same variable and so do processors 2 and 3. The regime for solving write conflicts determines the value of the (two distinct) variables x after the assignment.

Above we have described the assignment of basic values. Our language supports assignment of structured values (e.g., $a := b$, where a and b are arrays of the same type), which is carried out component by component synchronously.

3.2.2 *The if statement*

Assume an *if*–statement

if <*expr*> **then** $<stats>_1$ **else** $<stats>_2$ **endif**

is executed. Then all processors of G first evaluate the expression <*expr*> synchronously. Depending on the result of this evaluation the processors synchronously execute the statements of the *then* or of the *else* part, respectively. Since different processors may evaluate <*expr*> to different values, we cannot make sure that all of them continue to work synchronously. The group hierarchies and the new maximally synchronous groups executing the *then* and the *else* part (if present), respectively, are determined according to the constants, variables and functions on which the expression <*expr*> "depends". Precisely, we say <*expr*> *depends* on a variable x if x occurs in <*expr*> outside any actual parameter of a function call. The case of constants and functions is analogous. We have to treat two cases:

1. *the expression does not depend on any private variables, constants, or functions.*

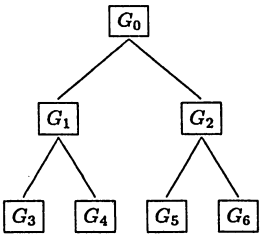
   In this case we do not change the group hierarchy H. Only the synchronism among the processors may change. Consider a processor p of G. We choose as the maximally synchronous group containing p the maximal group g_p of H which satisfies the following conditions:

   - g_p is a subgroup (not necessarily a proper one) of G.
   - p is contained in g_p.
   - The condition <*expr*> does not depend on any shared variables or other shared data relative to a proper subgroup of g_p. Note that the return value of a shared function is a shared datum only relative to a leaf group (see Section 3.3).

   Under these conditions all processors in g_p evaluate <*expr*> to the same value. Therefore all these processors choose the same branch of the *if* statement and hence are at the same program point.

   Note that a processor outside of g_p runs asynchronously with the processor p even if it evaluates expression <*expr*> to the same value.

   Example Assume that during program execution we have obtained the following group hierarchy.



   and that all processors execute synchronously the *if* statement

   if $x = 5$ **then** S_1 **else** S_2 **endif**

   where an instance of x is a shared variable relative to G_1, and another instance of x is a shared variable relative to G_2. Then the processors of group G_3 work synchronously with the processors of group G_4 and the same holds for groups G_5 and G_6, respectively. The processors of group G_4 work asynchronously with the processors of group G_6 even if the two instances of variable x contain the same value. □

   When a processor of G has finished the execution of <*stats*> it waits until the other processors of G have finished their statement sequences. When all processors of G have arrived at the end of the *if* statement, G becomes maximally synchronous again. This means that even if two processors of G work asynchronously inside the *if* statement they become synchronous again after the *if* statement.

2. *The expression depends on private variables, constants, or functions*

   In this case we cannot even be certain that all processors inside the same leaf group evaluate <*expr*> to the same value. In this case both the group hierarchy H and the synchronism are changed as follows.

   Each leaf group of G generates two new leaf groups: The first one contains all processors of the leaf group that evaluate <*expr*> to *true*, and the second one contains the rest. All new leaf groups obtain the group number of their father group. The new leaf groups become maximally synchronous. Thus, the processors inside the same new subgroup work synchronously, while the processors of different subgroups work asynchronously.

   Again, when a processor reaches the end of the *if* statement, it waits until the other processors reach this point. When all the newly generated leaf groups have terminated the execution of the *if* statement, i.e., when all processors have reached *endif*, the leaf groups are removed. The original group hierarchy H is reestablished and G is again maximally synchronous.

3.2.3 *The while statement*

A *while* statement

while <*expr*> **do** <*stats*> **enddo**

is semantically equivalent to

if <*expr*> **then**
  <*stats*> ;
  while <*expr*> **do** <*stats*> **enddo**
endif

Thus, its semantics is determined by the semantics of *if—then—else* and recursion. The semantics of other kinds of loops, e.g., *for* or *repeat* loops, may be reduced to the semantics of the *while* loop.

3.2.4 *Procedure calls*

Syntax and semantics of procedure calls are similar to those of PASCAL. There are some slight differences due to the fact that processors can access two different kinds of objects: shared and private objects. This imposes some restrictions on the actual parameters of a procedure call. One only has to be careful of the access types of formal and actual parameters:

1. If a formal parameter is a shared–var–parameter then the corresponding actual parameter has to be a variable which does not depend on any private object, e.g., ar[‖] is not allowed as an actual shared var parameter even if ar is a shared array.

2. If a formal parameter is a private–var–parameter then the corresponding actual parameter can be a private or a shared variable.

3. If a formal parameter is a shared–const–parameter then the corresponding actual parameter may denote a private or a shared value.

4. If a formal parameter is a private–const–paramenter then the corresponding actual parameter may denote a private or a shared value.

The case where a formal shared–var–parameter is bound to a private variable is explicitly excluded since it allows some processor to modify the contents of this private variable, which may belong to a different processor. In contrast, in the case of a formal shared–const–parameter the value of that formal parameter during the execution of the procedure is determined by the regime for solving write conflicts. This is done in the same way as when determining the value of a shared variable after the assignment of a private value (see Section 3.2.1).

Assume the maximally synchronous group G executes a procedure call

$$<name>(<expr>_1, \ldots, <expr>_n) .$$

First the actual parameters $<expr>_1, \ldots, <expr>_n$ are synchronously evaluated and bound to the corresponding formal parameters synchronously from left to right. Then G synchronously executes the procedure block, i.e. the declarations and the statements of that procedure.

3.2.5 *The start statement*

The *start statement* is used for activating new processors. The new processors are not allowed to access the private objects of the old processors. Therefore, statements inside the *start* statement should not refer to any private types, variables or constants, except @ and ‖, that are declared outside the *start* statement.

When the processors of G execute the statement

start [<*expr*>$_1$..<*expr*>$_2$] <*stats*> **endstart**

they first evaluate the range <*expr*>$_1$ .. <*expr*>$_2$. This range must not depend on any private data. Thus, the evaluation gives for all processors in the same leaf group g the same range $v_{g,1},\ldots,v_{g,2}$. At every leaf group g of G a new leaf group is added which contains $v_{g,2}-v_{g,1}+1$ new processors numbered with the elements of $\{v_{g,1},\ldots,v_{g,2}\}$. The group numbers of the new leaf groups are the same as for their father groups. G remains maximally synchronous. Now G executes <*stats*>, which means that the new processors of the new leaf groups execute <*stats*> synchronously. When G reaches *endstart*, the leaf groups are removed, i.e. the original hierarchy H is reestablished.

3.2.6 *The fork statement*

The *fork statement* is used to generate several new leaf groups explicitly. The new leaf groups obtain new group numbers and the processors inside the new leaf groups are renumbered.

Assume the processors of G execute the statement

fork[<*expr*>$_1$ .. <*expr*>$_2$]
@ = <*expr*>$_3$;
♯ = <*expr*>$_4$;
<*stats*>
endfork

First each processor p of G evaluates the expressions <*expr*>$_1,\ldots,$<*expr*>$_4$. The expressions <*expr*>$_1$ and <*expr*>$_2$ must not depend on any private data. Thus, all processors within the same leaf group g of G evaluate <*expr*>$_1$ and <*expr*>$_2$ to the same values $v_{g,1}$ and $v_{g,2}$ respectively. At every leaf group g of G $v_{g,2}-v_{g,1}+1$ new leaf groups are added which are numbered with the elements of $\{v_{g,1},\ldots,v_{g,2}\}$. The new leaf group with number i is labeled by the subset of those processors of g which evaluate <*expr*>$_3$ to i. Each of these processors obtains the value of <*expr*>$_4$ as its new processor number. G remains maximally synchronous and executes <*stats*>. When G reaches *endfork*, the new leaf groups are removed, i.e., the original group hierarchy is reestablished.

3.3 *Expressions*

Syntax and semantics of expressions are similar to those of PASCAL. There are just two new predefined constants: the processor number ♯ and the group number @, which both are private constants of type integer. Every processor of the maximally synchronous group G evaluates the expression and returns a value. The return value of a shared function is determined according to the choosen regime for solving write conflicts seperately for each leaf group of G and is treated as a shared object of this leaf group. Note that the evaluation of expressions may cause read conflicts, which are solved according to the choosen regime for solving read conflicts.

Example

```
...
shared var a: array[1 .. 10] of integer;
...
... := a[3]+3
...
```

Determining the variables corresponding to the subexpression a[3] does not cause a read conflict, whereas determining the values of these variables may cause read conflicts. □

4 Implementation

In this section we sketch some ideas showing that programs of *FORK* can not only be translated to semantically equivalent *PRAM* code, but also to code that runs efficiently. These considerations are

supposed to be useful both to theoreticians and to compiler writers, who may have different realizations of *PRAMs* available, possibly without powerful operating systems for memory management and processor allocation.

Our basic idea for compiling *FORK* is to extend the usual stack-based implementation of recursive procedure calls of, e.g., the P-machine [17] by a corresponding regime for the shared data structures, a synchronization mechanism, and a management of group and processor numbers. Hence, here we address only the following issues:

- creating new subgroups;
- synchronization;
- starting new processors with *start*.

4.1 Creating new subgroups

The variables which are shared relative to some group have to be placed into some portion of the shared memory of the *PRAM*, which is reserved for this group. Therefore, the crucial point in creating new subgroups is the question of how the subgroups obtain distinct portions of shared memory. There are (at least) two ways to do this with little computational overhead:

1. by address arithmetic, as suggested in [5],
2. taking into account that in practice the available shared memory always is finite, by equally subdividing the remaining free space among the newly created subgroups.

The first method corresponds to an addressing scheme where the remaining storage is viewed as a two–dimensional matrix. Its rows are indexed by the group numbers, whereas the column index gives the address of a storage cell relative to a given group. For the second method the role of rows and columns are simply exchanged. In both cases splitting into subgroups can be executed in constant time. Also, the addresses in the physical shared memory can be computed from the (virtual) addresses corresponding to the shared memory of a subgroup in constant time. This memory allocation scheme is well suited to group hierarchies with balanced space requirements. It may lead to an exponential waste of space in other cases. Consider the following while loop, whose condition depends on private variables:

```
while cond(♯) do                (1)
        work(♯)                 (2)
enddo                           (3)
```

Whenever the group of synchronously working processors executes line (1), it is subdivided into two groups, one consisting of the processors that no longer satisfy *cond*(♯), and one consisting of the remaining processors. Hence, the first group needs no shared memory space (besides perhaps some constant amount for organizational reasons); a fair subdivision into two equal portions would unnecessarily halve the space available to the second group to execute *work*(♯) of line (2).

However, there is an immediate optimization to the above storage distribution scheme: we attach only a fixed constant amount of space to groups of which it is known at compile time that they do not need new shared memory space, and subdivide the remaining space equally among the other subgroups.

This optimization clearly can be performed automatically for loops as in the given example, but also for one-sided *if*s, or *if*s where one alternative does not involve blocks with a non–empty declaration part.

4.2 Synchronization

In order to reestablish a group g, the runtime system has to determine when all the subgroups of group g have finished. This is the termination detection problem for subgroups.

If all processors run synchronously, no explicit synchronization is necessary. In the general case where the subgroups of group g run asynchronously, there are the following possibilities for implementing termination detection:

1. Use of special hardware support such as a unit–time *fetch&add* operation which allows the processors within a group to simultaneously add an integer to a shared variable.

2. Static analysis (possibly assisted by user annotation); most of the *PRAM* algorithms published in the literature are of such a simple and regular structure that the relative times of execution sequences can be determined in this way.

3. Use of a termination detection algorithm at runtime. The latter is always possible; however, complicated programs cheating a static analyzer will be punished by an extra loss of efficiency.

4.3 Starting new processors

The following method which is analogous to the storage distribution scheme works only for concurrent–read machines with a finite number of processors. Before running the program, all processors are started, all of them with processor number $\sharp = 0$. If in a subsequent *start* statement of the program fewer processors are started than what are physically available in the leaf group executing this statement, then several physical processors may remain "identical", i.e., receive the same new processor number. These identical processors elect a "leader". All of them execute the program but only the leader is allowed to perform write operations to shared variables. Consider the following example. Assume that we are given 512 physical processors.

```
start [0 .. 127]                                    (1)
    if ♯ < 64 then                                  (2)
        start [0 .. 212]                            (3)
            compute(212)                            (4)
        endstart                                    (5)
    endif                                           (6)
endstart                                            (7)
```

Before line (1), all the 512 physical processors are started. After line (1), there are always four processors having the same processor number ♯. Having executed the condition of line (2) all the processors whose processor number is less than 64 enter the *then* part: these are 256. All of them are available for the *start* instruction of line (3), where they receive the new numbers $0,\ldots,212$. Two physical processors are assigned to each of the first 43 logical processors whereas one physical processor is assigned to each of the remaining 170 logical processors. The original processor identities are put onto the private system stack. When the *endstart* in line (5) is reached, the processors reestablish their former processor numbers.

If more processors are started than physically available in the present group, then every processor within that group has to simulate an appropriate subset of the newly started processors.

In both cases *start* can be executed in constant time by every leaf group consisting of a contiguous interval of processors: this is the case, e.g., for *starts* occuring in the statement sequence of the toplevel block. Thus, a programming style is encouraged where the logical processors necessary for program execution are either started at the beginning, i.e., before splitting the initial group into subgroups, or are started in a balanced way by contiguous groups.

To maximally exploit the resources of the given *PRAM* architecture, a programmer may wish to write programs which use different algorithms for different numbers of physically available processors. Therefore, a (shared) system constant of type integer should be provided, whose value is the number of physical processors available on the given *PRAM*. This allows programs to adopt themselves to the underlying hardware.

References

[1] F. Abolhassan, J. Keller, and W.J. Paul. On physical realizations of the theoretical PRAM model. Technical Report 21/1990, Universität des Saarlandes, SFB 124, 1990.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison–Wesley, Reading Massachusetts, 1974.

[3] S.G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1989.

[4] Y. Ben–Asher, D.G. Feitelson, and L. Rudolph. ParC — An extension of C for shared memory parallel processing. Technical report, The Hebrew University of Jerusalem, 1990.

[5] P.C.P. Bhatt, K. Diks, T. Hagerup, V.C. Prasad, S. Saxena, and T. Radzik. Improved deterministic parallel integer sorting. *Information and Computation*, to appear.

[6] A. Borodin and J.E. Hopcroft. Routing, merging and sorting on parallel models of computation. *J. Comp. Sys. Sci. 30*, pages 130 – 145, 1985.

[7] M. Dietzfelbinger and F. Meyer auf der Heide (ed.). Das GATT-Manual. In: Analyse paralleler Algorithmen unter dem Aspekt der Implementierbarkeit auf verschiedenen parallelen Rechenmodellen. Technical report, Universität Dortmund, 1989.

[8] F.E. Fich, P. Ragde, and A. Widgerson. Simulations among concurrent-write PRAMs. *Algorithmica 3*, pages 43 – 51, 1988.

[9] S. Fortune and J. Wyllie. Parallelism in random access machines. In *10th ACM Symposium on Theory of Computing*, pages 114–118, 1978.

[10] N.H. Gehani and W.D. Roome. Concurrent C. In N.H. Gehani and A.D. McGettric, editors, *Concurrent Programming*, pages 112–141. Addison Wesley, 1988.

[11] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms.* Cambridge University Press, 1988.

[12] T. Hagerup, A. Schmitt, and H. Seidl. FORK — A high–level language for PRAMs. Technical Report 22/1990, Universität des Saarlandes, SFB 124, 1990.

[13] P.B. Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engeneering 1(2)*, pages 199–207, June 1975.

[14] H.F. Jordan. Structuring parallel algorithms in a MIMD, shared memory environment. *Parallel Comp. 3*, pages 93–110, 1986.

[15] Inmos Ltd. *OCCAM Programming Manual.* Prentice Hall, New Jersey, 1984.

[16] United States Department of Defense. Reference manual for the Ada programming language. ANSI/MIL-STD-1815A-1983.

[17] St. Pemberton and M. Daniels. *Pascal implementation: The P4 compiler.* Ellis Horwood, 1982.

NEURAL NETWORK-BASED DECISION MAKING FOR LARGE INCOMPLETE DATABASES

A. R. Hurson, B. Jin and S. H. Pakzad
Department of Electrical and Computer Engineering
The Pennsylvania State University
University Park, PA 16802

Abstract

As an extension to the relational algebra, maybe algebra operations have been proposed to handle incomplete information. Such a set of operations allows the user to investigate the potential set of data values (i.e. tuples) to draw his/her own conclusions. However, maybe algebra operations could return nonrelevant data, generate low quality results, and offer low physical performance. Hence, it is appropriate to design a scheme to investigate the results generated by the maybe operations, in order to improve the data quality and performance of large databases. Such a mechanism should be dynamic to adjust itself according to the user's query and the characteristics of the underlying databases. In this paper, an artificial neural network-based decision support system for handling large databases containing incomplete information is proposed. It is a subsystem which learns and constructs a knowledge base to filter out the data that is not of any importance to the user. The network accomplishes the decision-making task in a massively parallel manner. This paper also discusses the implementation of the decision-making network based on the VLSI design of a Basic Neural Unit (BNU). Using a weight-centered design principle, BNU can be expanded and reconfigured to satisfy the requirements of the underlying environment.

I. Introduction

Missing information presents a problem for any data model. Such a data value can be interpreted as the "value at present unknown". The inclusion of missing information provides unknown data for the attributes in a database. Within the scope of relational model, a partial tuple is the one that contains one or more incomplete (null) data. A tuple is said to be total if it contains no null values. Similarly, a relation is total if all of its tuples are total. As the relational database model has matured, researchers have examined the question of how to handle missing data [2,4,5,6,8,9,14,15]. The extensions of relational algebra have been addressed in [4,8,9,15]. This study is based on Codd's maybe algebra[4] which adopts a three-valued logic for handling partial relations. Such an extended capability allows the user to probe the database for potential data relationships that might be useful but can not be retrieved by true relational algebra.

While such a set of maybe operations brings a better information utilization, it also has drawbacks.

For example, maybe join operator has the potential to return nonrelevant data relationships which are not defined according to the semantics of the underlying database. Moreover, it might generate extremely large result relations, which could cause drastic degradation in performance if most of the resultant tuples are not of interest to the user. [8] shows that the loss in physical performance is paralleled by a loss in logical performance. Zaniolo's generalized join [19] removes the join over two nulls, but it still has the potential to produce extremely large relations.

What is needed then is a dynamic mechanism which functions as a filter with learning capability to adjust itself according to the specific characteristics of the underlying database and the requirement of the user's query. To fulfill the requirement of such a dynamic mechanism, we propose a neural network-based decision support system. The neural network-based approach has advantages over conventional knowledge-based systems, due to its strong dynamic and self-adaptive characteristics [7]. In a neural network, learning algorithm sets the connection strengths to their appropriate values based on valid input/output pairs (i.e. training pairs) [1]. In this way, the network learns and constructs a knowledge base to filter out the useless information. This paper will explore these points and will show how a knowledge acquisition model can be built for the decision-making network.

The dynamic nature of the databases and user's query require expandability and reconfigurability at the underlying decision-making network. Such a requirement creates a major problem for the traditional neuron-centered design techniques. In this paper, a weight-centered design principle is used instead, which yields a high parallelism and reconfigurability for the constructed networks. A Basic Neural Unit (BNU), based on the weight-centered approach, is designed to be used as basic building block in the construction of an expandable and reconfigurable decision-making network.

This paper is organized as follows: Section 2 overviews the concept of neural network-based decision support system. The VLSI design of BNU and the construction of a decision-making network using BNUs are discussed in Section 3. Section 4 presents the simulation results to analyze the performance of the proposed decision support system. Finally, Section 5 concludes the paper.

II. A Neural Network-Based Decision Support System

The maybe operations could offer a low-level logical and physical performance. To improve such a behavior, it is desired to build an intelligent mechanism which can make appropriate judgments over the resultant tuples, i.e., filtering out low quality and erroneous data and providing higher quality result to the user. To achieve such a goal, we propose a neural network-based decision support system, as shown in Figure 1. This system learns, adjusts and adapts itself to make certain decisions by applying its learned knowledge to the real database environment.

The neural network-based decision support system is composed of four modules: i) decision-making network; ii) knowledge acquisition; iii) decision-controlled buffer; and iv) user-system interface.

2.1 Adaptive Learning and Decision-Making Network

This module is a three layer neural network, an input layer, a hidden layer, and an output layer (Figure 2). The decision-making network is characterized by a set of neurons, pattern interconnection, propagation rule, output function, and learning rule. It is a fully connected network, i.e., all input units are connected to all hidden neuron-units, and all hidden neuron-units are connected to all output neuron-units.

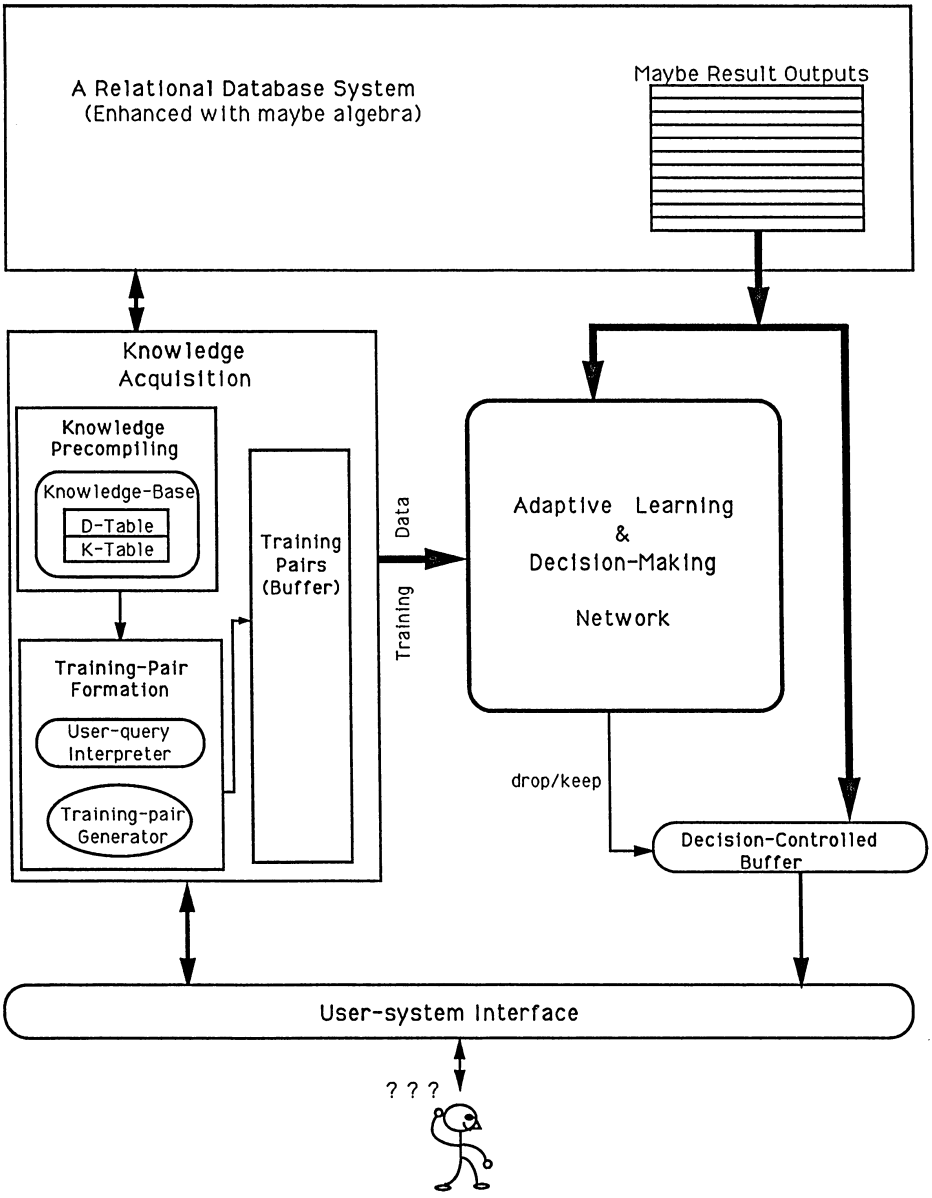


Figure 1. A neural network-based decision support system

No computation function is performed by the units of the input layer. They simply distribute an input pattern to the hidden layer. Input is a binary bit pattern, representing the resultant tuples generated by maybe operations, such as maybe join. Each input unit can take a value of either "1" (meaning known attribute value) or "0" (meaning unknown or null attribute value). The number of input units in the decision-making network varies with the number of attributes in the resultant tuple. The proposed decision-making network is designed to have a reconfigurable architecture in both software simulation and hardware implementation [7]. Hidden neuron-units compose a layer of abstraction, pulling features from the input pattern[11,13,16,18]. In the decision-making network, the number of hidden units is also adjustable [7]. The decision responses are shown in the output layer. It represents the classification associated with each input pattern. Classifications (or network outputs) are binary and can be interpreted as categories. The leftmost output unit, in Figure 2, represents the "keep" decision, which means that the associated input tuple should be passed to the user for its high quality information; while the rightmost unit represents the "drop" decision due to the low quality and/or erroneous information. Besides these two categories, we can define certain intermediate categories representing some degree of data quality. This will provide the user a more flexible "keep"/"drop" range. It can be recognized as a flow-controlled filter which controls the quantity (as well as the quality) of the filtered tuples. Hence the number of output neuron-units should also be adjustable.

The decision-making network functions in either of the two operation modes, learning mode or decision-making mode. When the network works in the learning mode, it receives a number of training pairs from the knowledge acquisition module. Each training pair consists of a known input pattern and a desired output pattern. Using supervised learning algorithm of the "generalized delta rule", the network adjusts its weights so that it can respond to the input patterns as closely to their respective desired responses as possible. Once the network is trained, it is ready to switch to the decision-making mode. Working in the decision-making mode, the network receives input patterns generated by maybe operations, and responds to the closest classification (e.g. "keep", "drop", etc.). Generalization is considered to be successful if the network responds correctly, with high probability, to input patterns that were not included in the training set [11]. The decision made by the network is transferred to decision-controlled buffer as a control signal .

2.2 Knowledge Acquisition

The goal of knowledge acquisition in the decision support system is to gather a set of training pairs for the decision-making network. In the knowledge acquisition module, knowledge can be obtained from the database semantics, i.e. data dependency relationships and key attributes. Decision-making process requires to understand the semantics of the underlying database as well as the requirements of the user's query. We propose a two-level hierarchical knowledge acquisition module for the decision support system (Figure 1). In the first level, knowledge can be acquired by extracting relationships that are implicitly/explicitly defined in a database system, independent of the user's query. This can be done by a knowledge precompiling unit. It provides long-term knowledge with a strong portability. However, to form the training pairs proper information should be also extracted from the user's query. This part of the knowledge can be extracted by a user-query interpreter unit in the second level of knowledge acquisition. In this level, knowledge acquisition takes place after a query is submitted. The acquired knowledge, referred to as short-term knowledge, differs from query to query and can not be shared. Since we assume that the knowledge-base is constructed in advance, only the speed of the short-term knowledge acquisition could affect the performance of knowledge acquisition module in the real-time situation.

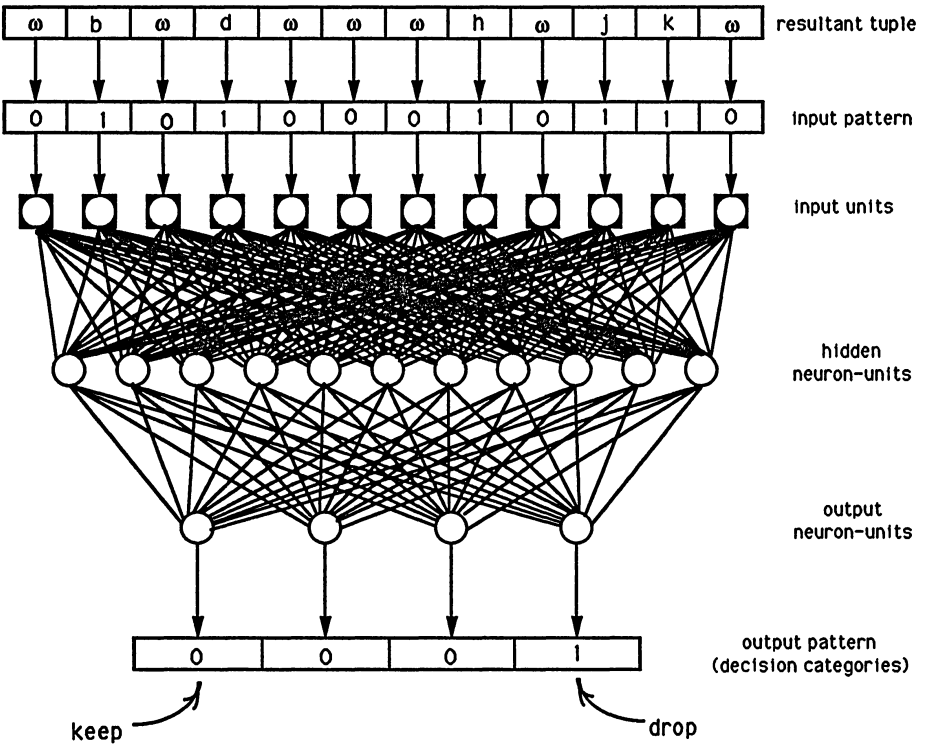


Figure 2. A neural natwork-based decision-making network

Level 1: Knowledge Precompiling

In the decision support system, accumulating useful information directly from the underlying relational database system and reorganizing it for the formation of training pairs, is referred to as knowledge precompiling. Two concepts in a relational database system form the theoretical foundation for knowledge precompiling: *data dependencies* and *key attributes* . The knowledge base consists of a D-table and a K-table. D-table stores the knowledge captured by analyzing the data dependency relationships. A K-table is constructed as part of the knowledge base to store the knowledge acquired from various kinds of key attributes, i.e., *candidate keys* (including *primary keys*, *alternate keys*) and *foreign keys* [17].

Level 2: Training-Pair Formation

The second level of the knowledge acquisition is the training-pair formation. Each time a query is submitted to the system, a set of training pairs ought to be generated based on the user's query. A user-query interpreter is designed to analyze and recognize the set of query dependent information needed to generate the training pairs. Consequently, the long term and short term knowledge are used, along with a set of heuristic criteria, to form the training pairs. This set of heuristic rules is determined by a careful analysis of the user's requirements and the semantics of the database, such as:

- If a resultant tuple has null values for all the candidate keys, then the decision should be "drop".
- If a resultant tuple has null value for the primary key attribute, but with one of the alternate keys defined, then the decision should be "keep".
- If, in the resultant tuple, attribute A functionally determines attribute B, and A is null, then the tuple should be "dropped".
- Under the previous dependency condition, if A is defined, but B is null, then this tuple should be "kept".
-
-
-

These guiding criteria should be enforced by the system designer. Moreover, it should be flexible enough to satisfy the requirements of the ever changing environment. Finally, all generated training pairs are stored in a training-pair buffer to be sent to the decision-making network.

2.3 Decision-Controlled Buffer

Decision-controlled buffer is a delay device which simply holds the resultant tuples and waits for the output from the decision-making network. By the time when a decision arrives, the buffer will either pass a resultant tuple to the user (if a "keep" decision was made) or filter out the tuple (if a "drop" decision was made). The delay time would depend on the execution speed of the decision-making network.

2.4 User-System Interface

It functions as an interface between the user and the decision support system. It provides a convenient communication environment for both the user and the system. It explains the result to the user, and more importantly, links the user to the knowledge acquisition module through a dialogue system.

III. Implementation of Decision-making Network Using BNU Chips

Software simulations have been conducted to demonstrate the feasibility of the proposed decision support system, e.g., the decision-making network and the knowledge acquisition modules[7]. Simulations were run to investigate two performance parameters, the accuracy and the training time of the decision-making network. As a result, the design of such a decision support system was verified by the obtained average accuracy of network response between 70 percent to 97 percent. It was shown that the decision-making network converged within short training times. As part of the simulation, it was demonstrated that these two parameters are also a function of the number of hidden units[7]. While the simulation results have shown the feasibility of the proposed scheme, it did not give any insight regarding the parallel capability of the proposed decision-making network. To demonstrate the parallel capability of the proposed decision-making network, a Basic Neural Unit (BNU) was designed and fabricated. The BNU is used as the basic building block in the construction of a decision-making network. The effectiveness of parallel processing in decision support was exploited by hardware implementation of the decision-making network.

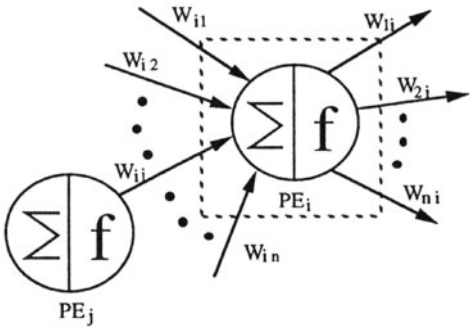
3.1 The VLSI Design of BNU(Basic Neural Unit) chip

Neuron-centered Design v.s. Weight-centered Design

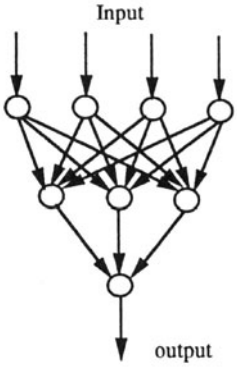
The decision-making network is composed of a set of processing elements (PEs) and the synaptic weight interconnections. Each PE simulates a neuron which sums all the weighted inputs from other PEs (neurons). Typically, this is a neuron-centered PE, as shown in Figure 3(a). In neuron-centered design, once a PE or a set of PEs is implemented, it is usually difficult to expand the size of the network with the increased number of inputs. For example, by combining two networks of Figures 3(b) and 3(c), an expanded 9-input, 3-output, three layer network can be constructed, as shown in Figure 3(d). However, such a small scale expansion requires 36 additional interconnections. In VLSI implementation, the requirement for additional connections complicates the design strategy where strong modularity, expandability and reconfigurability are necessary.

Due to this restriction we propose a new approach, namely weight-centered design. Figure 4(a) shows a 4-input, 1-output weight-centered processing element. It consists of 4 weight units, 1 accumulator, and a nonlinear function unit. It is functionally equivalent to a 4-input, 1-output neuron-centered PE. Each weight unit stores a weight value. Similar to a neuron-centered PE where each weight value has an associated input, in a weight-centered PE each input should be received by its associated weight unit to form the weighted input value that is shifted down to the accumulator. The accumulator then sums all the weighted inputs and sends the result to a nonlinear function unit to generate the final output. In the proposed weight-centered PE, two ports are left open for the purpose of expansion and reconfiguration. Horizontally, inputs can be sent to more weight units to increase the number of output neurons. For example, by adding another set of weight units, the network of Figure 4(a) can be expanded to a 2-output processing unit (Figure 4(b)). In this way, the number of output neurons can be easily expanded and reconfigured. Vertically, weight-centered PEs can be stacked up to increase the number of input neurons. Figure 4(c) shows an 8-input, 1-output processing unit. Note that the nonlinear function unit is not a part of the basic building block, it is required only at the final output.

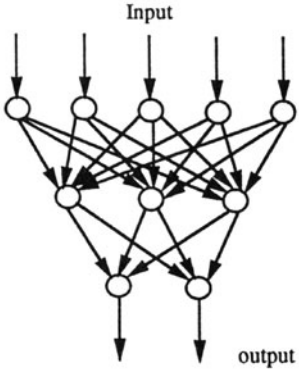
To form a multilayer decision-making network, weight units are divided into groups, called layer-



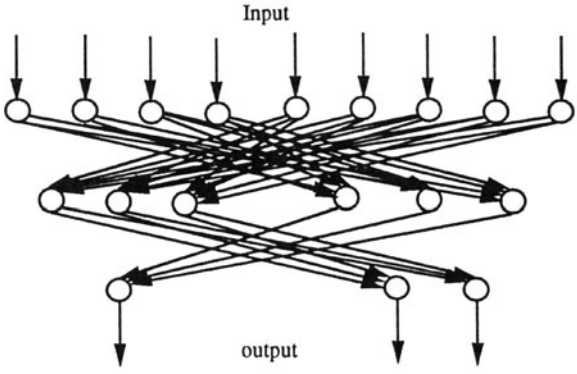
(a) A neuron-centered processing element



(b) A 4-input, 1-output three-layer network

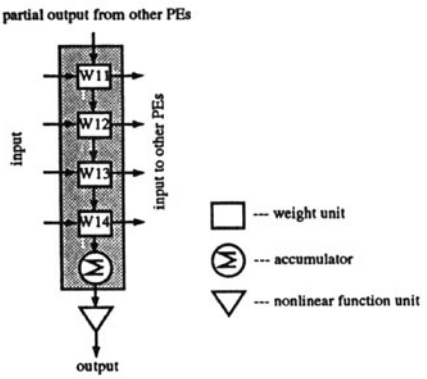


(c) A 5-iuput, 2-output three-layer network

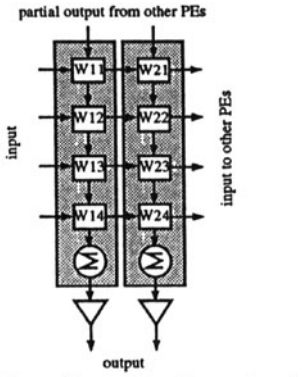


(d) A 9-input, 3-output three-layer network constructed by combining (b) and (c)

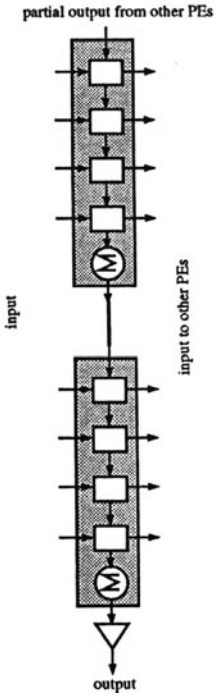
Figure 3. Neuron-centered design of decision-making networks



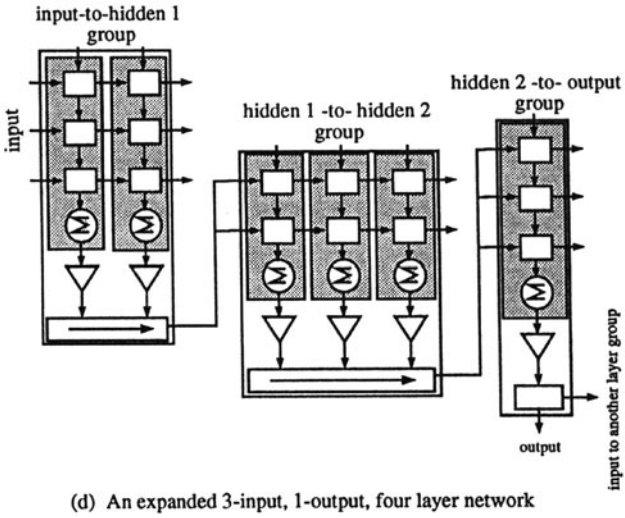
(a) A weight-centered processing element



(b) A 4-input, 2-output, two layer network



(c) An expanded 8-input, 1-output, two layer network



(d) An expanded 3-input, 1-output, four layer network

Figure 4. Weight-centered design of decision-making networks

to-layer board. Each group (board) represents weight units between each two adjacent layers. For example, Figure 4(d) is a four layer network constructed by using weight-centered design principle.

The Design of Basic Neural Unit (BNU)

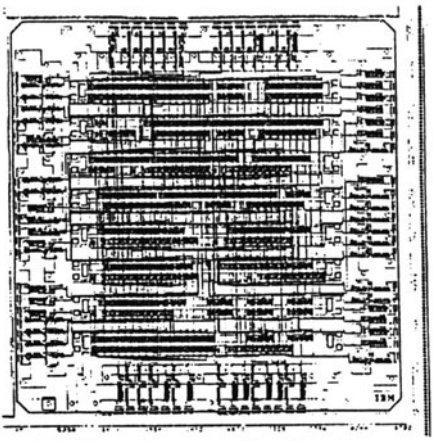
Weight-centered design technique is promising due to its modularity and expandability. In contrast to the neuron-centered design, it simplifies the inter-neuron connections especially when expanding a network. Based on the concept of weight-centered design, a Basic Neural Unit (BNU) was proposed and fabricated. Besides the expandability and reconfigurability, embedding more weight units into a single silicon chip serves the main objective in designing the BNU. Due to the fabrication process available to us, a BNU consisting of 30 weight units, 10 inputs and 3 outputs was fabricated by the IBM corporation. A photomicrogragh of BNU chip is shown in Figure 5(a). BNU is packaged in a 120-pin package. The BNU chip has a maximum time delay of 153.6 ns to generate the partial output sums.

Figure 5(b) shows the circuit schematic of a BNU chip. Each BNU forms three identical and independent processing strings, i.e., three weight groups, running in a parallel fashion. W's represents the weight input. The 10 inputs are denoted by a's. b(k+1) is the output. And, b(k) represents the partial output coming from other BNU chip when more than one BNU are used in constructing a network. One weight group is functionally equivalent to a neuron-centered PE which has 10 inputs and 1 output. The BNU works in two functional modes. It first enters the initialization mode in which weight values are loaded into three weight groups, then it is switched to the execution mode in which three groups receive the same set of inputs, and generate three outputs. Figure 5(c) demonstrates the detail schematic of one weight group in a BNU chip. The 10 weight units are implemented as shift registers. At the top is a multiplexing register that can either load the weight values during the network initialization or form the WEIGHT ITERATION LOOP when the network is in the execution mode. Each group is augmented by two more multiplexing registers. One is used to receive the INPUT via an enable input ("en"). It either passes a weight value to the adder if the input value is "1" or rejects it (i.e. sends out a "0") if the input is "0". Another multiplexing register is used to loop back the partial sum of weighted inputs to form the ADDITION LOOP. With the help of these two registers, the adder generates a partial sum output of 10 weighted inputs. The latter register is also responsible for receiving partial output from another BNU and pass to the adder if more than one BNU are used in the network. It is noted that because of our limited fabrication process, the "a" values are passed in sequential fashion. Naturally, such approach has affected the execution time of the chip.

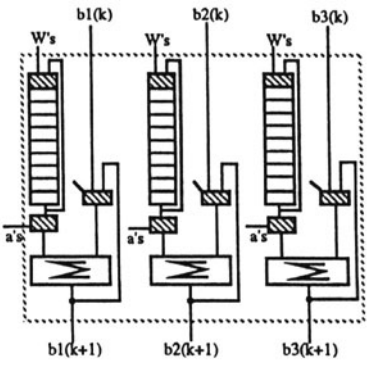
3.2 Constructing a Decision-making Network with BNU Chips

Varying physical characteristics of databases and dynamic nature of the user's queries require varying number of neurons on input, output and hidden layers. As a result, the supporting decision-making network should have the ability to be reconfigured to satisfy the requirements of the user's query. This can be accomplished by using BNU chips as basic building blocks in the construction of a decision-making network. On one hand a BNU itself forms a 10-input, 3-output, two layer network, on the other hand it can be connected with other BNUs to construct a larger network.

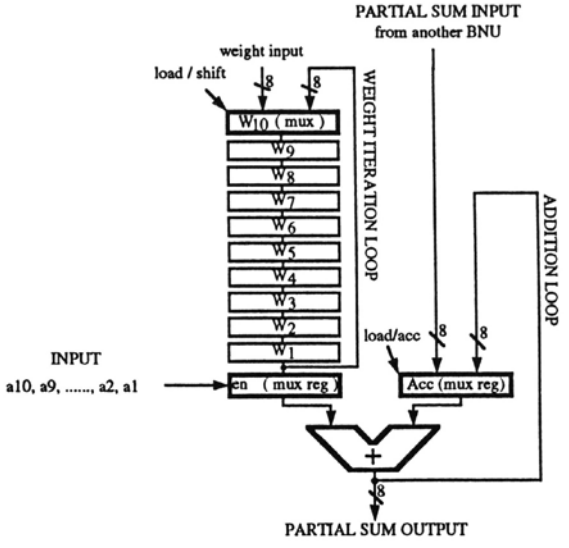
Each BNU chip is a 10-input, 3-output, two layer network. By adding another BNU chip along the vertical direction (i.e., the PARTIAL SUM OUTPUT of one BNU is connected to the PARTIAL SUM INPUT of another BNU), it can be extended to a 20-input, 3-output module. By the same token, attaching one BNU to another in the horizontal direction will extend the network to a 10-input, 6-output module. For example, Figure 6(a) depicts a 4 connected BNUs on the input-to-hidden layer board which forms 20



(a) Photomicrograph of a BNU chip

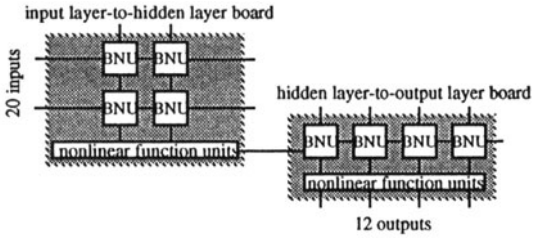


(b) Circuit schematic of a BNU chip

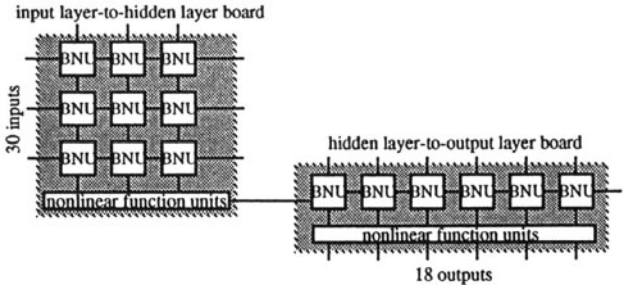


(c) single weight group in a BNU chop

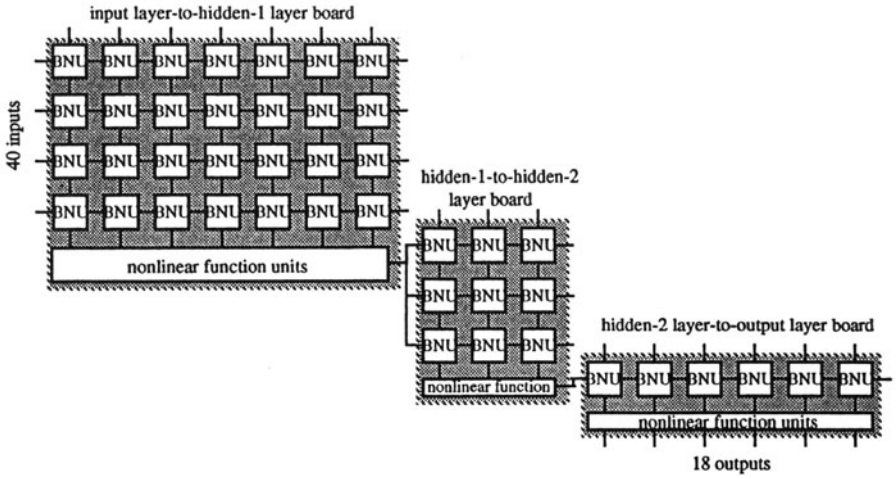
Figure 5. VLSI design of a Basic Neural Unit (BNU) chip



(a) A three layer network constructed by 8 BNU chips with 20 inputs and 12 outputs.



(b) A three layer network constructed by 18 BNU chips with up to 30 inputs and 18 outputs.



(c) A four layer network constructed by 48 BNU chips with up to 40 inputs and 18 outputs.

Figure 6. Constructing a decision-making network using BNU chips

inputs and 6 outputs. In general, adding a "row" of BNUs in the vertical direction of a network can increase the number of inputs by 10, while adding a "column" of BNUs in the horizontal direction can increase the number of outputs by 3. Figure 6(b) shows a 30-input, 9-output, two layer network on the input layer-to-hidden layer board.

The proposed BNU can be used to expand the number of layers in a multilayer network through the use of layer-to-layer board. In general, each two-layer network forms a group of BNUs which are mounted on a circuit board, called a layer-to-layer board. For example, in Figure 6(a) two layer-to-layer boards, an input-to-hidden layer board and a hidden-to-output layer board, are used to obtain a three layer network. When connecting the two, the number of outputs in the first group should match up with the number of inputs in the second group. Note that the second group has 10 inputs and 12 outputs. However, only 6 inputs are used to match the 6 outputs generated by the first group.

When expanding a multilayer network, the same design principle can be applied to an individual layer board. As an example, Figure 6(b) is an expanded three layer network based on the one shown in Figure 6(a). On the input-to-hidden layer board, a row of BNUs is added to increase the number of input neurons by 10. Adding another column of BNUs, the number of hidden neurons can be extended to 9. On the hidden later-to-output layer board, 2 more BNUs are used in order to form 18 outputs. Moreover, adding one more layer board will increase one layer in the decision-making network. Figure 6(c) demonstrates the construction of a four layer network consisting of 3 layer-to-layer boards. It can be easily observed that the network has maximum 40 input neurons, 21 neurons on the first hidden layer, 9 neurons on the second hidden layer, and 18 output neurons.

IV. Performance Analysis and Simulation Results

A number of simulation runs have been conducted to show the feasibility of the proposed decision support system. A generator was developed to generate two relations --- student relation and company relation. The student relation contained the social security number, class standing, major, area of interest, and GPA. The company relation was composed of company's name and it's hiring requirements (e.g., area of interest, major, minimum required GPA).

The simulation is based on the following assumptions:

1. The entire database resides in memory.
2. There are 138 majors offered.
3. There are approximately 600 areas of interest.
4. There are 500 tuples in the student relation, and 50 tuples in the company relation.

The simulation was run based on four different classes of queries, to represent a variety of join operations (i.e., join over single and multiple attributes, θ-join and attribute maybe join):

Query class 1 --- The first class of queries was performed using the company's hiring area of interest and a student's area of interest as the join attribute. The null value in the student relation simply meant that the student was not sure about his/her area of interest was.

Query class 2 --- The second class of queries concerns only those students who satisfied the minimum GPA requirement for hiring, i.e., a company's minimum GPA requirement. In this type of queries, the company's minimum GPA had a domain from 2 to 4.

Query class 3 --- The third class of queries was performed using area of interest, major, and the GPA as join attributes. Even if a student doesn't know his/her area of interest, the student's GPA and major might satisfy a company's criteria and therefore should be considered.

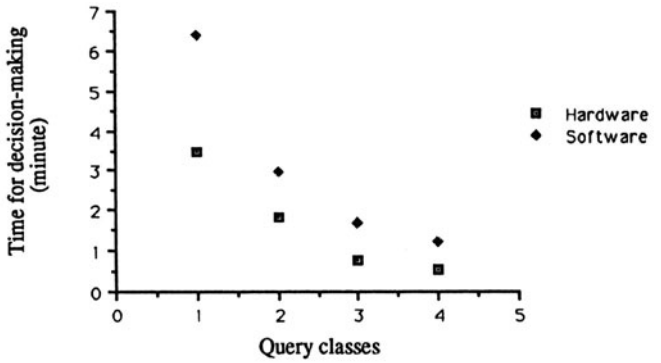
Query class 4 --- The fourth class of queries was based on the attribute maybe-join on major and area of interest.

The following table lists some information about the simulation results:

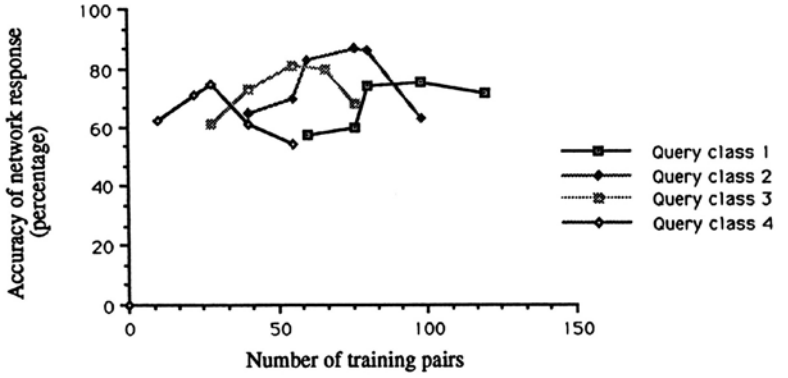
Query class	Total # of Resultant Tuples	# of training pairs Generated	# of Correct Decisions	# of Incorrect Decisions	Accuracy of Response
1	608	98	458	150	75.33%
2	412	76	357	55	86.65%
3	337	55	274	63	81.31%
4	153	28	114	39	74.51%

In the above table, the training pairs for each query class were generated by the knowledge acquisition module. As a verification of the performance of this module, the decision-making network achieved an accuracy of response between 74% to 86%. The higher accuracy obtained in query class 2 is mainly due to the high quality of training pairs. The issue of how to improve the quality of training pairs is under the consideration.

To investigate the performance improvement of the proposed BNU-based network against software simulator, the simulation was run based on the characteristics of fabricated BNU chip. Three layer decision-making network was used for all four query classes, while the size of the network varies among different classes, depending on the length of the resultant tuple. Figure 7(a) shows the simulation results. Note that the length of the resultant tuples of query class 1 is the longest among all. Therefore, class 1 requires the longest decision-making time. Moreover, the time difference between hardware and software implementation tends to be larger when the length of the resultant tuple is longer, i.e., the size of the decision-making network is larger. In another word, the length of the resultant tuple affects the decision-making time required by the software more than it affects the hardware. In a separate simulation run the accuracy of knowledge acquisition module was analyzed based on the size of the training-pair set for each query class (Figure 7(b)). We observed that the size is a function of the length of the resultant tuples. As a result, for query class 1 the module was able to generate more training pairs than the other classes of queries. This is due to the fact that for relations with larger length the module can extract more short-term and long-term knowledge. In addition, as can be observed the accuracy after a certain point, is a function of the number of training pairs. This can be interpreted that too many training pairs will bring too much noise which would degrade the accuracy of the network responses. On the other hand, fewer number of training pairs may decrease the generalization ability of the network which will also affect the accuracy.



(a) Time required for decision-making



(b) Accuracy led by the size of the set of training pairs

Figure 7. Some simulation results and performance analysis

V. Conclusion

A neural network-based decision support system for handling large databases containing incomplete information is proposed. The major difficulty which hinders the use of maybe algebra operations is due to its tendency in generating large volume of low quality resultant data. It is the learning capability and strong dynamic self-adaptive nature of a neural network which make it effective to filter out low-quality tuples and extract higher-quality ones.

The design of the proposed decision support system has resulted in a model consists of four modules: adaptive learning and decision-making network (neural network), knowledge acquisition module, decision-controlled buffer, and user-system interface. The principle of operation of each module and the interrelationships among modules have been discussed. The VLSI implementation of a Basic Neural Unit based on the concept of weight-centered design was addressed. It was shown that BNU offers a high degree of modularity, expandability and reconfigurability if it is used as the basic building block of the proposed decision-making network. The execution time and accuracy of the proposed decision-making network based on the architectural features of the implemented BNU chip was simulated and analyzed. In addition, we have shown that these performance measures are directly related to the length of the input data.

References

[1] Below, R.K., "Designing Appropriate Learning Rules for Connectionist Systems," in Proceedings, IEEE first International Conference on Neural Networks, June 1987, pp. II479-II486.

[2] Biskup, J., "A Foundation of Codd's Relational Maybe-Operations," ACM Transaction Database Systems, Vol. 8, No. 4, 1983, pp. 608-636.

[3] Burr, D.J., "A Neural Network Digit Recognizer," in Proceedings of the IEEE Conference on Systems, Man, and Cybernetics, 1986, pp. 1621-1625.

[4] Codd, E.F., "Extending the Database Relational Model to Capture Meaning," ACM Transactions on Database Systems, Vol. 4, No. 4, 1979, pp. 397-434.

[5] Codd, E.F., "Missing Information (Applicable and Inapplicable) in Relational Databases," SIGMOD Record, Vol. 15, No. 4, 1986, pp. 53-78.

[6] Grant, J., "Null Values in a Relational Database," Information Processing Lett. 6 (1977) 156-157.

[7] Jin, B., and Hurson, A.R., "Neural Network-Based Decision Support For Incomplete Database Systems," Proceedings of Analysis of Neural Net Applications Conference, ANNA-91, May 1991, To appear.

[8] Hurson, A.R., Miller, L.L. and Pakzad, S.H., "Incomplete Information and the Join Operation in Database Machines," Proceedings of Fall Joint Computer Conference, 1987, pp. 436-443.

[9] Hurson, A.R., and Miller, L.L., "Database Machine Architecture for Supporting Incomplete Information," Journal of Computer System Science and Engineering, Vol. 2, No. 3, 1987, pp. 107-116.

[10] Jin, B., and Raggad, B., "A Reconfigurable Architecture for A VLSI Implementation of Artificial Neural Networks," Proceedings of 1990 International Neural Network Conference (INNC 90), Paris, July 1990, pp. 665-668.

[11] Kohonen, T., "State of the Art in Neural Computing," In Proceedings, IEEE First International Conference on Neural Networks, June 1987, pp. I79-I90.

[12] Lendaris, G.G., "Neural Networks, Potential Assistants To Knowledge Engineers", The Journal of Knowledge Engineering, Vol. 1, No. 3, Dec. 1988, pp 7 - 18.

[13] Lippermann, R.P., "An Introduction to Computing with Neural Nets," IEEE ASSP Magazine, April 1987, pp. 4-22.

[14] Lipski, W., "On Semantic Issues Connected with Incomplete Information Databases, ACM Transactions on Database Systems, Vol. 4, No. 3, 1979, pp. 262-296.
[15] Pakzad, S.H., Hurson, A.R., and Miller, L.L., "Maybe Algebra and Incomplete Data in Database Machine ASLM," Journal of Database Technology, 1990.
[16] Rumelhart, D.E., Hinton, G.E., and Williams, R.J., "Learning Internal Representations by Error Propagation," in Parallel Distributed Processing (PDP): Explorations in the Microstructure of Cognition, Vol. I: Foundations, MIT Press, Cambridge, Massachusetts, 1986, pp. 318-362.
[17] Ullman, J.D., Principles of Database Systems, 2nd Edition, Computer Science Press, Rockville, MD 1982.
[18] Widrow, B., Winter, R.G., and Baxter, R.A., "Learning Phenomena in Layered Neural Networks," in Proceedings, IEEE first International Conference on Neural Networks, June 1987, pp. II411-II429.
[19] Zaniolo, C., "Relational views in a database system: support for queries," IEEE COMPSAC, 1977, pp. 267-275.

An Optical Content-Addressable Parallel Processor for Fast Searching and Retrieving *

Ahmed Louri
Department of Electrical and Computer Engineering
The University of Arizona, Tucson, Arizona 85721

Associative processing based on content-addressable memories has been argued to be the natural solution for non-numerical information processing applications. Unfortunately, the implementation requirements of these architectures using conventional electronic technology have been very cost prohibitive, and therefore associative processors have not been realized. Instead, software methods that emulate the behavior of associative processing have been promoted and mapped onto conventional location-addressable systems. This however, does not bring about the natural parallelism of associative processing, namely the ability to access many data words simultaneously.

The inherently parallel nature and high speed of optics, combined with the recent technological advancements in optical logic, storage and interconnect devices are raising hopes for practical realization of highly parallel optical computing systems. This paper presents the principles of designing an optical content-addressable parallel processor, called OCAPP, for the efficient support of high speed symbolic computing. The architecture is designed to fully exploit the parallelism an high speed of optics. Several parallel algorithms are mapped onto OCAPP in bit-parallel as well as word-parallel fashion, resulting in efficient symbolic algorithms with execution times dependent only on the precision of the operands and not on the problem size. This makes OCAPP very suitable for applications where the number of data sets to be operated on is high e.g., massively parallel processing. A preliminary optical implementation of the architecture using currently available optical components is also presented.

1 Introduction

The "information explosion" seen in recent years has stimulated the development of computer-based information systems to assist in the creation, storage, modification, classification, and retrieval of mainly textual or symbolic data. For example, progress in database management systems, expert systems, and intelligent knowledge-based systems is increasing demand for symbolic information processing such as text editing, file processing, table sorting, searching, and retrieval. In fact a substantial proportion of the work-load of modern information processing systems involve searching and sorting symbolic data[1]. Nevertheless, a majority of today's computers are designed mainly for numerical computations, and suffer from a fundamental handicap, which stems from the principle of addressing the memory.

When a search for a value is made through a location-addressable memory, the entire memory may need to be searched one word at a time (if the data is not sorted in memory) which consumes a great deal of time. There is no logical reason why the search must be done sequential. The only reason stems from the fundamental handicap of separating processing and memory and addressing memory one word at a time. This fundamental flaw has forced system analysts and programmers to develop sophisticated software techniques for symbolic information processing such as hashing

*This research was supported by an NSF Grant No. MIP-8909216.

and indexing[2]. However, the implementation of such software techniques on location-addressed computers has lead to complex, expensive, and inefficient information processing systems.

Searching, retrieving, sorting and modifying symbolic data can be significantly improved by the use of content-addressable memory (CAM) instead of location-addressability. In a content-addressable memory data is addressed by its contents[2]. An associative processor is a parallel processing machine in which the data items are content-addressable with the added capability to write in parallel into words satisfying certain criterion. It may be that the entire contents of stored words may be changed or just a few bits of the words. Using this model, processing is carried out within the associative memory, without transfer to an independent processing unit. Since there is no addressing of data and no data movement, this implies the elimination of the fundamental von Neumann bottleneck encountered in conventional systems. Moreover, the amount of time required for searching, retrieving, and updating information is independent of the data set sizes.

However, this model of computing is not being largely used because of the difficulty and high cost of implementing it in conventional electronic technology. This can be seen from the following:

1. Each bit cell in an associative memory is much more complex and requires more circuitry than does a conventional cell. Even with the advent of VLSI technology, the single cell complexity still does not allow for the use of large associative memories.

2. The memory storage provides poor storage density compared with conventional memory.

3. The third major difficulty is the complexity of the interconnects. Recall that in order for all cells to compare their values to that of the comparand register, the control unit must broadcast the value to all cells involved in the comparison. However, using conventional technology, the time delays associated with the broadcasting function are very appreciable. Moreover, inter-cell interconnects become cumbersome for large array size.

4. The fourth difficulty is the lack of efficient means of implementing parallel access to the cells, namely parallel input and output.

There are two hypotheses underlying this paper:

1. that CAM-based processing provides a sound basis to uncover inherent parallelism in symbolic processing and information retrieval applications, and

2. that optics is, potentially, the ideal medium to exploit such parallelism by providing efficient implementation support for it.

2 Optical Content-Addressable Parallel Processor

Optical systems hold the promise for providing efficient support for future parallel processing systems. Optics advantages have been cited on numerous occasions[3, 4, 5, 6]. These include inherent parallelism, high spatial and temporal bandwidths, and non-interfering communications. For CAM-based processing, optics may be the ideal solution to the fundamental problems faced by electronic implementations, namely cell complexity, interconnects latency, difficulty of implementing information broadcasting and parallel access to the stored data. Optics can alleviate the cell complexity by migrating the implementation of wiring and logic into free-space. The multi-dimensional nature of optical systems allows for data storage and logic to be performed on two-dimensional planes while the third dimension can be used for interconnects. The high degree of connectivity available in free-space space-invariant optical systems (10^6 to 10^8), and the ease with which optical signals can be expanded (which allows for signal broadcasting) and combined (which allows for signal funneling) can also be exploited to solve the interconnects problems[7, 8]. Moreover, optical

and electro-optical systems can offer a considerable storage capacity and parallel access than do pure electronic systems[9].

Figure 1 depicts a preliminary organizational structure for an optical content-addressable parallel processor called OCAPP. The architecture is organized in a modular fashion, and consists of a *selection unit*, a *match/compare unit*, a *response unit*, an *output unit*, and a *control unit*. The architecture is developed to meet four goals, namely: (1) exploitation of maximum parallelism; (2) amenability to optical implementation with existing devices; (3) modular design in that it can be scalable to bigger problems; and (4) ability to efficiently implement information retrieval, and symbolic computations. Moreover, the programming methodology for OCAPP is compatible with that of existing single-instruction multiple data (SIMD) systems. In what follows we describe the role of each unit. Detailed optical implementation of OCAPP will be presented in Sec.3.

The selection unit is schematically described in Fig.2. It is comprised of (1) a storage array of n words, each m bits long (in actuality, the storage array capacity is $n \times 2m$, since each bit position is comprised of a true bit w_{ij} and its complement $\bar{w}_{ij}$); and (2) word and bit-slice enable logic to enable/disable the words and/or the bit-slices that participate in the match operation, and reset the rest. It is assumed that the storage array can be loaded in parallel and (if need be) read in parallel.

The match/compare unit shown in Fig.3, contains a (1) $1 \times m$ interrogation register I; (2) logic hardware to perform parallel bitwise comparison between the bits of the interrogation register and the enabled bits of the storage array; (3) two $n \times 1$ working registers, G and L, which are used for magnitude comparisons (to be explained later); (4) a $n \times 1$ response register R for displaying the result of the comparison; and (5) a single indicator bit called the match detector MD, which indicates whether or not there is any matching words. This unit allows comparison of a single operand stored in the interrogation register and the words stored in the storage array. As such it is considered an SIMD (single-instruction-multiple data) unit. Bit position R_i of R is set to one when word W_i of the storage array matches the contents of I. The I register is a combination of the comparand register C and the mask register M as shown in table 1. As such, it holds the operand (depending on masking information if any) being searched for or being compared with. It is assumed that register I is available in dual-rail logic (both true and compliment bits available).

The response unit is responsible for selecting one or several matching words. It comprises several scratchpad registers and a priority circuit for selecting the first matching word. Depending on program control, the output of the response unit is routed either to the output unit for outputting the result or fed back to the selection unit for further processing of the matching words. All units are under the supervision of a conventional control unit with conventional storage (eg., a local RAM) which stores the program instruction. Its role is to load/unload the storage array, set/rest various registers such as the I, R, G and L of the match/compare unit, enable/disable memory words, perform conditional instructions, monitor the MD bit, and test program termination. In what follows, we describe the implementation of several parallel algorithms on the OCAPP in order to show its use and processing benefits.

Table 1: **Formulation of the interrogation register**

Search bit c_j	Mask bit m_j	Interrogation bits I_j I_j
0	0	0 1
1	0	1 0
0	1	1 1
1	1	1 1 (no comparison is performed at this bit position)

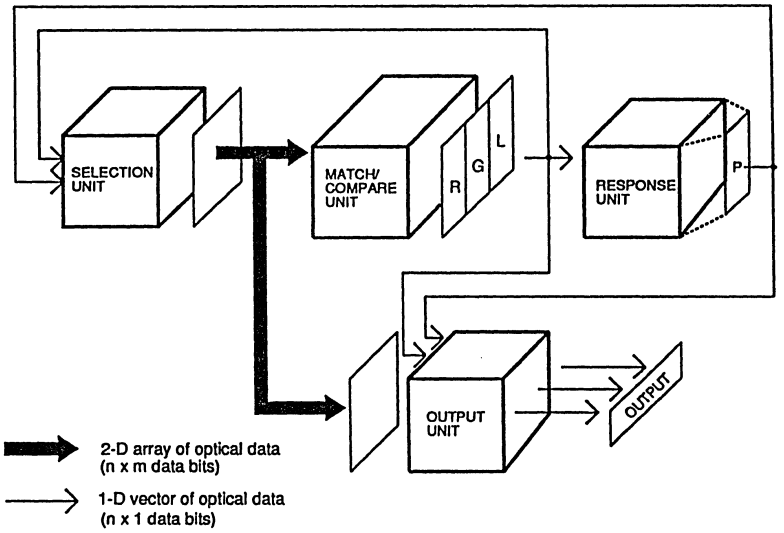


Figure 1 : A schematic organization of the proposed optical content-addressable parallel processor : OCAPP.

3 Parallel Search Algorithms on OCAPP

We classify search operations as basic and compound operations. A basic search operation is one which can be completed in one sweep over all the bit-slices of the storage array. It does not involve any feedback processing. A compound search operation requires a feedback from the response unit to the selection unit. As a consequence, it takes more than one sweep over the storage array to complete. Under basic search operations, we group the following operations:

- *Equivalence Search*: The equality search, the not-equal-to search, and the similarity search (search for a match within a masked field).
- *Threshold Search*: The smaller-than, the not-smaller-than, the greater-than, and the not-greater-than searches.
- *Extrema Search*: The greatest value search, and the smallest value search.

Compound search operations can be implemented in a series of basic search operations. Under the compound search, we group the following operations:

- *Adjacency Search*: Next-above search, and next-below search.
- *Between-Limits Search*: Search for words z, between two limits X and Y $(X < Y)$: a) $X \leq z \leq Y$, b) $X < z \leq Y$, c) $X \leq z < Y$, and d) $X < z < Y$.

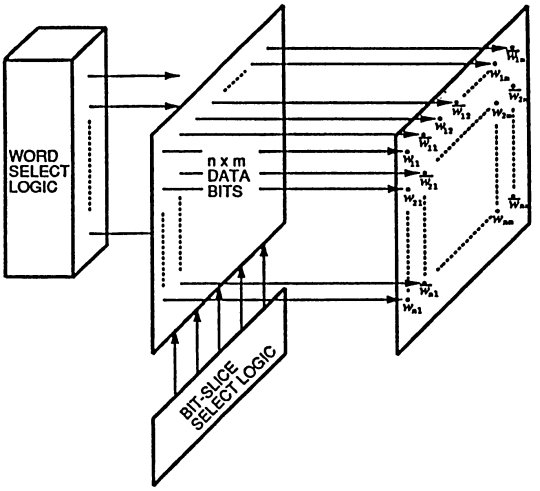


Figure 2 : Organization of the selection unit.

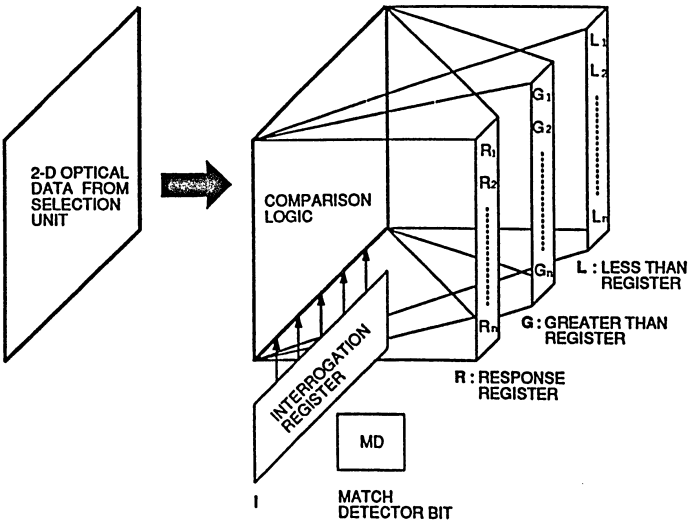


Figure 3 : Organization of the match/compare unit.

- *Outside-Limits Search*: Search for words z, outside two limits X and Y ($X < Y$): a) $X \leq z$ or $z \geq Y$, b) $X > z$ or $z \geq Y$, c) $X \geq z$ or $z > Y$, and d) $X > z$ or $z > Y$.
- *Ordered Retrievals (sorting)*: Ascending order retrieval, and descending order retrieval.

Of course many more compound search operations can be formulated using the basic search operations. The above search operations are the most frequently used in information retrieval applications.

3.1 Parallel Algorithms for Basic Search Operations on OCAPP

In what follows, we denote a memory word as $W_i = (w_{im} w_{im-1} \dots w_{i1})$ where w_{ij} is the jth bit cell of the word W_i. We denote the jth bit-slice by $B_j = (w_{1j} w_{2j} \dots w_{nj})$, which is made up of the jth bit of every word in the storage array. The interrogation and response registers are denoted by $I = I_m I_{m-1} \dots I_1$, and $R = R_1 R_2 \dots R_n$ respectively. The comparand word (search argument) and the mask register words are denoted by $C = (c_m c_{m-1} \dots c_1)$, and $M = (m_m m_{m-1} \dots m_1)$.

3.1.1 Equivalence Search

In this type of search, the memory is partitioned according to the magnitude of the search word C into two sets, namely, words which are equal to C and words which are not. The equality and masked search operations can be implemented by a bitwise match. For equality match all the bits of the search word need to be matched, whereas for the masked search, only a subset of the bits of the search word is compared with the respective bits of the memory words. For $m_j = 0$ means that c_j is not masked, while $m_j = 1$ means c_j is masked. These two search modes can be combined as shown in Table 1. Given an interrogation word I, a bit match denoted by b_{ij} on the jth cell of the ith word is given by:

$$b_{ij} = (I_j \wedge w_{ij}) \vee (\bar{I}_j \wedge \bar{w_{ij}}) \quad (\text{ equivalence }) \tag{1}$$

where the symbols $\wedge$, $\vee$, and the bar ($\bar{}$) denote the logical AND, logical OR and logical NOT respectively. Now the exact matching of memory word W_i with interrogation vector I requires the logical product of the bits b_{ij} for $j = 1, \dots, m$, therefore:

$$R_i = \bigwedge_{j=1}^{j=m} b_{ij} = b_{im} \wedge b_{im-1} \wedge \dots \wedge b_{i1}. \tag{2}$$

where $\bigwedge$ denotes a logical AND over all bits. The above equation indicates that matching words in memory will be flagged by having their corresponding R bit set to one, and all mismatches will have their R bits set to zero. Equations 1 and 2 are space-invariant and can be implemented in bit-parallel as well as word-parallel fashion. Therefore, all R_is for $i = 1, \dots, n$, are computed at the same time with a single access to the storage array.

Equivalence Search Algorithm:

1) *Initialization:*

   a) Load I (this will depend on the search word and the masking condition);

   b) Clear R (clear all bits of the R register);

2) *Perform comparison:*

   a) $b_{ij} = (I_j \wedge w_{ij}) \vee (\bar{I}_j \wedge \bar{w_{ij}})$;

   b) $R_i = \bigwedge_{j=1}^{j=m} b_{ij}$ for $i = 1, \dots, n$. ($R_i = 1$ if and only if W_i matches I).

3.1.2 Threshold Search

This mode of search partitions the memory according to the magnitude of the search word C into three sets, namely words which are equal to C, words which are less than C, and words which are greater than C. The result of the search is stored in the three registers of the response unit, namely R, G and L. Initially, all memory words are made active by making control registers RGL = 100. The memory is scanned from the most significant to the least significant bit position by enabling a single bit-slice at a time. When the comparand bit c_j is one, we select all active memory words with $w_{ij} = 0$ as "less than" by setting their corresponding bit position RGL = 001. These words are then disabled from further comparisons (the disabling process will be explained later). Similarly, when $c_j = 0$, we select all active memory words with $w_{ij} = 1$ as "greater than" by setting their corresponding bit position RGL = 010, and then disable them from further processing. At the end of the last bit position, words still in the state RGL = 100 are equal to the comparand, words in the state RGL = 010 are greater than the comparand, and words in the state RGL = 001 are less than the comparand. It is important to note that, even though we are scanning the memory from most significant bit to least significant bit, the search process can be terminated any time there are no matching words at a given bit position ($R_i = 0$ for all $i = 1, \ldots, n$). Such a condition is easily detectable by checking the MD bit. The detailed algorithm follows.

Threshold Search Algorithm:

1) *Initialization:*

   a) Load I (depending on the search word and masking condition);

   b) Enable memory words;

   c) Set R, clear G, clear L, set $j = m$ (the variable j is used by the control unit to scan the storage array);

2) *Perform Magnitude Search at bit-slice j:*

   a) $R_i = \bigwedge_{j=1}^{j=m} b_{ij}$, $G_i = \bigwedge_{j=1}^{j=m} I_i \wedge \bar{w}_{ij}$, $L_i = \bigwedge_{j=1}^{j=m} \bar{I}_i \wedge w_{ij}$ for $i = 1, \ldots, n$ (note that only the enabled bit-slice j determines the values of R_i, G_i and L_i, all other bit-slices are disabled at this time, and therefore have no influence);

   b) Test if MD = 1 (is there any words that match the I register at the current bit position j ?);

3) *If $MD = 1$ do :*

   a) Disable memory words whose corresponding bits in R are zero (memory words W_i with $R_i = 0$ have already been decided on);

   b) Decrement j: $j \leftarrow j - 1$, and test if $j = 0$?;

   c) If $j \neq 0$, go to step 2;

   d) If $j = 0$, go to step 4;

4) *If ($MD = 0$) or ($j = 0$)*, then we are done and the search result is reported in R, G, and L.

The following example illustrates the algorithm for a magnitude search of 7 words, each 5 bits long:

Example 1: Threshold Search

Search word, S :	10110
Mask word:	00000
I register:	10110 (effective word search)

Memory word i	W_i	State of RGL at the end of the jth iteration					
		j = 5	j = 4	j = 3	j = 2	j = 1	(last iteration)
1	10111	100	100	100	100	010	($W_1 > S$)
2	11000	100	010	010	010	010	($W_2 > S$)
3	10010	100	100	001	001	001	($W_3 < S$)
4	10110	100	100	100	100	100	($W_4 = S$)
5	10101	100	100	100	001	001	($W_5 < S$)
6	01101	001	001	001	001	001	($W_6 < S$)
7	11101	100	010	010	010	010	($W_7 > S$)

3.1.3 Extrema Search

This type of search refers to finding the maximum (or minimum) of a set of (or all) memory words. We consider first the search for maximum.

A. Maximum Search

To find the maximum, we scan memory words from the most to the least significant bit positions. As we scan the bit-slices, we determine if any of the enabled words have a one in the current bit position. If we find some, we disable all those words that do not have a one in this position. If none of the words at the current position possess a one, we do nothing. At any given time, all remaining candidates are equal as far as we have examined them, because for every bit position either everybody had a zero in that bit position, or whenever some words have ones, we disable the ones with zeros. Therefore, at bit position j, enabled words with $w_{ij} = 1$ are larger than enabled words with $w_{ij} = 0$. Since we are seeking the maximum, we disable the ones with $w_{ij} = 0$. This process is repeated until we exhaust all bit positions at which time the maximum word will be indicated by $R_i = 1$.

Algorithm for Finding the Maximum:

1) *Initialization:*
   a) Load I :$I \leftarrow 11\ldots11$ (I is loaded with all bits set to one);
   b) Clear MD, set $j = m$;
   c) Enable memory words, and set $R_i = 1$ for $i = 1, \ldots, n$;
2) *Perform equivalence search at bit-slice j:*
3) *Test if* $MD = 1$ (is there any words with a one in the current bit position j ?);
4) *If* $MD = 1$ *do :*
   a) Disable all words which do not have a one in the current bit position (these words are indicated by $R_i = 0$).
   b) Clear R and MD;
   c) Go to step 5;
5) *Decrement* j : $j \leftarrow j - 1$, and test if $j = 0$?;
   a) If $j \neq 0$, go to step 2;
   b) If $j = 0$, output maximum value indicated by $R_i = 1$.

B. Minimum Search

The search for the minimum is very similar to the search for the maximum except that the I register is initially loaded with zeros and that if any enabled word has a zero in the current bit

position (There exists a memory word W_i such that its corresponding $R_i = 1$), we disable the words with a one in the current bit position ($R_i = 0$). These words are bound to be greater than the minimum sought. The process is repeated until we exhaust all bits of the enabled words. The minimum value will also be indicated by a one in register R.

3.2 Parallel Algorithms for Compound Search Operations on OCAPP

Compound search operations such as the ones stated earlier can not be economically implemented by a single sweep over the memory words. We therefore choose to implement such operations as a series of basic searches. The rationale is to keep the architecture as simple as possible, and therefore making it highly amenable to optical implementation. Of course speed improvements can be gained by implementing these search operations as basic search, but the amount of logic circuits may be extensive.

3.2.1 Double Limits Search (Between and Outside Limits)

Given two numbers called HIGH and LOW, the double limits search consists of finding those words that are between this limits and/or words that are outside these limits. This gives rise to eight different searches which can be accomplished in a very similar manner. Let us consider the between limit search. Given the two numbers HIGH and LOW, we wish to find those words that are greater than LOW but less than HIGH namely, find all W_i such that $LOW < W_i < HIGH$. We can accomplish this search by using the magnitude comparison search as follows. First, we determine the words that are less than the comparand HIGH. These words will be indicated by a one in the L register. We then disable all other words except the ones that are less than HIGH, and perform another threshold search using the comparand LOW. After the second search, words that are less than HIGH and greater than LOW will be marked with a one in the G register, which could be routed to the output unit for outputting the search result.

3.2.2 Adjacency Search

To find the word that is next-above the comparand (the smallest word larger than the comparand), we search for all words that are larger than the comparand and then select their minimum. Similarly, to find the word that is next-below the comparand (the largest word smaller than the comparand), we search for all words less than the comparand and select their maximum. The search for the largest word smaller that the comparand (next-below search) can be carried out by a similar algorithm as the one above. In this case, step two of the next-above algorithm is replaced by a search for words that are less than the comparand, and step four is replaced by a maximum search.

3.2.3 Ordered Retrieval (Sorting)

The sorting or ordered retrieval of a set of data can be achieved by performing the extrema search repeatedly until all the data are retrieved. For the ascending order retrieval, we enable the memory words to be sorted, and determine their minimum (using the minimum search operation). We output the obtained minimum value and disable it from the storage array. We repeat these steps until we retrieve (in ascending order) all the enabled words. For descending order retrieval, we select the maximum value at each step.

4 Optical Implementation

In this section we identify the fundamental and basic operations required to implement the optical architecture, and describe possible optical components for achieving them. Detailed practical implementation issues and experimental setups will be the subject of a different publication.

4.1 Basic Operations and Hardware Components Required

An analysis of the conceptual OCAPP, the basic operations, and the algorithms reveals that in order to optically implement OCAPP, the following functions are required: (1) data is optical and must be available in dual-rail format (both the value and its complement is required); (2) parallel access for writing into and reading from the storage array as well as the various control registers; (3) disabling/enabling a memory word (or several memory words) based on certain criteria; (4) logical AND, and logical OR; (5) space-invariant optical transmission of information (one-to-one connections); (6) spreading a single bit (actually two bits due to the dual-rail format) of information to several spatial locations (one-to-many connection); (7) combining several bits of information into a single spatial location (many-to-one connection); and (8) dynamic routing of information (e.g., routing contents of register R to selection unit, or output unit, or response unit depending on the algorithm). The optical components required to accomplish the above operations can be divided into (1) logic elements, (2) storage elements, and (3) information transfer elements (or interconnects).

For optical logic and storage, many approaches are being investigated. One approach is the adaptation of the spatial light modulator (SLM) technology to optical logic[10]. Another approach for realizing optical components capable of performing logic, is to optimize the device from the beginning for digital operations. The recent emergence of the quantum-well self-electrooptic effect device (SEED) and its derivatives (S-SEED, T-SEED, D-SEED) is one such a product[11]. The SEED devices can be used to realize both logic operations such as NOR, OR, AND, NAND, etc. as well as for storage such as S-R latches[11]. Optical resonators are another family under this approach intended for optical logic[12]. Two similar bistable devices, etalons, and interference filters both based on the Fabry-Perot resonator are being actively pursued[13, 12]. All data movements and information transfer in OCAPP are space-invariant which may render their implementation easier. Classical optical components such as lenses, mirrors, beam splitters, holographic deflectors, and delay elements are most likely to be used for this purpose[14]. In addition, halfwave plates, shutters, and masks may be used for dynamic routing.

4.2 A Modular Implementation of OCAPP

In this paper, we present a modest design example of OCAPP, using existing optical hardware in order to highlight the potential implementation issues of a practicable realization. The implementation of this first version will make use of the SEED device operating as a NOR gate for optical logic, and of the S-SEED device operating as a S-R latch for storage[11]. The NOR gate is preferable to any other form of thresholding nonlinearity because it only requires distinguishing between the state where no light comes in and the state where light come in. Thus the NOR gate requires an SNR better than one only. In addition, a NOR function constitutes a complete logic set capable of implementing any boolean or arithmetic function[15]. The family of SEED devices seem to be easy to use, capable of high speed, low energy operation, and can be fabricated in 2-D format. Space-invariant optical interconnects, dynamic masking components, and beam spreading and combining devices are assumed for data routing[16].

The S-SEED device has two inputs, S, R, and two outputs Q and $\bar{Q}$. The state of the device is set by a pair of unequal signal beams labeled S (for setting the output $Q = 1$, $\bar{Q} = 0$) and R (for resetting the output $Q = 0$, $\bar{Q} = 1$). The device is set (Q = 1) when the power incident on

the S input is much higher than the power incident on the R input. The state of the device is read by applying two equal-power (clock signal) beams to both inputs. During the setting of the device, the clock beams must be low, compared to the signal beams. The device holds its state when no clock signal is incident. Thus the device can operate as a latch. Moreover, during the application of the clock signal (the reading process) the state of the device is unaltered. As described earlier, the optical processor can be constructed from several units: the selection unit, the match/compare unit, the response unit, the output unit, and the control unit. In what follows, we describe the optical implementation (architectural rather than experimental setup) of each of these units. Moreover, the details in the routing and imaging paths such as lenses, holographic elements, masks, beam splitters, and polarizers have been omitted in this version to assist the reader's conceptual understanding of these configurations.

4.3 The Optical Selection Unit

The optical selection unit of Fig.4 is composed of a storage array which consists of a 2-D $n \times m$ array of clocked S-SEED devices (each entry in the array at position i, j has two incoming bits S, R and two outgoing bits w_{ij}, $\bar{w_{ij}}$), a $n \times 1$ word register A which serves at setting and resetting data words in the storage array, a $1 \times (m+1)$) bit-slice loading register B for loading a single bit-slice of the storage array. The first bit B_0 and its complement $\bar{B}_0$ are called set-E and reset-E respectively, since they are used for setting and resetting the $n \times 1$ enable register E which is used for the matching process (to be explained below). Memory words are disabled through the $n \times 1$ NOR gate array, representing the D register. The D register can be loaded from R, G or L registers. In addition, the E register can also be reset from the priority register P of the response unit (to be explained below).

A. Writing a Word/Bit-Slice into the Storage Array:

The storage array is assumed to be loaded in parallel at the beginning of the program. During program execution, the contents of the storage array can be altered by the use of the A and the B registers. To write a word in the storage array, say at word position i, the word is first written in the flip-flops of the B register. In the next clock cycle, the clock signals of B bits are pulsed high, and the contents of the B register is spread out vertically such that each bit B_j impinges on the set ports of the j-th column of the storage array. Next, bit A_i of A (corresponding to word position i) is pulsed high and spread out horizontally such that it impinges on the set/reset ports of the i-th row of the storage array. A one bit is written in bit position w_{ij} of the storage array if and only if a high A_i and a high B_j coincide at the set port of bit w_{ij}. Similarly, a zero bit is written in bit position w_{ij}, if and only if a high A_i and a high $\bar{B}_j$ coincide at the reset port of w_{ij}. This of course assumes that the set/reset thresholds of the S-SEED devices are so designed. Similar operations take place for writing a bit-slice in the j-th column of the storage array, with the exception of interchanging the roles of the B and A registers.

B. Enabling/Disabling Memory Words:

By enabling a memory word W_i, it is meant including it in the matching process. Similarly, by disabling it, it is meant excluding it from further matching operations. To allow a memory word W_i in participating in the matching process, its corresponding bit E_i in the E register must be set high. Similarly, to disallow a memory word w_i from participating in the matching process, its corresponding bit E_i in the E register must be made low. To enable/disable the entire memory words, the set-E/reset-E bits ($B_0/\bar{B}_0$) are spread out vertically and broadcast to all the set/reset ports of E. To selectively disable memory words whose R, or G or L bits are not asserted (R= 0, or G = 0 or L = 0), requires the routing of the appropriate register (R, G, or L) to the NOR gate array D. The output of D (which represents the complement of the routed register) is imaged onto the reset ports of register E. For example, to disable memory words whose R bits are not asserted (R = 0) from further matching operations, first contents of R is routed to D, which in turn image

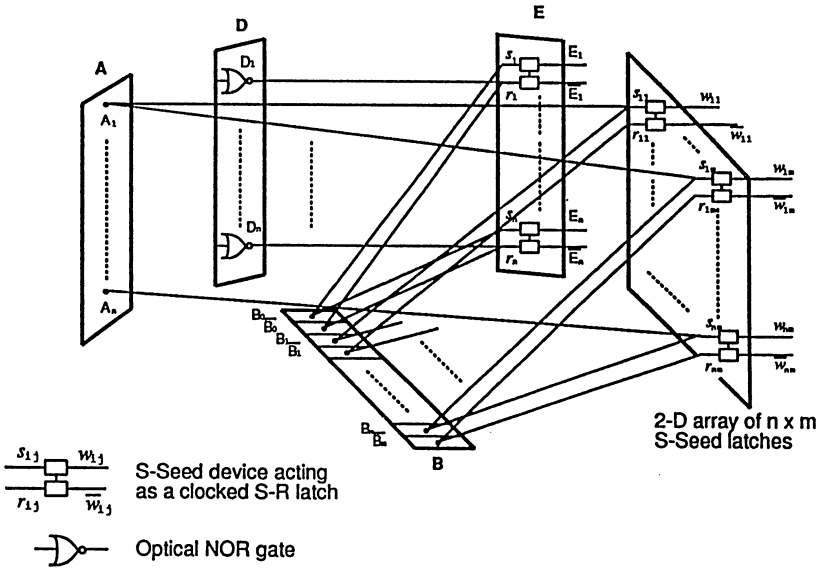


Figure 4 : Optical Implementation of the selection unit.

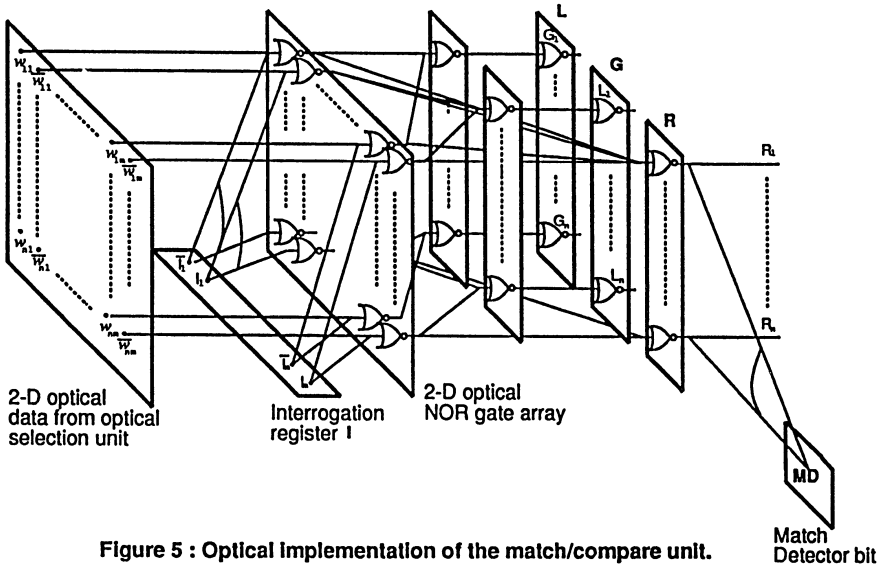


Figure 5 : Optical Implementation of the match/compare unit.

the complement of R onto the reset ports of E. Thus a low bit R_i of the R register will disable the i-th bit of the E register, which in turn disables memory word W_i from participating in further comparisons. The role of the E register in the match operation is explained next.

4.4 The Optical Match/Compare Unit

This unit performs exact match, and magnitude comparison searches between the interrogation register I and words of the storage array. As shown in Fig.5, It contains several SEED arrays operating as NOR gate arrays, and three registers, namely the response register, R, the greater than register G, and the less than register L. Parallel comparison takes place between memory words emanating from the storage array and the interrogation register I. A match at bit w_{ij} is detected by an exclusive-and principle as indicated in Eq.1. For that, register I needs to be spread out vertically so that each bit I_j impinges on one port of the NOR gates of the j-th column of the array, while data bits w_{ij} impinge on the second port of the NOR gates of the same j-th column. Matches between I_j and w_{ij} are reported in bit R_i of the R register. Otherwise, the G and L registers indicate the relative magnitude. The contents of R , G, and L are routed to the response unit as well as fed back to the selection unit.

As stated above, the enable register E determines whether or not a memory word participates in the matching process. Thus, the word match condition of Eq.2 is rewritten as follows:

$$R_i = [(I_{m+1} \wedge E_i) \vee (\bar{I_{m+1}} \wedge \bar{E_i})] \wedge [(I_m \wedge w_{im}) \vee (\bar{I_m} \wedge \bar{w_{im}})] \wedge, \ldots, \wedge [(I_1 \wedge w_{i1}) \vee (\bar{I_1} \wedge \bar{w_{i1}})] \quad (3)$$

where bit I_{m+1} is set to one ($\bar{I_{m+1}} = 0$) during a match operation. It can be seen from Eq.3 that a memory word W_i will participate in the match process if and only if its enable bit E_i is set to one. The R register bits are logically ORed to form the Match Detector (MD) bit. The MD flip-flop indicates if there is any match between I and memory words.

The optical match/compare unit of Fig.5 consists of a single interrogation register I, and therefore allows comparison of one search argument with the words stored in the storage array. However, using the multi-dimensionality of optical systems, this unit can be extended to perform multiple search operations in a single step. That is, several search arguments are compared simultaneously with the words of the storage array. An extended MIMD match/compare unit would have a $k \times m$ two-dimensional array of k search arguments, a two-dimensional storage array of n words each m bits long, and a $m \times k$ two-dimensional response array as shown in Fig.6. Each response register R_l ($l = 1, \ldots, k$) would indicate the match between interrogation register I_l ($l = 1, \ldots, k$) and the words of the storage array. The two-dimensional match operation can be thought of as an optical binary matrix-matrix multiplication which can be implemented using several optical techniques [17].

4.5 The Optical Response Unit

The response unit, contains a combinational priority circuit, and a priority register P for indicating the first matching word in memory (It may also contain few scratchpad registers for temporary storage). The priority circuit allows only the first responder (the first memory word W_i whose R_i is one) to pass to the priority register P. The priority circuit can be implemented using several stages of the NOR gate arrays in the form of a binary tree with space-invariant interconnections between them[18]. Contents of the P register are routed to the output unit, and also fed back to the selection unit.

4.6 The Optical Output Unit

The output unit outputs memory word whose corresponding bit in the priority register P, R, G or L is set to one (Fig.7). These latter registers are routed to a $n \times 1$ NOR gate array, denoted

by N in Fig.7, whose sole purpose is to invert their values. Each bit N_i of N is logically NORed with memory word W_i using a 2-D NOR gate array. Next, each column of the NOR gate array is logically ORed to form output bit O_i of the output register. This latter could be a photosensitive device which only detects the presence of light and outputs electrical signals, or a 1-D array of SEED devices acting as OR gates, and outputting optical signals. It should be noted that parallel readout of selected memory words is also achievable by replacing P with a 2-D output device and eliminating the OR function.

4.7 The Control Unit

OCAPP is under the control of a memory control unit which comprises a local memory for storing programs and a program sequencer for executing instructions that control the optical hardware such as the S-R latches, the NOR gate arrays, the routing shutters, and splitters, etc. The instruction set is composed of conventional assignment and conditional statements, and additional instructions required to implement associative parallel processing. This includes data movement between units, comparison operations, memory loading and unloading, monitoring the MD bit, etc. These additional instructions are very few in nature and are derived from the required fundamental operations described above. It should be noted that application programs for OCAPP can be written in conventional high-level languages such as Pascal or C, with few calls to external procedures which support parallel associative processing.

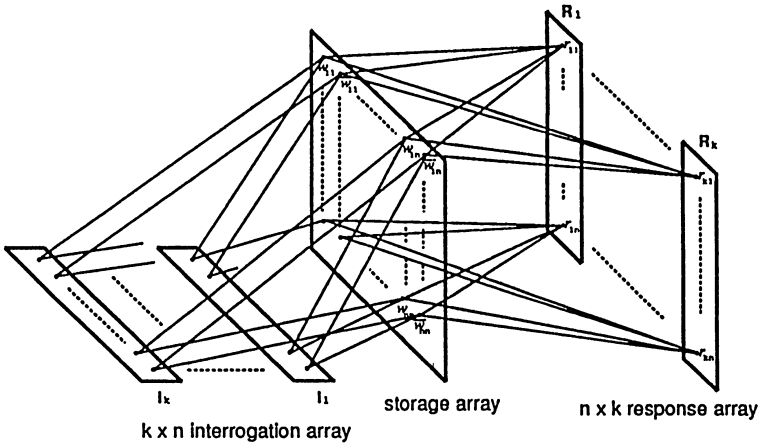
4.8 Estimated Execution time

An exact performance analysis of the proposed OCAPP including cost and power budget breakdown is currently not feasible due to the lack of optical S-R latches and thresholding devices with reasonable size (e.g., 500 × 500 gates), low operating energy, and fast response time. However, major efforts are being pursued[10, 11] in developing these devices in larger sizes. These efforts will soon culminate in the required devices and components for OCAPP as well as for other digital optical computing models.

We therefore try to theoretically estimate the execution time in terms of gate delay of the various basic search and arithmetic algorithms presented. This time does not include memory loading and unloading. We assume that the response times of the S-R latches (S-SEED) and that of the NOR gate arrays (SEED) are comparable and are both equal to T_s, and T_p is light propagation time through the processing loop (from the selection unit and back). It is assumed that the reading of memory words from the storage array and the I register is done at the same time, and takes one gate delay T_s; enabling/disabling of memory words is achieved in one gate delay; testing of the MD bit takes one gate delay, and the priority circuit takes $\log_2 n$ gate delays, where n is the number of words participating in the matching process.

Table 2 presents the estimated execution time of the algorithms presented. It can be seen that the execution time in equivalence search is a constant factor, and independent of the number of words in memory. The time in threshold search, double limits search, adjacency search, and extrema search, ordered retrieval (minimum time only) is proportional to the precision (number of bits) of the operands, and is independent of the number of words involved in the operation.

Note that the availability of the Match Detector (MD) bit provides major speed improvements to the above algorithms, in that certain conditions to terminate the computation as early as possible can be easily detected. Take for example, the threshold search algorithm. After the first comparison operation, if there are no words that match the comparand at that bit position (a condition that can be easily be detected by checking the MD bit), then all the words have been decided on in only $4T_s$ delay time, and the result is obtained in a much shorter time. The same considerations take place for double limits search, and adjacency search.



Register R_i indicates the match/mismatch of memory words with interrogation registers I_i(for i=1 to k).

Figure 6 : Optical implementation of a 2-D match/compare unit : the interrogation as well as the response registers of figure 5 are replaced by two-dimensional arrays of search arguments and response registers respectively.

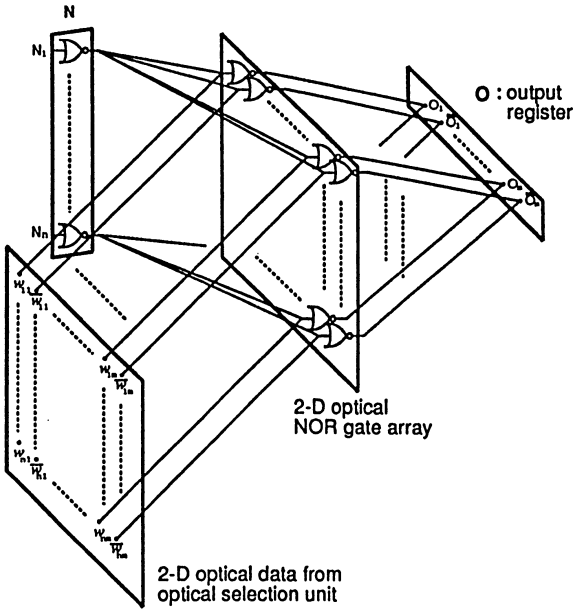


Figure 7 : Optical implementation of the output unit.

Table 2: Estimated execution time of the parallel algorithms on OCAPP

Search Algorithm	Minimun Execution Time	Maximum Execution Time
Equivalence Search	$3T_s$	$3T_s$
Threshold Search	$4T_s$	$(5T_s + T_p) \times m$
Minimum Search	$(6T_s + T_p) \times m$	$(6T_s + T_p) \times m$
Maximum Search	$(6T_s + T_p) \times m$	$(6T_s + T_p) \times m$
Double Limits Search	$10T_s + 2T_p$	$T_s + T_p + 2(5T_s + T_p) \times m$
Adjacency Search	$5T_s + T_p + (6T_s + T_p) \times m$	$T_s + T_p + (11T_s + 2T_p) \times m$
Ordered Retrieval Search	$((6T_s + T_p) \times m + T_p + 3T_s)$	$((6T_s + T_p) \times m + (\log_2 n)T_s + T_p + 3T_s) \times n$

The parameters m and n in the above table represent the word length and the number of operands respectively.

5 Conclusions

CAM-based processing has been argued to be the natural solution for non-numerical information processing applications. Unfortunately, the implementation requirements of these architectures using conventional electronic technology have been very cost prohibitive. This paper presented the principles and initial design concepts of an CAM-based parallel processing architecture that matches well with optics advantages, and therefore is highly amenable to optical implementation. The architecture relies heavily on the use of space-invariant interconnections, optical signal broadcasting and funneling (combining), and the simultaneous application of the same operation to many data points (SIMD mode of computing). The motivations behind this is the ease with which these operations can be realized with optics. A representative set of search algorithms have been presented to show the use and merits of the architecture. These algorithms are key components which occur in large computing tasks. It is important to note that these fundamental search algorithms are implemented on the optical architecture with an execution time independent of the problem size (the number of words to be processed). This indicates that the architecture would be best suited to applications where the number of data sets to be operated on is high. Some of the applications being investigated are: (1) real-time information retrieval, (2) database management, (3) knowledge-base and expert system implementation, and (4) list and string processing.

We presented a preliminary and simple version of an optical implementation of OCAPP. This version is only meant to show the feasibility of the architecture with existing optical devices. No optimization attempts were made. Nevertheless, this preliminary version reveals several key design issues that will determine the physical realization of such an optical architecture. Even if we assume the availability of optical nonlinear devices (latches, and NOR gates) in large sizes, the effective memory size will be critically determined by the beam spreading/combining optics, the contrast ratio and the fan-in/fan-out factors of the logic elements to be used. These practical implementations issues will be fully detailed in a follow-up paper.

References

[1] K. Hwang and D. Degroot, *Parallel Processing for Supercomputers and Artificial Intelligence*, McGraw-Hill, New York, 1988.

[2] T. Kohonen, *Content-addressable memories*, Springer-Verlag, 1980.

[3] A. A. Sawchuk and T. C. Stand, "Digital optical computing," *Proceedings of The IEEE*, vol. 72, no. 7, pp. 758–779, July 1984.

[4] W. T. Cathey, K. Wagner, and W. J. Miceli, "Digital computing with optics," *Proceedgins of the IEEE*, vol. 77, pp. 1558 – 1572, Oct. 1989.

[5] A. Louri, "A parallel architecture and algorithms for optical computing," *Optics Communications*, vol. 72, no. 1, pp. 27 – 37, July 1, 1989.

[6] A. Louri, "3-D optical architecture and data-parallel algorithms for massively parallel computing," *IEEE MICRO*, April 1991.

[7] B. K. Jenkins, P. Chavel, R. Forchheimer, A. A. Sawchuk, and T. C. Strand, "Architectural implications of a digital optical processor," *Applied Optics*, vol. 23, no. 19, , October 1984.

[8] J. W. Goodman, F. J. Leonberger, S. Y. Kung, and R. A. Athale, "Optical interconnections for VLSI systems," *Proceedings of the IEEE*, vol. 72, no. 7, pp. 850–866, July 1984.

[9] P. B. Berra, A. Ghafoor, M. Guizani, S. J. Marcinkowski, and P. A. Mitkas, "Optics and supercomputing," *Proceedings of the IEEE*, vol. 77, pp. 1797 – 1815, Dec. 1989.

[10] J. A. Neff, R. A. Athale, and S. H. Lee, "Two-dimensional spatial light modulators: a tutorial," *Proceedings of the IEEE*, vol. 78, pp. 836 – 855, May 1990.

[11] A. L. Lentine, H. S. Hinton, D. A. B. Miller, J. E. Henry, J. E. Cunningham, and L. M. F. Chirovsky, "Symmetric self-electrooptic effect device: optical set-reset latch, differential logic gate, and differential modulator/detector," *IEEE J. of Quantum Electron.*, vol. 25, pp. 1928 – 1936, Aug. 1989.

[12] J. L. Jewell, M. C. Rushford, and H. M. Gibbs, "Use of a single non-linear Fabry-Perot etalon as optical logic gate," *Appl. Phys. Lett.*, vol. 44, pp. 172 – 174, Jan. 1984.

[13] S. D. Smith, J. G. H. Mathew, M. R. Taghizadeh, A. C. Walker, B. S. Wherret, and A. Hendry, "Room temprature, visible wavelength optical bistability in ZnSe interference filters," *Optics Communications*, vol. 51, pp. 357 – 362, Oct. 1984.

[14] A. W. Lohmann, "What classical optics can do for the digital optical computer," *Applied Optics*, vol. 25, no. 10, pp. 1543 – 1549, 15 May 1986.

[15] A. Louri and K. Hwang, "A bit-plane architecture for optical computing with 2-d symbolic substitution algorithms," In *Proc. 15th Int'l. Symp. on Computer Arch.*, Honolulu, Hawaii, May 30 - June 4, 1988.

[16] K. Hwang and A. Louri, "Optical multiplication and division using modifed signed-digit symbolic substitution," *Optical Engineering, Special issue on Optical Computing*, vol. 28, no. 4, pp. 364 – 373, April 1989.

[17] R. A. Athale, "Optical matrix processors," *In Proc. SPIE, Optical and Hybrid Computing*, vol. 634, pp. 96 – 111, 1986.

[18] C. C. Foster, "Determination of priority in associative memories," *IEEE Transactions on Computers*, vol. C-17, pp. 788 – 789, Aug. 1968.

Towards an Efficient Hybrid Dataflow Architecture Model

Guang R. Gao† *Herbert H.J. Hum*‡ *Jean-Marc Monti*†

†McGill University
School of Computer Science
McConnell Engineering Building
3480 University St.
Montréal, Canada, H3A 2A7.

‡Centre de recherche informatique
de Montréal
3744 Jean Brillant, Bureau 500
Montréal, Canada, H3T 1P1.

Abstract

The dataflow model and control-flow model are generally viewed as two extremes of computation models on which a spectrum of architectures are based.

In this paper, we present a hybrid architecture model which employs conventional architecture techniques to achieve fast pipelined operation, while exploiting fine-grain parallelism by data-driven instruction scheduling. A mechanism for supporting concurrent operations of multiple instruction threads on the hybrid architecture model is presented and a compiling paradigm for *dataflow software pipelining* which efficiently exploits loop parallelism in loops is outlined. Simulation results attest that hybrid evaluations can indeed be beneficial.

1 Introduction

There have been two basic models in computer architectures: (1) the von Neumann sequential control model; and (2) the data-driven distributed control model. The parallel architectures based on the von Neumann model are aimed at exploiting coarse-grain parallelism, while the traditional dataflow architecture model was conceptualized to handle fine-grain parallelism. For the past decade, researchers have been debating on which model is a "better" basis for future large-scale parallel computer systems.

The work described in this paper is based on our view that the two models are not orthogonal, and that a flexible architecture model can be developed by extending the dataflow model to allow hybrid dataflow and control-flow evaluation. As a result, the grain-size of the parallelism supported by such a hybrid parallel architecture model is "flexible" – compiler and hardware techniques can be combined to "tune" the effective grain size for the needs of efficient exploitation.

We believe that the dataflow model of computation offers a sound, simple, yet powerful model of parallel computation. In the dataflow programming and architectures, there

is no notion of a single point or locus of control. Dataflow architectures have promised solutions addressing the two fundamental problems of von Neumann computers in multiprocessing: the memory latency and synchronization overhead [2]. However, we must not ignore the efficiency and simplicity of the instruction sequencing mechanism in von Neumann architecture models, as well as over 40 years of optimizations in the instruction execution mechanism. We have compared dataflow architectures designed as direct execution engines for dataflow graphs and those that perform the execution of dataflow graphs using features of conventional von Neumann computer architectures. We believe that the latter has the potential of offering a better performance/cost ratio.

In this paper, we develop a framework for a hybrid architecture model. The essential elements of such a framework must consist of both a simple architecture model and an effective compiling methodology which can structure code to expose parallelism for hybrid evaluation. Our application domain is general-purpose scientific computations where a program usually consists of a number of *code blocks* (or loops) which define the major array values for the computation. A compiling paradigm is established which exploits loop parallelism through *dataflow software pipelining*. We establish a set of basic results which show that the fine-grain parallelism in a loop can be fully exploited by a a simple scheduling scheme, achieving time and space efficiency simultaneously.

In Section 2, we describe one processing element of the McGill Dataflow Architecture Model (MDFA) which employs the *argument-fetching principle*. We then describe the basic architecture mechanisms for supporting concurrent (recursive) function invocations. For multiprocessing support, an efficient inter-processor synchronization and communication mechanism is described for sending and receiving data through an interconnection network. In Section 3, we present a hybrid evaluation model based on the MDFA which is structured around the notion of instruction threads, i.e., instructions in each thread are executed sequentially, while multiple (sequential) instruction threads can operate concurrently through a data-driven style instruction scheduling mechanism. In Section 4, we present the principle of dataflow software pipelining and the limited balancing technique – a technique that can be used to obtain instruction threads (or "macro" dataflow actors) for the hybrid model. Simulation results which attest to the need for hybrid evaluation are briefly discussed in Section 5. However, a major assumption made for the simulation runs has created a need to further extend the hybrid model so that executions can be performed more efficiently. We outline these other extensions in section 5.3. Finally, conclusions and future work are briefly outlined in the last section.

2 The McGill Dataflow Architecture Model

In this section, we introduce the operational model of the McGill Dataflow Architecture.

The operational semantics of dataflow programs for this model are defined in terms of familiar concepts used in conventional architectures. This architecture is based on the *argument-fetching* principle [5], where the instruction scheduling is decoupled from the main execution datapath, yielding a unique dataflow model which makes the extension to hybrid evaluation straightforward.

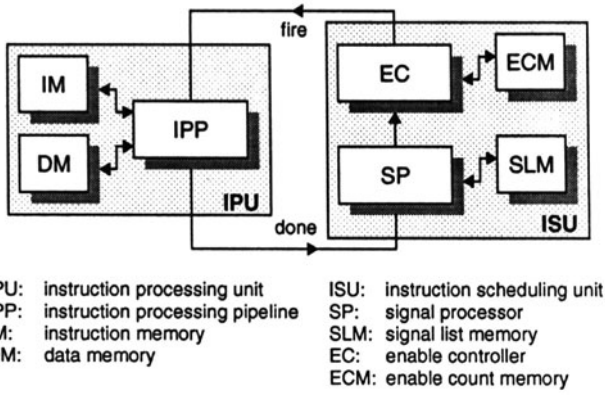


Figure 1: The McGill Dataflow Architecture Model.

2.1 The Architecture and the Program Tuples

The computation in our dataflow model is specified by a dataflow program and its operational semantics. A dataflow program is a tuple $\{P, S\}$, where the P portion, called P-code, is a set of 3-address instructions, and the S portion, called S-code, is a directed graph named the *signal graph*.

The architecture consists of an instruction processing unit (IPU) and an instruction scheduling unit (ISU) as shown in Figure 1. A three-address instruction (called a p-instruction or p-node) in the P-code is similar to that in a conventional architecture, and is stored in the IM. The operands and results of instructions are stored in the DM. For example, in a multiplication instruction $<$ times a b c $>$, the identifiers a and b are the memory addresses of the two operands and c is the address of the result. The storage model of the DM will be introduced shortly. Each p-instruction is uniquely identifiable by a p-instruction address in the IPU, pointing to its program memory in IM. In Figure 2, we show an example of a program tuple.

When a p-instruction address, say i, is presented to the "fire" input (we call this the *fire signal*), it is executed by the IPU in a fashion similar to that of any von Neumann style architecture, i.e., it goes through the usual stages of instruction fetch, operand fetch, execution, result store, etc., and accesses the appropriate memories accordingly. The major difference is that after the execution is completed, the IPU delivers the p-instruction address (in this case, i) to the "done" link as the *done signal*.

The scheduling of p-instruction execution is performed by the processing of S-code in the ISU. S-code consists of a set of nodes (named s-instructions or s-nodes), interconnected by directed arcs. Each s-node has a status field with the information required for scheduling: an *enable count*, indicating how many signals are yet to be received for the s-instruction to become enabled, and a *reset count* indicating the total number of signals it requires to become enabled again after the s-instruction is executed. The status field of the s-nodes are stored in the *enable count memory* (ECM) of the ISU. Each s-node is identified by an s-node address (or s-instruction address). For the purpose of this paper, we assume that there is a one-to-one correspondence between each s-instruction address

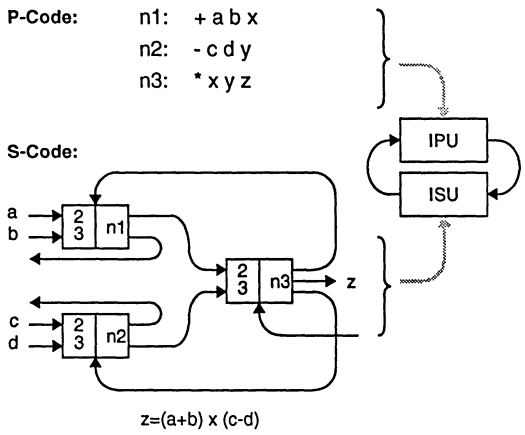


Figure 2: A Dataflow Program Tuple.

and each p-instruction address.

In reality, there are different ways the graph structure in S-code can be specified. For the purpose of this paper we assume that each s-node contains a list of s-node addresses as the destinations of its output arcs. This list is stored in the *signal list memory* (SLM) within the ISU. The principal function of S-code will be described shortly.

2.2 Run-time Storage Model

One of the important features of the architecture is its run-time storage model. In dataflow computations, it is perfectly legal (and also desirable) to have multiple function invocations active concurrently. Therefore, the stack model in a conventional architecture is not appropriate: there is no single point of control corresponding to the "top of stack".

A program tuple is often structured as multiple code modules, i.e., named function modules. Each function module corresponds to code generated from functions in a high level language program. A function module can be shared by many invocations of the function. To support multiple invocations, the memories in the IPU and the ISU are organized to support dynamic allocation of frames of locations, one for each invocation. A frame is also called a *function overlay* in [8]. Although an overlay appears to play a similar role to that of a stack frame in conventional processors, the run-time structure of the overlays in the memory is very different.

An overlay is composed of an IPU overlay and an ISU overlay. For each invocation of a function f, a data memory overlay is allocated to store the result values of each p-instruction composing the function body. The enable status fields of all s-nodes of a particular invocation of f are stored in f's corresponding ISU overlay in ECM. A value stored in a function overlay can be accessed through an address consisting of two parts: a base address of the overlay and an offset. For the purpose of this paper, we assume that the base address is generated and managed (together with memory overlays) by the run-time storage mechanism. This is done through the execution of *apply* and *return* operators for function invocations described in [8].

2.3 The Operational Model

Now, we can present the operational semantics of our dataflow model by the following *firing rules* of a dataflow program tuple:

- a p-instruction i is *enabled* if its S-code counterpart, an s-instruction i in S-code becomes *enabled*, i.e., it has received all the signals it needs;
- an enabled p-instruction may be selected to fire when a fire signal $< i, b >$ (where i is the p-instruction address and b is the base address of the corresponding function overlay) is generated;
- an enabled p-instruction i is processed (or *fired*) in the IPU by accessing the values stored in its function overlay using the base address b; a result value may be generated and stored in a location within the function overlay (note that in an unoptimized form, one location is allocated to each p-instruction);
- after the p-instruction processing is completed, a done signal $< i, b >$ is generated, signaling the completion of the firing of the p-instruction; and
- a scheduling phase (S-code processing) is activated by a done signal $< i, b >$; an s-instruction i is processed by sending signals to all s-instructions in its output signal list, and the status fields (referenced through the base address b) of these s-instructions are updated accordingly. If an s-node has received all the signals it needs, it will become enabled and its enable status will be reset accordingly.

The acute readers may recall the classical formulation of the firing rules based on token flow in a dataflow graph as described in [4]. Although our firing rules do not introduce the concept of tokens, they remain equivalent to the classical firing rules. It is beyond the scope of this paper to present a proof, but interested readers may convince themselves by considering the following hint: an actor in a dataflow graph can be mapped into a pair of p-instruction and s-instruction, and the classical firing rules can be implemented by the firing rules above.

Furthermore, it should be possible to extend the concept of well-behaved dataflow graphs [4] and its translation rules from high-level languages for this model.

2.4 Interprocessor Communication Supports

Thus far, we have only described one processing element of the MDFA. For multiprocessing support, we introduce an Interprocessor Communications Unit (ICU) [16]. Figure 3 shows a schematic block diagram of the ICU and the MDFA within a multiprocessor system.

In this section, we present an efficient interprocessor synchronization method to allow two data-dependent nodes residing in different processing elements (PEs) to synchronize their execution. Also, since dataflow architectures can effectively tolerate long latency memory operations, we have provided the system with a shared global memory, physically distributed among the PEs.

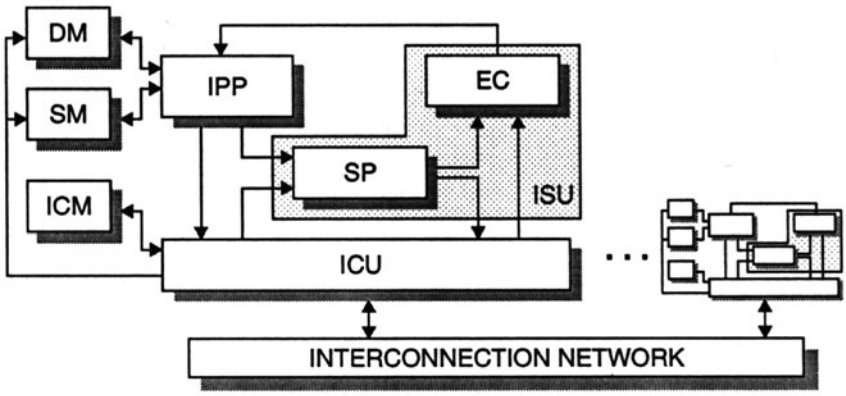


Figure 3: The MDFA in a multiprocessor context.

2.4.1 The Interprocessor Communication Unit

An important issue in the synchronization of remote events is the reduction of network traffic and one of the most effective ways of reducing network traffic is by pairing signals and data in the same packet whenever possible. Also, we believe that it is more efficient to implement interprocessor communications by using attributed elements in the signal list memory instead of special "send" and "receive" actors. Hence, in our scheme, we introduce one more kind of signal list elements: those tagged with *ip-count*. Regular signal list elements send their count signals to the EC, while the ones tagged with ip-count trigger the ICU to send interprocessor count signals. There are two kinds of ip-counts: *data-count* signals, which signifies that data has to be transmitted along with the signal, and *remote-count* which only convey the signal. The remote specifications of the target node as well as the type of packet that has to be sent onto the network are stored in the interprocessor communication memory (ICM). The following are the packet structures for the data-count and remote-count signals:

$$<\text{data-count}, PE_{rem}, \text{s-node}_{rem}, \text{dm\_add}_{rem}, \text{data\_value}>$$
$$<\text{remote-count}, PE_{rem}, \text{s-node}_{rem}>$$

where PE_{rem} is the remote PE address, s-node_{rem} is the target node address (within PE_{rem}), dm_add_{rem} is the address where the data_value has to be stored in the remote DM.

At the remote PE, reception of ip-signals is straightforward. For a data-count signal, the data conveyed in the packet is stored into DM before the count signal is sent to the EC unit while a remote-count ip-signal just sends a count signal to the EC.

2.5 A Summary of Distinct Features of the MDFA

The features of the MDFA in comparison with other proposed dataflow architecture models can be summarized as follows:

- *Elimination of instruction pipeline gaps due to operand matching.* The data-driven instruction scheduling mechanism is clearly separated from the instruction processing unit. Thus, the MDFA does not perform token "matching" in the critical instruction execution datapath, either explicitly using a *matching store* as in [1, 11] or implicitly at the frame slot with the direct token addressing scheme as in [18]. This eliminates pipeline "gaps" due to operand matching (in dyadic operators), provided that the ISU can handle the signals generated by IPU to "hide" the matching cost from the IPU. (The tradeoff will be elaborated later).

- *Avoiding token duplication.* In the MDFA, a result value never needs to be duplicated (copied) and routed to the input "arcs" of destination nodes. It is stored in the operand memory (only once when it is produced), and will be directly fetched when the subsequent instruction execution needs it as an operand – in a manner similar to any conventional architecture.

- *Efficient dataflow software pipelining of loops.* This is the subject of a more detailed discussion in later sections.

The MDFA supports multiple recursive function invocations, in this sense the model has the power of dynamic dataflow architecture models [1]. However, the MDFA treats loops differently from the loop unraveling scheme used in most dynamic dataflow models [1]. It retains the simplicity of the static dataflow model by limiting the number of concurrent activations of an instruction (to one instance per instruction) and multiple initiations of a loop body can be concurrently executed through dataflow software pipelining. In Section 4, we will comment on the relation between our scheme and the *loop bounding* scheme proposed as a practical realization for the loop unraveling scheme [3].

3 Hybrid Evaluation

In this section, we start with the operational model of the MDFA model and show how it can be easily extended to support sequential instruction execution.

3.1 Support of Sequential Instruction Execution on the MDFA Model

Although the processing of a p-instruction in the argument-fetching dataflow model is similar to that in a von Neumann architecture, there is one missing feature – the sequencing mechanism of instruction execution. In the control-flow model of computation, this mechanism implies that: 1) a program counter is maintained to contain the unique address of the currently executable instruction; 2) the program counter is updated during an instruction execution by default (i.e., for instructions other than branch instructions) – it will be incremented by one – automatically pointing to the next instruction in the sequence; and 3) only branch instructions may update the program counter with a destination address possibly different from the default value.

There is little doubt that the program counter based control mechanism is simple and effective for sequential instruction processing. It turns out that the extension of the basic MDFA to support hybrid evaluation is very simple. All that needs to be done is to provide an alternative for the instruction continuation, after the execution phase of a p-instruction is completed. That is, if desirable, the IPU can directly generate the next p-instruction address to be executed, instead of going through the scheduling phase in the ISU. Of course, this implies that some notion equivalent to the program counter of the control-flow model must be introduced in the dataflow model. In fact, we need a number of program counters: one for each instruction sequence.

To implement this hybrid model, each p-instruction is extended to carry an extra field – a tag field (called a *von Neumann bit*) which indicates whether the instruction is following a dataflow style scheduling or a von Neumann style scheduling. For example, a multiplication instruction now has the following format:

times a b c v-tag

where the *v-tag* is the tag field.

Depending on the value of the v-tag field, the instruction is either scheduled in dataflow mode (*D-mode*) or von Neumann mode (*V-mode*). If v-tag = 0 (dataflow mode), the done signal is generated as usual, i.e., it contains the address of the p-instruction so that it will be processed by the ISU. Otherwise, a new address is generated as the address of the next p-instruction to be fired, in a way similar to the update of the program counter in a von Neumann architecture. Note that this is different from the *repeat-on-input* mechanism in the Epsilon dataflow processor which was proposed to exploit the locality inherent in parameter duplication [10].

3.2 The Hybrid Operational Model

Since in the basic model, a p-instruction is processed by the IPU in a style similar to von Neumann style processing, the above hybrid evaluation model can be implemented in a straightforward fashion. The only change to be made is to allow a "short-cut" path from the done link to the fire link, thus allowing a V-mode p-instruction to directly "fire" the next instruction, bypassing the scheduling phase (ISU). In Figure 4, we illustrate how a sequence of dataflow nodes can be grouped into a "macro" dataflow node, and a short-cut signal mechanism is used for their sequencing.

The operational model for hybrid evaluation becomes:

- a p-instruction i is *enabled* if
  - its S-code counterpart, an s-instruction i in S-code becomes *enabled*, i.e., it has received all the signals it needs;
  - or, it is the next instruction in a sequence of V-mode p-instructions, and its predecessor p-instruction $(i-1)$ has completed its execution;
- An enabled p-instruction may be selected to fire when a fire signal $< i, b >$ (where i is the p-instruction address and b is the base address of the corresponding function overlay) is generated;

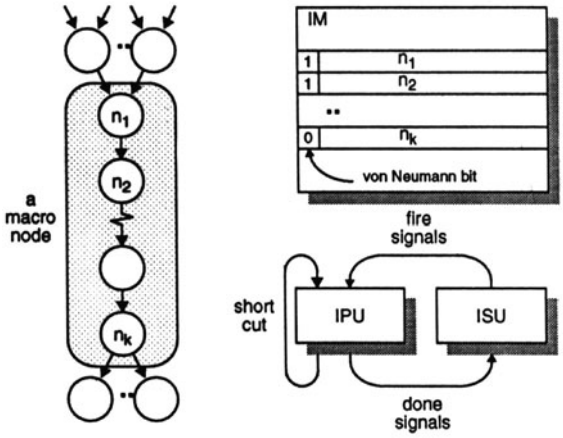


Figure 4: The Hybrid Evaluation Model.

- An enabled p-instruction i is processed (or *fired*) in the IPU by accessing the values stored in its function overlay using the base address b.
- After the p-instruction is fired and depending on its mode, one of the following happens:
  - for a V-mode instruction (vtag = 1), a fire signal is sent through the short-cut path, which contains the next p-instruction address; and
  - for a D-mode instruction, a done signal $< i, b >$ is generated, signaling the completion of the firing of the p-instruction and the start of the ISU scheduling phase below.
- a scheduling phase (S-code processing) is activated by a done signal $< i, b >$; an s-instruction i is processed by sending signals to all s-instructions in its output signal list, and the status fields (referenced through the base address b) of these s-instructions are updated accordingly. If an s-node has received all the signals it needs, it will become enabled and its enable status will be reset accordingly.

3.3 Features of the Hybrid MDFA Model

We briefly summarize the features of the hybrid MDFA model:

- *Generality*: The hybrid MDFA model supports both thread level and instruction level parallelism through efficient fine-grain synchronization. At any time, the IPU can execute several instructions in parallel: any instruction may be a D-mode instruction or a V-mode instruction, and the V-mode instructions may themselves come from different instruction threads. Thus our model is different from so-called "macro-dataflow" schemes where dataflow scheduling can only be done at the inter-procedural level [14]. It retains the advantage of dataflow models in terms of dealing

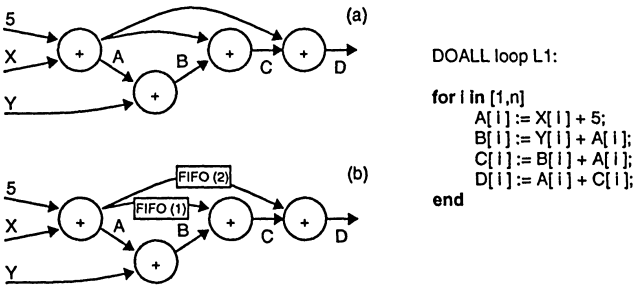


Figure 5: Example of Dataflow software pipelining

with the two fundamental issues of von Neumann multiprocessing as discussed in the introduction.

- *Flexibility*: There is no restrictions to the number of instruction threads which can be supported by this model. In fact, a variable number of instruction threads each with a different size can be active concurrently. This is different from some other multi-threaded architectures where the maximum number of threads are fixed *a priori* such as in the HEP [19].

- *Simplicity*: Under the hybrid MDFA model, any instruction in a program can be set to one of two modes, regardless of its function or type. This flexibility certainly makes the job of a compiler easier, since the mode control and the operation of an instruction become orthogonal. Note that ISU performs the fork/join operations of the threads implicitly through signal processing, while some other multi-threaded architectures may execute explicit fork and join instructions [17].

4 Dataflow Software Pipelining

In this section, we present a compiling paradigm which exploits loop parallelism through *dataflow software pipelining* using *limited balancing*. The limited balancing technique is one method which can be employed to obtain macro dataflow nodes for the hybrid architecture model.

4.1 Background

The principle of dataflow software pipelining can be explained using loop L1 as an example (in fig. 5). The loop is translated into a pipelined dataflow graph as shown in figure 5 (a). Successive waves of elements of the input arrays X and Y will be fetched and fed into the dataflow graph, so that the computation may proceed in a pipelined fashion. This is called *dataflow software pipelining*: the arcs drawn between actors correspond to addresses in stored dataflow machine code, not to the wired connections between logic elements.

Previous work in dataflow software pipelining are based on an idealized dataflow machine model. Here is a summary of some known results which are valid under the ideal

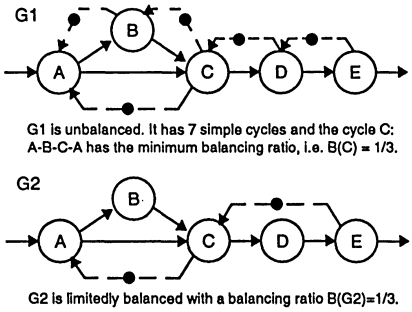


Figure 6: Balancing ratio for some example graphs.

model. A graph is (fully) *balanced* if every path between a pair of nodes contain exactly the same number of actors. To achieve maximum pipelining, a basic technique (called *balancing*) is used to transform an unbalanced dataflow graph into a (fully) balanced graph by introducing FIFO buffers on certain arcs.

For example, the dataflow graph in figure 5 (a) is not balanced We can introduce two FIFO buffers as shown in figure 5 (b) so the resultant graph will become fully balanced, and hence can be maximally pipelined in an ideal dataflow machine model.

Unfortunately, the previous work does not cover the case where dataflow graphs contain cycles. We will discuss an extension to include these loops in the next section. More about dataflow software pipelining can be found in [6].

4.2 Limited Balancing

A loop may contain loop-carried dependencies, in which cycles may exist with some back edges due to inter-iteration data dependencies.

Let us introduce the notion of *balancing ratio*, denoted by $B(G)$, to describe the degree of balancing of a dataflow graph G. We give an informal definition of the balancing ratio based on the observation of token and space duality in dataflow graph models suggested by Kung et. al. [15]. Let us augment G with *acknowledgment arcs* and assign one initial (dummy) token on each acknowledgment arc in G, denoting the fact that the corresponding forward (data) arc has one empty place – the SDFP is initially empty under our assumptions. Then our (augmented) SDFP is similar to Kung's *augmented dataflow graph* where the acknowledgment arcs correspond to his "reverse arcs". For any directed simple cycle C in G, [1] let the length of C (number of arcs) be K_C, and the number of dummy tokens be D_C. The ratio $\frac{D_C}{K_C}$, called the balancing ratio of C (denoted by $B(C)$) in this paper, determines the fastest rate at which the node in the cycle can be activated [15]. A graph is *(limitedly) balanced* with balancing ratio $B(G)$ if all directed cycles C_i in the graph have the same balancing ratio, i.e., $B(C_i) = B(G)$ for all i.

Figure 6 gives an example to illustrate the notion of limited balancing and the concept of balancing factor. It is beyond the scope of this paper to present the algorithm for limited

[1] Simple cycles are cycles in which each node appears at most once. For our purposes, only simple cycles are considered.

graph balancing. Readers may find a dedicated discussion in [9]. The basic intuition is that adding buffers as well as removing redundant signals (both data and acknowledgment arcs) may be required during the limited balancing process. In our simple example in figure 6, the nodes A, B, C can be grouped into a thread, so can D, E, F. This can have the potential of significantly reducing the number of signals to be handled by the ISU in dataflow software pipelining. Note that actors grouped into threads will form a macro dataflow actor for hybrid evaluation. We will demonstrate this advantage in the simulation section.

4.3 Related Work

There have been several different suggestions as how to compile von Neumann instruction threads for a hybrid computing model [13, 20]. In one interesting approach proposed by Iannucci [13], a compiler is responsible for partitioning program graphs into *scheduling quantums*. A methodology is outlined which will generate multiple scheduling quantums without deadlock. His method is tightly-coupled with the hybrid architecture described in his doctoral dissertation [13]. Another interesting approach proposed by Traub is to view partitioning of a program into threads as the central problem (or in his words: "first order business") of functional language compilation [20]. As a result of his "compiling-as-partitioning" strategy, one gains a much cleaner understanding of the relationships between lazy vs. lenient evaluation, as well as sequential vs. parallel execution. These techniques can also be applied to the architecture described in this paper.

5 Simulation Results

5.1 The Impact of ISU Signal Processing Capacity

In this section, we first present some simulation results to show that reducing the number of signals to be handled by the ISU is very important, thus motivating us to study the limited balancing technique.

In a balanced design of the MDFA, the ISU has to supply fire signals to the IPU just fast enough to keep the instruction execution pipelines operating continuously at full capacity. There could be cases in which the signal processing demand in the ISU cannot be satisfied fast enough, and this could cause a bottleneck.

Figure 7 shows the performance of Livermore Loop 7 using various machine configurations. When the signal processing capacity is one, the execution time of loop7 remains the same, no matter how many IPUs are used. In this case the computation is *ISU bound*, that is, the ISU is unable to satisfy the demand for manipulating incoming signals, causing a performance degradation. As the signal processing capacity of the ISU is increased, more enabled actors are presented to the IPUs per cycle, and thus, optimum processing is achieved, as shown by the leveling off of run times for the various numbers of IPUs modeled within a PE. From the shape of the curves we postulated that there is a relationship between signal processing capacity and number of IPUs needed to obtain optimum cost-effective program execution. Let us examine this phenomenon in more detail. To

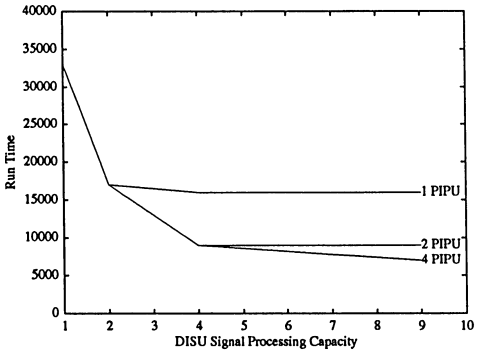


Figure 7: Varying machine configurations for loop7.

achieve such optimality, the following factors are important:

1. IPU capacity (denoted by P) must match the amount of parallelism within the program (*computational parallelism*), and
2. ISU signal capacity (denoted by C) must match the demand for processing the signals required to exploit the computational parallelism (*synchronization requirement*).

Our studies show that, even for a program having a computational parallelism sufficiently high to keep the IPU filled in an idealized ISU (one having infinite signal processing capacity), performance on a realistic machine might be far below that of the ideal value; the outcome depends upon the average number of signals S that are needed to fire an instruction. This number, called the *average signal density*, is given by the following formula:

$$S = \frac{TotalCountSignals}{TotalInstructionsExecuted}$$

So for a given machine configuration, the condition to keep the IPU pipeline usefully busy is:

$$\frac{C}{S} \geq P \tag{1}$$

Conceptually this means that to fully exploit the computational parallelism of a program, the ISU capacity should be at least equal to the product of the average signal density of the program and the number of IPUs of the given configuration. This is not surprising; *fine-grain parallelism has its price*, and for our architecture, the ISU pays.

Both the ISU capacity and IPU capacity can be taken as important parameters in compiler optimizations for the McGill dataflow architecture. In a machine in which signal traffic plays such an important role, the compiler should try to minimize the number of signals, while at the same time, trying to expose sufficient parallelism. This is one of the main motivations behind limited balancing.

IPU: 1, ISU Signal capacity: ∞, Pipeline Stages: 8 l: 8, n: 60, CB(G) = $\frac{1}{7.5}$						
Balance Factor (B)	$\frac{1}{2}$	$\frac{1}{3}$	$\frac{1}{4}$	$\frac{1}{5}$	$\frac{1}{6}$	$\frac{1}{7}$
Avg. Init. Cycle	60	60	60	60	60	66
Avg. Instr. Delay	11.08	11.14	6.34	3.68	1.73	1.28
Run Time	12,119	12,113	12,096	12,102	12,114	13,305
Utilization	99.1%	99.2%	99.3%	99.3%	99.2%	90.3%

Table 1: Simulation results for 1 IPU

5.2 The Effects of Limited Balancing

In this section, we demonstrate the effect of limited balancing. In each simulation run, the limited balancing has only been applied to one simple cycle of the graph in order to ease the job of manipulating the large number of signal arcs manually.

Table 1 shows the simulation results for the machine configuration of 1 pipeline (1 IPU) with 8 pipelined stages. The top row of the table indicates the associated machine configuration and the size of the graph n. The program graph does not have to be fully balanced to keep the IPU fully utilized. Instead, a *critical balancing ratio CB(G)* can be estimated based on the graph structure [7], which can be used to balance the graph. Each column of the table gives the corresponding results of applying a distinct balancing ratio (B) to the graph. The *Average Initiation Cycle* records the average pipelining period of each node in the graph; the average initiation rate of the graph is simply its reciprocal. The *Average Instruction Delay* records the average time delay of the enabled instruction waiting in the fire queue, which implicitly represents the amount of parallelism exposed in the graph that have not been fully utilized by the execution pipe.

Here is a summary of the major observations from table 1:

- In all simulation runs, we have observed that the loop initiation sequence will enter a steady state: a constant average initiate rate is reached.
- Once the average initiation rate *1/n* is achieved, the pipeline is maximally utilized.
- The critical balancing ratio for this loop is 1/7.5. Our simulation results are very close to this prediction. At B = 1/6, the pipeline is near fully utilized.
- The average instruction delay decreases gradually as the balancing ratio decreases. When the balancing ratio reaches the critical value, no excessive parallelism is exposed which may not be efficiently utilized, and the average instruction delay approaches its minimum.

The limited balancing scheme will reduce the amount of count signals to be handled by the ISU. In this case, if the limited balancing is applied to the entire graph, the total number of signal arcs can be reduced considerably.

5.3 Discussions

As we have shown, applying fine-grain synchronization to all actors in a dataflow graph does not necessarily yield optimal performance due to the fine-grain synchronization costs which must be absorbed by the synchronization mechanism in the underlying architecture. If there are enough parallelisms in the application, limiting the amount of exposed parallelism to the underlying machine by means of grouping actors into aggregates and then scheduling the aggregates themselves can lead to lower fine-grain synchronization costs and thus a more balanced use of the underlying processing and scheduling resources. This is fine, but in all the simulation runs, we have assumed that memory latencies are unity and as we all know, memory latencies in an actual implementation can never be kept fixed and low, especially not unity.

To address this problem, memory hierarchies must be investigated where fast and small memory like register files and conventional caches are utilized. However, in a multi-threaded architecture, lifetime analysis for register values are difficult, thus rendering registers less than effective. Moreover, the frequent switching between active threads wreaks havoc on the locality principle on which the conventional cache obtains its power, so caches are not ideal. In a paper appearing in the same conference [12], we introduce the *Register-Cache* mechanism which is a hybrid register file and cache mechanism. Register allocations are performed dynamically at run-time, so lifetime analysis of values are not necessary, and furthermore, all required values are loaded into the register-cache before a thread executes so that the execution pipe will see a fixed and low memory latency time. This register-cache mechanism is employed in the context of an extended MDFA model called the *Super-Actor Machine*, and readers are referred to [12] for more details.

6 Conclusion

We have presented an operational model for a hybrid dataflow and control-flow architecture. Based on the argument-fetching dataflow principles, we have demonstrated that a straightforward extension to the basic McGill dataflow architecture model can accommodate concurrent operations of multiple instruction threads. The flexibility of the new model not only resides in the support of hybrid evaluation, but also in exploring parallelism at any desirable level in a fine-grain fashion. A unique program mapping scheme for loops based dataflow software pipelining and the limited balancing technique was also presented. To take memory latencies into consideration, the hybrid MDFA model presented in this paper is further extended to be the *Super-Actor Machine* [12].

At McGill University, we are using the MDFA model and its variants as vehicles for studying a range of architectural issues for multi-threaded architectures, as well as compiling techniques for them. It is our plan to conduct a more extensive analysis and comparison of the advantages and disadvantages between the architectural models of the MDFA genre and other hybrid dataflow architecture models. We are also investigating the impacts of the hybrid model on language designs for multi-threaded computing.

7 Acknowledgment

We would like to thank the National Science and Engineering Research Council (NSERC) for their support of this work. Thanks also to the Bell Northern Research (BNR) for their support of research in parallel processing and dataflow.

We would also like to thank the members of the Advanced Architecture and Program Structures Group for joining us in the adventure of dataflow research. In particular, we thank René Tio, Robert Yates, Zaharias Paraskevas, Yue-Bong Wong, and Russel Olsen for working on the architecture, compiler, and simulation testbed for the McGill Dataflow Architecture. Without their support, the work described in this paper would not have been possible. Finally, we would like to thank J.B. Dennis for his valuable contributions.

References

[1] Arvind and D. E. Culler. Dataflow architectures. *Annual Reviews in Computer Science*, 1:225–253, 1986.

[2] Arvind and Robert A. Iannucci. Two fundamental issues in multiprocessing. In *Parallel Computing in Science and Engineering*, pages 61–88. Springer-Verlag, LNCS-295, 1987. *Proceedings of the 4th International DFVLR Seminar on Foundations of Engineering Sciences*, Bonn, June 1987.

[3] D. E. Culler. Managing parallelism and resources in scientific dataflow programs, Ph.D thesis. Technical Report TR-446, Laboratory for Computer Science, MIT, 1989.

[4] J. B. Dennis. First version of a data-flow procedure language. In *Proceedings of the Colloque sur la Programmation*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springler-Verlag, 1974.

[5] J. B. Dennis and G. R. Gao. An efficient pipelined dataflow processor architecture. In *Joint Conference on Supercomputing*, pages 368–373, Florida, November 1988. IEEE Computer Society and ACM SIGARCH.

[6] G. R. Gao. *A Code Mapping Scheme for Dataflow Software Pipelining.* Kluwer Academic Publishers, Boston, December 1990.

[7] G. R. Gao, H. H. J. Hum, and Y. B. Wong. An efficient scheme for fine-grain software pipelining. In *Proceedings of the CONPAR '90–VAPP IV Conference*, Zurich, September 1990.

[8] G. R. Gao, H. H. J. Hum, and Y. B. Wong. Parallel function invocation in a dynamic argument-fetching dataflow architecture. In *Proceedings of the PARBASE '90 Conference*, Miami Beach, FL, March 1990.

[9] G.R. Gao and Qi Ning. Loop storage optimization for dataflow machines. ACAPS Technical Memo 23, School of Computer Science, McGill University, Montréal, Qué., February 1991. In preparation.

[10] V. G. Grafe, G. S. Davidson, J. E. Hoch, and V. P. Holmes. The Epsilon dataflow processor. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 36–45, Israel, June 1989.

[11] J. R. Gurd, C. C. Kirkham, and I. Watson. The Manchester prototype dataflow computer. *Communications of the ACM*, 28(1):34–52, January 1985.

[12] H. H. J. Hum and G. R. Gao. A novel high-speed memory organization for fine-grain multi-thread computing. In *in the same Proceedings*, June 1991.

[13] R. A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140. ACM, June 1988.

[14] D. Kuck, E. Davidson, D. Lawrie, and A. Sameh. Parallel supercomputing today and the cedar approach. *Science Magazine*, 231:967–974, February 1986.

[15] S. Y. Kung, S. C. Lo, and P. S. Lewis. Timing analysis and optimization of VLSI data flow arrays. In *Proceedings of the 1986 International Conference on Parallel Processing*, 1986.

[16] J.M. Monti. Interprocessor communication supports for a multiprocessor dataflow machine. Master's thesis, School of Computer Science, McGill University, Montréal, Qué, March 1991.

[17] R. Nikhil and Arvind. Can dataflow subsume von Neumann computing? In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 262–272, Israel, 1989.

[18] G. M. Papadopoulos and D. E. Culler. Monsoon: An explicit token-store architecture. In *Proceedings of the Seventeenth Annual International Symposium of Computer Architecture, Seattle, Washington*, pages 82–91, 1990.

[19] Burton Smith. The architecture of HEP. In J. S. Kowalik, editor, *Parallem MIMD Computation: HEP Supercomputer and its Application*, pages 41–55. The MIT Press, 1985.

[20] K. R. Traub. Sequential implementation of lenient programming languages. Technical Report MIT/LCS/TR-417, Laboratory for Computer Science, MIT, 1988.

Data Flow Implementation of Generalized Guarded Commands

R. Govindarajan
VLSI Design Laboratory
McGill University
Montreal, H3A 2A7, Canada
govindr@pike.ee.mcgill.ca

Sheng Yu
Department of Computer Science
University of Western Ontario
London, N6A 5B7, Canada
syu@uwocsd.uwo.ca

Abstract

Earlier approaches to execute generalized alternative/repetitive commands of Communicating Sequential Processes (CSP) attempt the selection of guards in a sequential order. Also, these implementations are based on either shared memory or message passing multiprocessor systems, which exploit parallelism only among the processes of a CSP program. In contrast, we propose a data flow implementation for CSP with generalized guarded commands in which both inter-process and intra-process concurrencies are exploited. A significant feature of our implementation is that it attempts the selection of guards of a process in parallel. A simulated model empirically demonstrates correctness properties, namely 'safety' and 'liveness', of our implementation. The simulation experiments are also helpful in obtaining certain efficiency and fairness parameters of the implementation.

1 Introduction

Hoare's Communicating Sequential Processes (CSP) [14] has been widely accepted as a paradigm for programming parallel computation. But the initial definition of CSP does not allow output commands to appear in the alternative/repetitive commands. Not only is this a hindrance to the symmetry of the language, but this considerably constraints the expressibility [6, 7, 14, 15, 18]. This can be easily understood from the bounded buffer program or the dining philosopher program, where the inability to use output guards in the guarded commands necessitate additional signals. Following these arguments, the guarded commands of CSP have been generalized to allow even the output command to appear in them. While such a generalization is easy to perceive, an implementation is quite involved and requires reaching an agreement among the communicating processes. We explain this with the following example.

```
Process  P1              Process P2              Process P3
...   ...   ...         ...   ...   ...          ...  ...  ...
*[ true;P2!x1 --> S1    *[ true;P1?y1 --> S3    *[ true;P1!z1 --> S5
[] true;P3?x2 --> S2    [] true;P3!y2 --> S4    [] true;P2?z2 --> S6
]                        ]                       ]
...   ...   ...         ...   ...   ...          ...  ...  ...
```

The guarded statements in P1, P2 and P3 can be executed only if (i) both P1 and P2 agree to select the first guard (in their respective repetitive commands), or (ii) both P2 and P3 agree to select the second guard, or (iii) the processes P3 and P1 agree to select, respectively, the first and second guards. Let P1 and P2 make the agreement (or rendezvous). Then it is clear that P3 should not initiate a rendezvous with either P1 or P2. Thus the agreement is not only between the processes that rendezvous, P1 and P2 in the above example, but also with other processes, process P3 in this example, with which P1 and P2 can potentially communicate in the alternative/repetitive command. Thus a global agreement among mutually communicating processes is required to select a guard in the generalized guarded command.

Proposing an implementation for the generalized guarded command has been of constant interest to the research community [2, 3, 4, 7, 10, 18, 20]. Fujimoto's solution [10] was based on a shared memory model while the others employ a loosely-coupled architecture. A commonality observed in all these solutions is they are based on the conventional von Neumann framework, adopting either shared memory or message passing architecture. Also, the selection of guards in a particular process has so far been done sequentially. In contrast to the earlier proposals, we use data-driven evaluation [21] as the basic computation model for executing CSP programs and propose an implementation which parallelizes the selection of guards in a process. Attempting guards in parallel can significantly reduce the execution time of an alternative/repetitive command. It is important to note that our implementation does not sacrifice the semantics of the guarded commands. Other advantages of our implementation include exploiting fine-grain concurrency, allowing parallelism not only among the various processes of a program but also within a single process. Also, in our implementation, Processing Elements (PEs) do not busy wait for synchronization of guarded commands.

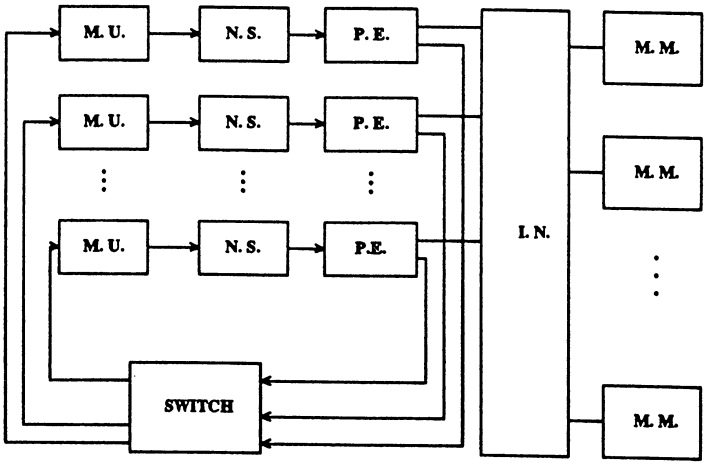
This paper is organized as follows. In the following section, we present the basic architecture and discuss the issues related to implementing generalized guarded commands. Section 3 describes the implementation scheme. A simulated model is developed to empirically demonstrate the correctness of our implementation. Further, certain performance parameters concerning efficiency and fairness of our implementation are also obtained from the simulation experiments. These results are reported in Section 4. Finally, we conclude by comparing our implementation with related works.

2 The Architecture and Related Issues

2.1 The Architecture

Data-driven evaluation has been chosen as the model of computation for our implementation for the following two reasons. Firstly, data flow model does not impose any order (other than what is dictated by data dependency) on the execution. The guards in a process can thus be concurrently attempted in a natural way. Further, fine grain asynchronous concurrency exploited by data flow machines ensures both inter- and intra-process parallelisms in a CSP program. The reader is referred to [1, 13, 21] for an introduction to data-driven evaluation.

Though the our implementation is suitable for any data flow computer, Manchester multi-ring data flow machine [5] is used as the base architecture in this paper. On top of the data flow architecture, we assume a shared memory module accessible to the Processing Elements (PEs) of the machine through an interconnection network. The architecture with the shared memory is shown in Fig.1. In order to increase the memory bandwidth — for obvious performance improvement reasons — the shared memory is low-order interleaved. The PEs are connected to the memory modules by means of a multi-stage interconnection network. Lastly, we assume the availability of a synchronization primitive such as the *fetch and increment* [11].



M. U. -- Matching Unit
N. S. -- Node Store
P. E. -- Processing Element
I. N. -- Interconnection Network
M. M. -- Memory Module

Fig. 1 A Multi-Ring Data Flow Machine

2.2 Implementing CSP

A CSP program consists of a set of communicating processes. We propose to execute a CSP program by converting each process into a data flow graph; the resulting graphs are executed on a data flow machine. The processes of a CSP program can be executed concurrently. A distributed execution of the data flow graph ensures both inter- and intra-process concurrencies. The implementation of the simple commands of CSP, namely skip, assignment, and parallel commands, is straightforward. Also, work on implementing CSP with constrained guarded commands (i.e., allowing only input commands in the guards) has already been reported [17]. Hence we restrict our attention to the implementation of the generalized guarded commands.

2.3 Notations

In a CSP program, the processes are given distinct process identifiers, called *process-ids*. Each invocation of an alternative or repetitive command is referred as a *transaction*. A unique identifier, called *trans-id*, is generated by concatenating the process identifier (of the process which invokes the transaction) and a sequence number. The *trans-ids* are totally ordered. In a transaction, there are a number of guarded commands, each guard having a distinct index. Consider a guarded statement $b; c \rightarrow S$ appearing in a process P_i, where b and c are the boolean and IO guards respectively. If c addresses the process P_j, then P_j is the communicating complement (or simply complement) of P_i and *vice-versa*.

As there can be many guarded commands in a transaction, there are many potential communicating pairs. So, implementing a generalized alternative or repetitive command is equivalent to reaching an agreement among the potential communicating processes. If processes P_i and P_j agree to communicate, then we say P_i and P_j rendezvous. When P_i and P_j rendezvous, it is implicit that the IO guards of them are complementary (that is, one is an input guard and the other an output guard).

2.4 Related Issues

In the first place, data flow implementation forbids the possibility of a process owning a Processing Element (PE). Also, since we allow parallel execution within a single process, various instructions (constructs) belonging to a process can reside on a number of PEs. As a consequence, a single (process) state cannot be assigned to a process. This is in contrast to the earlier implementations [2, 7, 10, 18] which rely heavily on the existence of a unique state for each process.

The execution of input/output commands in CSP warrants the synchronization of the communicating processes. If a PE executing a communication construct is allowed to 'busy wait', then this may lead to a deadlock situation: a situation where each PE executing a communication construct is waiting for synchronization, but the corresponding communicating complements are denied of PEs due to non-availability. Thus, to avoid

deadlocks busy waiting in PEs should be prohibited. That is, a PE must be set free while the communication construct executed by it (PE) waits for synchronization. The details of the construct have to be stored in the shared memory to enable the execution of the construct later when its complement becomes ready.

3 The Implementation

Let us first consider the implementation of an alternative command; the same arguments can be extended to repetitive commands. Also, only those guards whose boolean components succeed are considered. In the following discussion, P_l refers the local process executing the guard g_l in transaction t_l. The process P_r addressed by the guard g_l is called the remote process.

Before we go into the details of the implementation, the basic principle is briefly described below. An alternative command of a process is converted into a data flow graph as shown in Fig.2. The execution proceeds by attempting the guards. The guard g_l of a process P_l is attempted by executing a *Tryguard* actor (to be explained subsequently). The actions performed by this actor are as follows:

1. First, the arrival of guard g_l is marked by writing an entry in a data structure stored in shared memory.
2. Then a matching guard for g_l is searched in the shared data structure.
3. A failure to find a matching guard results in termination of execution of the *Tryguard* actor. However, if a match is found, then the PE executing the *Tryguard* actor attempts to capture the respective processes (in the strict order of their *trans-ids*). This is done using the synchronization primitive *fetch and add*.
4. If the PE cannot capture either of the processes, then the execution of the *Tryguard* actor terminates. Otherwise (*i.e.*, when the PE captures both the processes), the rendezvous occurs.

It may be observed that execution of a *Tryguard* actor may or may not result in a rendezvous. However, when all the guards of a process fail to effect a rendezvous, the process gets into a state similar to the 'waiting' state of [10]. It is guaranteed that, eventually, some other process will find the 'waiting' process willing to rendezvous with it. At that time, the 'waiting' process is woken up by means of tokens.

3.1 Alternative Commands

We store the following variables in shared memory for each transaction t_l.

Committed (t_l): A boolean variable indicating whether transaction t_l is committed to some other transaction.

Excl (t_l): If a transaction t_l wants to rendezvous with t_r, then the PE attempting the rendezvous must acquire exclusive access to the *Committed* variables of both t_l and t_r. To accomplish this, the *Excl* flags are used. The rendezvous takes place only if the PE successfully acquires exclusive access to the *Committed* variables of t_l and t_r, and the *Committed* flags remaining *False* when the access was acquired. The capturing of local and remote processes goes by strict *trans-id* order to avoid cyclic dependencies and deadlocks.

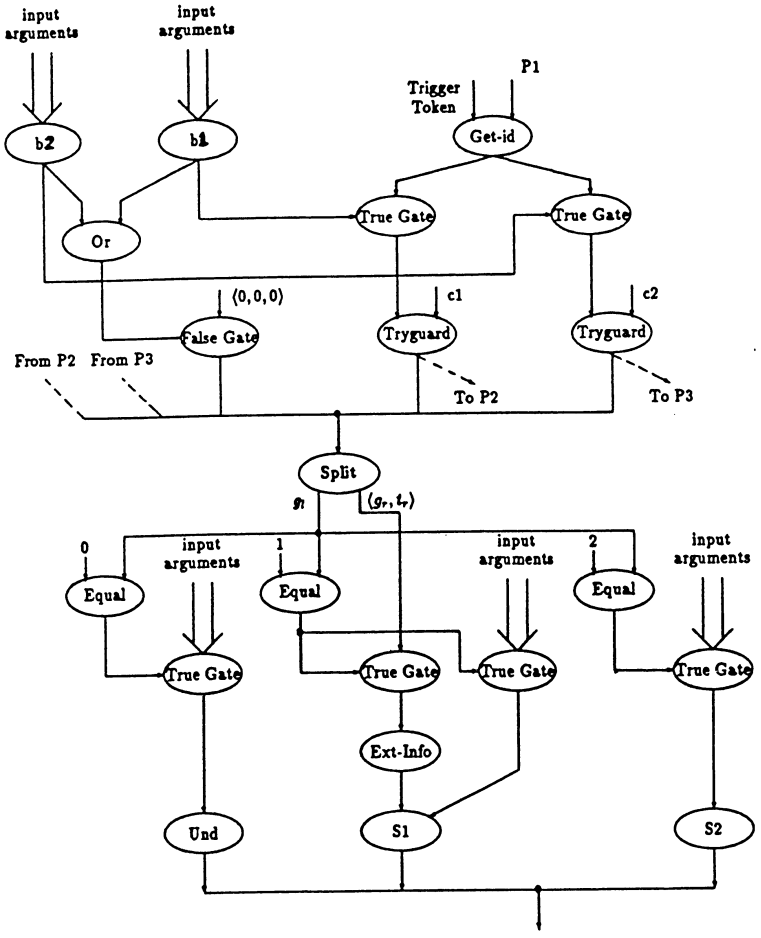


Fig.2 Data Flow Graph for $[b1; P2?x \rightarrow S1[]b2; P3!y \rightarrow S2]$

G-list (t_l): *G-list* (t_l) is an array of linked lists indexed on the processes. For each process P_r, irrespective of whether P_l wishes to communicate to P_r or not, there is an entry *G-list* (t_l, P_r)[1]. Initially all entries of *G-list* (t_l) will contain nil pointers. When a guarded command g_l is attempted for execution, a new entry will be linked to *G-list* (t_l, P_r), where P_r is the remote process addressed by g_l. The entry is a tuple $\langle g_l, c \rangle$, where g_l is the guard index and c provides the communication details of g_l. Typically, c stores the remote process name, signal name, and a bit to identify whether the command is input or output. An entry $\langle g_l, c \rangle$ in *G-list* (t_l, P_r) indicates that the IO guard g_l in process P_l is willing to rendezvous with P_r, provided *Committed* (t_l) is *False*.

The following variables are associated with each process.

Seq-no (P_l): The integer variable *Seq-no* (P_l), initialized to zero, is used for generating a unique *trans-id* for each active transaction in process P_l.

Active-trans (P_l): This variable stores the *trans-id* of the active transaction in P_l. If no transaction is currently active, the variable stores a null value.

An alternative command can be executed by converting it into a data flow graph as shown in Fig. 2. Certain abstractions have been followed in the data flow graph for simplicity. For example, a set of input arguments is passed to a guarded statement through a single *True* gate. In practice a number of *True* gates have to be used for this purpose. Also, we have assumed unlimited fanout for each data flow actor. A few new data flow actors have been introduced in Fig.2. These data flow actors with their respective input and output arguments are shown in Fig. 3. To understand their operational semantics, we need to know about the following two procedures.

Store (P_r, t_l, g_l,c): This procedure links the entry $\langle g_l$, c$\rangle$ to *G-list* (t_l, P_r).

Checkguard (g_l, P_r, c): This procedure is responsible for searching a 'matching' and 'compatible' entry for g_l in *G-list* (t_r), where t_r is active transaction in P_r. The 'matching' condition ensures the remote guard is a potential communicating complement. Two entries are 'compatible' if the IO commands specified in them are complementary in nature. When this procedure succeeds (in finding the matching and compatible guard), it outputs g_r, the guard index in the remote transaction. A failure is indicated by outputting 0.

The operational semantics of the new data flow actors used in our implementation is presented below.

Get-id: The invocation of an alternative command commences with this actor. When the *Get-id* actor is invoked from a process P_l, the PE executing this actor fetches the *Seq-no* (P_l) and increments its contents by one using the synchronization primitive *fetch and increment*. A unique *trans-id* is generated by concatenating P_l with the fetched *Seq-no*.

[1]The arguments for indexing *G-list* (t_l) on *process-ids*, rather than on guard indices are: (i) more than one guard can address the same process; (ii) when a complement process checks *G-list*(t_l), all guards of t_l which are ready and willing to communicate with the complement process need to be tried; and (iii) a linked list representation is an efficient way to access all available guards (*i.e.*, guards ready and willing to rendezvous with the complement process) compared to a scheme where the *G-list* is indexed by the guard indices.

Tryguard: An IO guard is attempted by executing a *Tryguard* actor. The details of the communication guard are specified by the input c as shown in Fig. 3. Note that the values P_l and c_l are constants derived and associated with the node at compile time. This actor invokes the *Checkguard* procedure to determine whether the corresponding guard is ready to rendezvous. If a matching guard is waiting, then an attempt to capture the *Committed* flags of the respective transactions is made. Successful capturing marks a rendezvous. Failure results in releasing captured transaction, if any. The operational semantics of this actor is presented in Fig. 4.

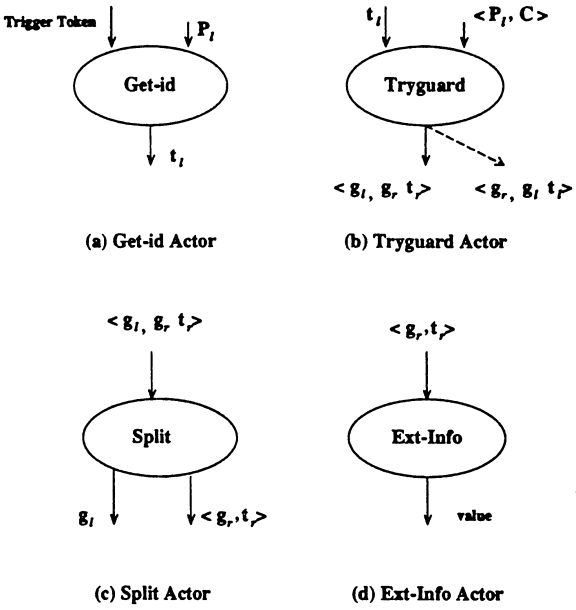


Fig.3 New Data Flow Actors

Split: The *Split* actor receives a triple $\langle g_l, g_r, t_r \rangle$ as its input from a *Tryguard* actor.[2] This actor splits the triple and outputs the g_l value on one output arc and the pair $\langle g_r, t_r \rangle$ on the other output arc. The value of g_l is used to enable the appropriate guarded statement. The $\langle g_r, t_r \rangle$ pair provides the necessary information for the extraction of input data in the IO guards whenever the guard is an Input command.

Ext-Info: When the IO guard is an input command, the corresponding *Tryguard* actor only achieves the synchronization. The actual communication (reception of data) has to be performed with the help of an *Ext-Info* (meaning, extract information) actor. As the necessary synchronization has already been achieved, the data can be input without further synchronization delay.

[2]Or from a *False*-gate when all boolean guards of a transaction are *False*.

The semantics of other data flow actors is same as that presented in [8].

```
PROCEDURE tryguard (t_l, g_l, c);
(* for executing the Tryguard actor the following information are assummed available at the PE
P_l, t_l, g_l : local process id, trans-id, and guard index;
c: communication details of the IO guard; *)
(* P_r, t_r, g_r refer to remote process-id, trans-id, and guard index *)
begin
    Store (P_r, t_l, g_l, c); (* store an entry in G-list (t_l) *)
    t_r := Active-trans (P_r);
    if t_r = 0 then
        skip;
    else
    begin
        g_r := Checkguard (g_l, c);
        if g_r ≠ 0 then
        begin
            t_min := min (t_l, t_r);
            t_max := max (t_l, t_r);
            flag := 0;
            while ((flag=0) AND (NOT Committed (t_min)) AND (NOT Committed (t_max))) do
                flag := fetch-and-increment (Excl (t_min))
                (* capture the transaction with a lower trans-id *)
            if (flag=1) AND (NOT Committed (t_max)) then
            begin release (t_min) := TRUE;
                flag := 0;
                while ((flag=0) AND (NOT Committed (t_max))) do
                    flag := fetch-and-increment (Excl (t_max));
                    (* capture the other transaction *)
                if (flag=1) then
                begin release (t_max) := TRUE;
                    Committed (t_min) := TRUE;
                    Committed (t_max) := TRUE;
                    output-token (⟨g_r,t_r⟩, Split-Node (t_l));
                    (* output a token to the Split actor of t_l *)
                    output-token (⟨g_r,t_l⟩, Split-Node (t_r));
                    (* output a token to the Split actor of t_r *)
                end
            end
        end
        if release (t_min) then Excl (t_min):= TRUE; (* release captured transactions, if any *)
        if release (t_max) then Excl (t_max):= TRUE;
    end
end ;
```

Fig. 4. Operational Semantics of *Tryguard* Actor

From Fig.2 and Fig.4, our implementation scheme can be understood as follows. The execution of an alternative command is started by sending a trigger token (see Fig.2) to the *Get-id* node in the data flow graph. The guards of a transaction t_l are attempted concurrently by executing the *Tryguard* actor. This results in writing an entry in *G-list*(t_l, P_r), where P_r is the remote process addressed by g_l. If there is an active transaction t_r in process P_r, then a search in *G-list* (t_r, P_l) is carried out for a matching compatible entry. The execution of the *Tryguard* actor terminates if this search is unsuccessful. Otherwise the PE executing the *Tryguard* actor tries to acquire exclusive access to the *Committed* flags of t_l and t_r in strict order of their *trans-ids*. If one of the *Committed* flags is *True* the execution of the *Tryguard* actor terminates, releasing captured transaction, if any. A rendezvous takes place when both *Committed* (t_l) and *Committed* (t_r) are *False* and have been successfully captured. The success in capturing the transactions results in outputting the triples $\langle g_l, g_r, t_r \rangle$ and $\langle g_r, g_l, t_l \rangle$ on the input arc of the *Split* actors in t_l and t_r respectively. When all the *Tryguard* actors of t_l fail to find a complement transaction willing to rendezvous, the transaction gets into a state similar to the 'waiting' state of [10]. Eventually, some other transaction will find t_l willing to rendezvous, and trigger t_l through a token to its *Split* actor. Following the data flow graph of Fig.2 it can be easily understood how the appropriate guarded statement gets executed. When all the boolean guards fail, the alternative commands results in a runtime error. The data flow actor *UND* is used for this.

It may be observed that more than one guard of a transaction can be executing the respective *Tryguard* actors in parallel; some of them may even succeed in finding their respective partners. However, only one guard can succeed in committing the rendezvous.

3.2 Repetitive and IO Commands

The above scheme can be extended to implement repetitive and simple IO commands. The data flow graph for a repetitive command is similar to the one shown in Fig.2, except for the recirculation of input arguments and trigger token at the end of execution of each iteration. The execution of a repetitive command terminates when all the guards in that command fails. A repetitive command is executed loop sequentially. It may be observed that the semantics of the repetitive command imposes such a loop sequential execution.

Finally, an input command $P_r?x$ generated by a process P_l is implemented by considering it as an alternative command

$$True; P_r?x \rightarrow skip.$$

Similarly, an output command $P_r!y$ is translated as:

$$True; P_r!y \rightarrow skip.$$

The reason for considering a simple IO command as an alternative command in the implementation is given below. Consider the situation in which a guard of some transaction has its complement as a simple IO command. This guard will never succeed, as the

proposed implementation searches the matching compatible guards only in other alternative/repetitive commands. To take care of this situation, the above search (for matching and compatible guard) should be extended to simple IO commands as well. Instead, IO commands are considered as alternative commands to make the implementation simple and uniform.

4 Simulation

It is necessary to ensure that the proposed scheme correctly implements the semantics of guarded commands. This can be established by proving the following: during the potentially infinite execution, all processes (of the application program) and their interplay maintain the invariant properties, namely 'safety' and 'liveness' [10, 16, 19]. The first property safety means any rendezvous that occurs is correct. Liveness ensures two processes which should rendezvous, eventually will, provided either of them does not rendezvous with any other.

Besides safety and liveness, in general, it is desirable to support 'fairness' in implementations involving non-deterministic choices. In particular two kinds of fairness, *weak* and *strong* fairness, have been defined in literature [9, 10]. An implementation of guarded commands is *weakly* fair, if it can be guaranteed that during an infinitely repetitive execution, a guard that remains *continuously* available (*i.e.,* enabled and complement process ready to communicate) will eventually rendezvous. If it can be guaranteed that a guard which is available infinitely often (though not necessarily continuously) will eventually rendezvous, then the implementation is *strongly* fair.

In [12], it has been formally proved that our implementation maintains safety and liveness properties. However, proving fairness (either *strong* or *weak*) is a difficult task, basically due to the absence of an order of execution of the guards. The guards of a single process can be executed in any order, possibly concurrently. Hence no assumption can be made on their execution order, leaving no basis for proving fairness.

In this paper, we establish safety, liveness, and fairness properties empirically using a simulation approach. Another motivation for conducting simulation experiments is to obtain certain performance parameters, concerning efficiency and fairness of our implementation. In this section we highlight the details of the simulation and the results.

4.1 Simulation Details

As mentioned earlier we have chosen the multi-ring data flow machine [5]. The memory modules are connected to the PEs through a multi-stage interconnection network. For simplicity, the number of rings (or PEs) in the data flow machine is made the same as the number of interleaved memory modules. Further buffering of memory requests/responses between the stages of the network has been assumed to avoid contention in a switch. The multi-ring data flow machine is simulated using a discrete event simulation approach.

Constant service times have been assigned to the functional units. This is based on our knowledge of the time taken to process a token by these functional units [5]. Shared memory access cost involves buffering delay, if there is any contention in the interconnection switches, in addition to the logarithmic delay in the multistage network.

Input Parameters

The application program that we run on this simulator is a repetitive command involving m processes and n guards per process. The repetitive commands in the m processes are such that the I/O guards in one have matching and compatible guards in the complement processes. Also, all guards in all processes are enabled; that is, their boolean components always evaluate to *True*. The number of rings in the multi-ring data flow machine (or the number of interleaved memory module) can be varied and is an input parameter for the simulation.

Output Parameters

First, let us concentrate on proving safety, liveness, and fairness properties. We supplement this result with performance parameters which are measures for the efficiency and fairness of the implementation.

In the simulation run we record a log-file for the transactions that successfully rendezvous. Using the log-file, we ensure that each rendezvous is safe. To establish liveness, let only one pair (chosen randomly) of matching guards be available and check whether the rendezvous takes place within a finite time. The implementation is said to be live, if the rendezvous takes place no matter which pair of complement guards is enabled. Further, this should remain true irrespective of the variations in the number of rings. Lastly, to prove *(weak)* fairness of our implementation, we enable all the guards continuously and expect each of them to participate in a rendezvous, at least once, during a finite simulation time.

Apart from the three properties, we define the following six output parameters which are measures for the efficiency and fairness of the implementations.

(i) Average Tries: Average tries is an important performance parameter from efficiency viewpoint. It can be evaluated as the ratio of the sum of the number of guards tried (before a rendezvous) in each transaction to the total number of transactions. The value of average tries for our implementation could be between 2 (corresponding to one guard in each of the complement processes) and 2*n (where n is the number of guards per transaction). The value of average tries can be normalized by dividing it by 2*n.

(ii) Best Case Figure: As mentioned above, in the best case, a transaction can succeed by attempting just two guards, one in each of the complement processes. Best case figure is the ratio of the number of rendezvouses that took place with just 2 tries to the total number of transactions in the simulation time.

(iii) Worst Case Figure: A rendezvous that occurs after all guards in the complement transactions have been tried corresponds to a worst case occurrence. Counting the number of such rendezvouses and normalizing it by the total number of transactions gives the worst case figure.

(iv) Relative Merit Figure: In other implementations [4, 10, 18] an agreement is reached only after all the n guards of one of the transactions have been attempted. So it is meaningful to count the transactions where an agreement is reached by trying fewer than n guards in the complement processes. Averaging this value over the total number of transactions in the simulation period gives the relative merit figure.

(v) Guard Bias Figure: In a fair implementation each guard in a process is equally like to participate in a rendezvous involving that process. The number of times a guard has participated in a rendezvous, called the success count, is measured for each guard. Ideally, the success count for a guard should be equal to

$$\frac{2 * total\,no\,of\,transactions}{number\,of\,guards}$$

The standard deviation of the success counts of guards in a process is a measure of biasing. This figure, referred as the guard bias figure, indicates the extent to which a guard is biased/favored. This value should be very low for a fair implementation.

(vi) Process Bias Figure: The process bias figure is a measure to what extent a process has been biased/favored in all rendezvouses that took place during the simulation period. This figure is similar to the guard bias figure and can be obtained by computing the standard deviation of the success counts of various processes.

4.2 Results

The input program for the simulator was run several times with the following values of m m, the number of processes, and n, the number of guards per process: (i) $m = 5$ and $n = 8$, (ii) $m = 5$ and $n = 16$, and (iii) $m = 9$ and $n = 16$. All three properties, namely safety, liveness, and (weak) fairness have been checked for all of the above cases. The experiments are repeated on different configurations of the data flow machine having 1, 2, 4, 8, 16, or 32 rings. To measure the performance parameters, we ran the application programs long enough for at least 100 rendezvouses to take place.

Average Tries: Table 1 lists the normalized value of average tries (for a rendezvous) for various simulation runs. We observe that this value varies from 0.6 to 0.95 for the different runs. In particular, when the number of guards per process is smaller and the number of rings in the data flow machine is larger, the value of average tries is high. This can be reasoned as follows. When there is low parallelism (*i.e.*, less number of guards to try) and high resource availability, the chances for more number of guards to be tried are high. Further, we have assumed (i) the alternative commands in all the processes are initiated at the same time, and (ii) it takes equal execution time for the all boolean guards. These two assumptions further add to the possibility that the *Tryguard* actors of a process can be executed simultaneously, accounting for the large value of normalized average tries.

Table 1. Normalized Average Tries

Input Programs with		Normalized Average Tries					
		Number of Rings					
Processes	Guards	1	2	4	8	16	32
5	8	0.694	0.690	0.727	0.795	0.822	0.833
5	16	0.600	0.606	0.633	0.662	0.709	0.701
9	16	0.666	0.669	0.639	0.689	0.755	0.750

Best Case, Worst Case, and Relative Merit Figures: We do not observe the occurrence of the best case. As the guards of a process are attempted concurrently and as the *Tryguard* actors of a process are enabled more or less at the same time, it is more likely that at least two guards in a process are tried simultaneously, forbidding the best case occurrence. The worst case and relative merit figures for the various runs are tabulated in Table 2. The worst case figure drops steadily as the parallelism increases. When the parallelism is high, the worst case virtually never occurs. Also, the relative merit figure shows an increasing trend with an increase in the parallelism.

Table 2. Worst Case and Relative Merit Figures

Input Programs with		Parameters	Performance Parameters					
			Number of Rings					
Processes	Guards		1	2	4	8	16	32
			(in %)					
5	8	Worst Case	0.0	0.0	0.0	7.4	10.6	9.7
		Relative Merit	0.0	0.0	6.8	1.6	3.8	0.0
5	16	Worst Case	0.0	0.0	0.0	0.0	0.0	0.0
		Relative Merit	0.0	0.0	6.7	4.9	0.0	0.0
9	16	Worst Case	0.0	0.0	0.0	0.0	4.4	0.9
		Relative Merit	0.0	12.5	21.9	10.0	0.0	0.0

Fairness Measures: The guard bias figures (in a typical process) and the process bias figures for various simulation runs are shown in Table 3. We notice these values are very low, at most 4 transactions out of nearly 100 transactions. Such a low degree of biasing is acceptable to an implementation involving non-determinism.

Table 3. Fairness Measures

Input Programs with			Bias Figures					
			Number of Rings					
Processes	Guards		1	2	4	8	16	32
			(in number of transactions)					
5	8	Guard	1.14	0.97	1.68	2.19	1.93	2.63
		Process	1.67	1.20	1.36	0.97	2.56	2.65
5	16	Guard	0.76	0.78	1.34	1.37	1.74	1.71
		Process	1.35	0.74	2.75	1.16	1.54	3.84
9	16	Guard	0.41	0.64	0.80	1.46	1.21	1.60
		Process	0.49	0.56	1.03	1.07	0.66	1.52

Variable Execution Time for Boolean Guards and Guarded Statements:

As mentioned earlier, in the earlier simulation runs an equal execution time is assumed for all boolean guards and guarded statements (statements following the guards). To study the effect of variable execution times (of boolean guards and guarded statements) on the performance parameters, we use a uniformly distributed execution time. The range of the uniform distribution is from 0 time units to 100 time units. Due to this difference, the *Tryguard* actors may be activated at different time. This is expected to reduce the average tries. The performance results indeed show a decrease in the average tries and a significant increase in the relative merit figure (refer Table 4). The results are for the repetitive command involving 5 processes and 16 guards.

Table 4. Results for Variable Execution Time

Number of Rings	Variable Execution Time for			
	Boolean Guards		Guarded Statements	
	Normalized Average Tries	Relative Merit	Normalized Average Tries	Relative Merit
1	0.60	0.0%	0.58	0.0%
2	0.57	11.1%	0.57	29.1%
4	0.56	33.3%	0.52	59.1%
8	0.54	35.8%	0.51	65.7%
16	0.63	12.6%	0.59	75.8%
32	0.60	23.7%	0.63	67.3%

5 Related Works and Comparisons

Our implementation, for the first time, uses data flow model of computation. As mentioned earlier, data-driven computation facilitates fine grain asynchronous parallelism. In our implementation parallelism exploited is not only among processes but also within a process. A marked difference from other models is the concurrent execution of guards in a process. Further, processors do not busy wait during synchronization.

5.1 Shared Memory Systems

The work reported in [10] is based on shared memory model and resembles our implementation in some aspects. However, our implementation performs better than the former in the following two ways.

In [10], a rendezvous between two transactions takes place only after one of them enters the 'waiting' state. This means that all guards of one transaction have to be attempted before the rendezvous. That is, if there are n guards in each process, then at least $(n+1)$ must be tried for a rendezvous. $2 * n$ tries wil be made in the worst case. The average case results are not reported in [10]. Theoretically, the best and worst case figures for

our implementation are 2 and $2 * n$ respectively. Also, the simulation results predict an average of $(0.6) * (2 * n)$ guards are tried for a rendezvous. The relative merit figure shows that our implementation indeed performs better than [10] in many cases.

It is important to realize that the above comparisons are based only on the number of tries and not on the execution time. Given that the guards of a process are attempted concurrently in our implementation, it easily outperforms the other [10] in terms of execution time. The improvement in performance is not without its price. In our implementation, we store the guard index and communication detail for each attempted guard in the *G-list* (t_l); whereas, only the guard details are stored in the *Alt-List* in [10]. However, with the developments in VLSI technology, it can be argued that an improvement in execution speed at the expense of storage is affordable.

5.2 Message Passing Systems

The implementations presented in [2, 3, 4, 7, 18] use message passing architecture. Some of them [2, 7] are based on a two phase algorithm. Our implementation, like the ones presented in [4, 10, 18], involves only one phase. There is no need for a second phase as every *Tryguard* actor terminates with a definite answer. That is, non-committal replies and retries do not occur in our implementation.

It is unfair to evaluate our implementation directly in the light of the six criteria listed in [2, 18] as these are essentially for a loosely-coupled multiprocessor system. However our implementation retains the spirit of these criteria. For instance, five of these six criteria[3] can be appropriately redefined as: (i) the amount of system information stored (in the shared memory) should be minimal; (ii) when both t_l and t_r are ready to select the guards g_l and g_r, respectively, then at least one of the transactions must select a guard (not necessarily g_l or g_r) within a finite time; (iii) the number of attempts made in selecting a guard of a transaction should be bounded; (iv) the time taken by a *Tryguard* actor to determine whether it can establish a rendezvous must be finite; and (v) if a process has a guarded command that is enabled continuously for an infinite time, then it should eventually succeed. It can be easily proved that our implementation satisfies these criteria. In fact it performs better than others [2, 3, 4, 7, 10, 18] with respect to (ii) and (iii), as parallelizing the selection of guards significantly reduces the execution time.

6 Conclusions

In this paper, we have presented a decentralized parallel implementation for the generalized guarded commands of CSP using data flow model of computation. A simulation study has been conducted to measure various performance and fairness parameters. Also, simulation experiments establish safety, liveness, and fairness of our implementation. Finally, a comparison with the existing ones reveals the superiority of our model. We have

[3]Criterion (i) of [18] is not relevant to the discussion.

not addressed the issue of process termination. However, the implementation can be easily extended to take care of process termination.

Acknowledgments

The authors acknowledge Dr.R.A. Nicholl and Dr.V.S.Lakshmanan for their useful comments. The authors are thankful to Bhama for her comments on the initial draft of this paper.

References

[1] Arvind and Gostelow, K.P., "The U Interpreter", *IEEE Computer*, vol.15, no.2, pp.42-49, Feb. 1982.

[2] Back, R.J.R, Ekslund, P., and Kurki-Suonia, R., "A Fair and Efficient Implementation of CSP with Output Guards", Technical Report, Ser. A, No. 38, Abo Akademic, Finland, 1984.

[3] Bagrodia, R., "A Distributed Algorithm to Implement the Generalized Alternative Command in CSP", in *Proc. of the 6th International Conference on Distributed Computing Systems*, pp. 422-427, 1986.

[4] Bagrodia, R., "Synchronization of Asynchronous Processes in CSP", *ACM Transactions on Programming Languages and Systems*, vol.11, no.4, pp.585-597, Oct.1989.

[5] Barahona, P.M.C.C., and Gurd, J.R., "Processor Allocation in a Multi-ring Data Flow Machine," *Journal of Parallel and Distributed Computing*, vol.3, no.3, pp.305-327, 1986.

[6] Bernstein, A.J., "Output Guards and Nondeterminism in Communicating Sequential Processes", *ACM Transactions on Programming Language and Systems*, vol.2, no.2, pp.234-238, April 1980.

[7] Buckley, G.N. and Silberschatz, A., "An Effective Implementation for the Generalized Input-Output Construct of CSP", *ACM Transactions on Programming Languages and Systems*", vol. 5, no. 2, pp. 223-235, 1983.

[8] Davis, A.L. and Keller, R.M., "Data Flow Program Graphs", *IEEE Computer*, vol.15, no.2, pp.26-41, Feb. 1982.

[9] Francez, N., *Fairness*, Springer-Verlag, New York, 1986.

[10] Fujimoto, R.N. and Hwa-chung Feng, "A Shared Memory Algorithm and Proof for the Generalized Alternative Construct in CSP", *International Journal of Parallel Programming*, vol. 16, no. 3, pp. 215-241, 1987.

[11] Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, Rudolph, L., and Snir, M., "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer", *IEEE Transactions on Computers*, vol.C-32, no.2, pp.175-189, Feb. 1983.

[12] Govindarajan, R. and Yu. S, "Attempting Guards in Parallel: A Data Flow Approach to Execute Generalized Guarded Commands", Technical Report # 273, Department of Computer Science, University of Western Ontario, London, May 1990.

[13] Gurd, J.R., Watson, I., and Kirkham, C.C., "The Manchester Prototype Data Flow Computer", *Communications of the ACM*, vol.28, no.1, pp.34-52, Jan.1985.

[14] Hoare, C.A.R., "Communicating Sequential Processes", *Communications of the ACM*, vol. 21, no. 8, pp. 666-677, Aug. 1978.

[15] Kieburtz, R.B. and Silberschatz, A., "Comments on Communicating Sequential Processes", *ACM Transactions on Programming Language and Systems*, vol.1, no.2, pp.218-225, Oct. 1979.

[16] Owicki, S. and Lamport, L., "Proving Liveness Properties of Concurrent Programs", *ACM Transactions on Programming Languages and Systems*, vol. 6, no. 2, pp. 455-495, July 1982.

[17] Patnaik, L.M. and Basu, J., "Two Tools for Interprocess Communication in Distributed Data Flow Systems", *The Computer Journal*, vol. 29, no. 6, pp. 506-521, Dec. 1986.

[18] Ramesh, S., "A New Implementation of CSP with Output Guards" in *Proc. of the 7th International Conference on Distributed Computing Systems*, pp. 266-273, 1987.

[19] Reed, D.A., Malony, A.D., and McCredie, B.D., "Parallel Discrete Event Simulation: A Shared Memory Approach", in *Proc. of the ACM SIGMETRIC S Conference on Measuring and Modeling Computer Systems*, vol. 15, no.1, pp. 36-38, May 1987.

[20] Silberschatz, A., "Communication and Synchronization in Distributed Systems" *IEEE Transactions on Software Engineering*, vol.SE-5, no. 6, pp.542-546, Nov.1979.

[21] Treleaven, P.C., Brownbridge, D.R., and Hopkins, R.P., "Data-Driven and Demand-Driven Architecture", *Computing Surveys*, vol. 14, no. 1, pp. 93-143, Mar. 1982.

ON THE DESIGN OF DEADLOCK-FREE ADAPTIVE ROUTING ALGORITHMS FOR MULTICOMPUTERS: DESIGN METHODOLOGIES

J. Duato
Dept. de Ingeniería de Sistemas, Computadores y Automática
Facultad de Informática. Universidad Politécnica de Valencia
P.O.B. 22012. 46071 - Valencia, Spain

Abstract

Second generation multicomputers use wormhole routing, allowing a very low channel set-up time and drastically reducing the dependency between network latency and internode distance. Deadlock-free routing strategies have been developed, allowing the implementation of fast hardware routers that reduce the communication bottleneck. Also, adaptive routing algorithms with deadlock-avoidance or deadlock-recovery techniques have been proposed for some topologies, being very effective and outperforming static strategies.

This paper develops the theoretical background for the design of deadlock-free adaptive routing algorithms for wormhole as well as store-and-forward routing. Some basic definitions and four theorems are proposed, developing conditions to verify that an adaptive algorithm is deadlock-free, even when there are cycles in the channel dependency graph. Also, two design methodologies are proposed. The first one supplies algorithms with a high degree of freedom, without increasing the number of physical channels. The second methodology is intended for the design of fault-tolerant algorithms. Some examples are given, showing the application of the methodologies.

1. Introduction

Multicomputers [1] rely on an interconnection network between processors to support the message-passing mechanism. The network latency [1] can be defined as the time from when the head of a message enters the network at the source until the tail emerges at the destination. In first generation multicomputers, a store-and-forward mechanism has been used to route messages. Each time a message reaches a node, it is buffered in local memory, and the processor interrupted to execute the routing algorithm. Accordingly, the network latency is proportional to the distance between the origin and the destination.

However, second generation multicomputers are most distinguished by message routing hardware that makes the topology of the message-passing network practically invisible to the programmer. The message routing hardware uses a routing mechanism known as

This work is partly supported by CICYT grant number TIC 87-0655

wormhole routing [9]. As messages are typically at least a few words long, each message is serialized into a sequence of parallel data units, referred to as flow control units, or flits [8]. The flit at the head of a message governs the route. As the header flit advances along the specified route, the remaining flits follow it in a pipeline fashion. If the header encounters a channel already in use, it is blocked until the channel is freed; the flow control within the network blocks the trailing flits.

This form of routing and flow control has two important advantages over the store-and-forward packet routing used in first generation multicomputers. Firstly, it avoids using storage bandwidth in the nodes through which messages are routed. Secondly, this routing technique makes the message latency largely insensitive to the distance in the message-passing network. Since the flits move through the network in a pipeline fashion, in the absence of channel contention, the network latency equals the sum of two terms:

- T_pD is the time associated with forming the path through the network, where T_p is the delay of the individual routing nodes found on the path, and D is the number of nodes traversed.

- L/B is the time required for a message of length L to pass through a channel of bandwidth B.

In second generation multicomputers, the network latency is dominated by the second term for all but very short messages.

Another improvement in message performance results from selecting the optimal topology for the implementation on printed circuit boards or VLSI chips. As message latency is dominated by the term L/B, more wirable network topologies will increase the bandwidth B at the expense of increasing the network diameter. An analysis [5,7] shows that, under the assumption of constant number of wires through the network bisection, a two dimensional network minimizes latency for typical message lengths for up to 1024 nodes. For larger sizes, a three dimensional network achieves better performance. Among these networks, meshes are preferred because they offer useful edge connectivity, which can be used for I/O controllers. Also, meshes partition into units that are still meshes, simplifying the design of routing algorithms that are independent of the network size, as well as the implementation of space-sharing techniques.

However, deadlocks may appear if the routing algorithms are not carefully designed. A deadlock in the interconnection network of a multicomputer occurs when no message can advance toward its destination because the queues of the message system are full. The size of the queues strongly influences the probability to reach a deadlocked configuration. First generation multicomputers buffer full messages or relatively large packets. By contrary, second generation machines buffer flits, being more deadlock-prone. So, the only practical way to avoid deadlock is to design deadlock-free routing algorithms.

Many deadlock-free routing algorithms have been developed for store-and-forward computer networks [12,14,20]. These algorithms are based on a structured buffer pool. However, with wormhole routing, buffer allocation cannot be restricted, because flits have no routing information. Once the header of a message has been accepted by a channel, the remaining flits must be accepted before the flits of any other message can be accepted. So, routing must be restricted to avoid deadlock.

Dally [9] has proposed a methodology to design static routing algorithms under general assumptions. He defines a channel dependency graph and establishes a total order among channels. Routing is restricted to visit channels in decreasing or increasing order to eliminate cycles in the channel dependency graph. This methodology has been applied to the design of routing chips for multicomputers [8] and multicomputer nodes with integrated communication support [2]. It has also been applied to systolic communication [19,2].

The restriction of routing, although avoids deadlock, can increase traffic jams, specially in heavily loaded networks with long messages. In order to avoid congested regions of the network, an adaptive routing algorithm can be used. Adaptive strategies have been shown to outperform static strategies in store-and-forward routing [3] and in packet-switched communications [18,21]. In general, adaptive routing needs additional hardware support.

Several adaptive algorithms have been developed for wormhole routing. A deadlock-free adaptive algorithm for the hypercube is the Hyperswitch algorithm [4], which is based on backtracking and hardware modification of message headers to avoid congestion and cycles. Another deadlock-free adaptive algorithm has been proposed for the MEGA [13]. This algorithm always routes messages, sending them away from their destination if necessary, like the Connection Machine [15]. If a message arrives to a node without free output channels, deadlocks are avoided by storing the message and removing it from the network. In this respect, it is similar to virtual cut-through [17]. Jesshope [16] has proposed an algorithm for n dimensional meshes, by decomposing them into 2n virtual networks. Inside each virtual network, displacements along a given dimension are always made in the same direction, thus avoiding cycles and deadlock.

An alternative way consists of recovering from deadlock. Reeves et al. [22] have used an abort-and-retry technique to remove messages blocked for longer than a certain threshold from the network. Aborted messages are introduced again into the network after a random delay. In [22] three adaptive routing strategies have been proposed and evaluated for a binary 8-cube.

In [10] we have proposed a very simple methodology to design deadlock-free adaptive routing algorithms for wormhole networks. The routing algorithms obtained from the application of that methodology to 2D and 3D-meshes have been evaluated by simulation.

Also, in [11] we have presented a first version of a theory for the design of deadlock-free adaptive routing algorithms. Some basic definitions and three theorems have been proposed, as well as a design example. However, that theory is only valid for single flit messages, contrary to one of the assumptions stated in the paper, being useful for store-and-forward routing.

This paper develops the theoretical background for the design of deadlock-free adaptive routing algorithms. For wormhole routing, some basic definitions and two theorems are proposed and proved, developing conditions to verify that an adaptive algorithm is deadlock-free, even when there are cycles in the channel dependency graph. For store-and-forward routing, the theory presented in [11] is updated. Also, two design methodologies based on the above mentioned theorems are proposed. The first one supplies adaptive algorithms with a high degree of freedom, without increasing the number of physical channels. The second methodology is intended for the design of fault-tolerant algorithms.

Section 2 develops the new theory for wormhole routing. Section 3 summarizes the theoretical aspects for store-and-forward routing. Section 4 proposes two design methodologies, giving some examples of their application. Finally, some conclusions are drawn.

2. Definitions and theorems for wormhole routing

This section develops the theoretical background for the design of deadlock-free adaptive routing algorithms for networks using wormhole routing.

The basic assumptions are very similar to the ones proposed by Dally [9], except that adaptive routing is allowed. These assumptions are the following:

1) A node can generate messages destined for any other node at any rate.

2) A message arriving at its destination node is eventually consumed.

3) Wormhole routing is used. So, once a queue accepts the first flit of a message, it must accept the remainder of the message before accepting any flits from another message.

4) A node can generate messages of arbitrary length. Packets will generally be longer than a single flit.

5) An available queue may arbitrate between messages that request that queue, but may not choose among waiting messages.

6) A queue cannot contain flits belonging to different messages or packets. After accepting a tail flit, a queue must be emptied before accepting another header flit. Then, when a message or packet is blocked, its header flit will always occupy the head of a queue.

7) The route taken by a message depends on its destination and the status of output channels (free or busy). At a given node, the routing function supplies a set of output channels based on the current and destination nodes. A selection from this set is made based on the status of output channels at the current node. So, *adaptive* routing will be considered.

Before to propose the theorems, some definitions are needed:

Definition 1: An *interconnection network* I is a strongly connected directed multigraph, I = G(N, C). The vertices of the multigraph N represent the set of processing nodes. The edges of the multigraph C represent the set of communication channels. More than a single channel is allowed to connect a given pair of nodes. Each channel c_i has an associated queue denoted queue(c_i) with capacity cap(c_i). The source and destination nodes of channel c_i are denoted s_i and d_i, respectively.

Definition 2: Let B be the set of valid *channel status*, B = {free, busy}. Let T: C $\rightarrow$ B be the status of the output channels in the network.

Definition 3: An *adaptive routing function* R: N x N $\rightarrow$ C^p supplies a set of p alternative output channels to send a message from the current node n_c to the destination node n_d, $R(n_c, n_d) = \{c_1, c_2, \ldots c_p\}$. In general, p will be less than the number of output channels per node to restrict routing and obtain deadlock-free algorithms. As a particular case, p = 1 defines a static routing function. Also, the channels in the set supplied by R are not necessarily different. So, p is the maximum number of choices. In particular,

$R(n, n) = \varnothing, \quad \forall\, n \in N.$

Definition 4: A *selection function* $S: C^p \times B^p \rightarrow C$ selects a free output channel (if any) from the set supplied by the routing function. From the definition, S takes into account the status of all the channels belonging to the set supplied by the routing function. The selection can be random or based on static or dynamic priorities. Also, in the same way the result of a static routing function may be a busy channel, if all the output channels are busy, any of them is selected. The decomposition of the adaptive routing into two functions (routing and selection) will be critical while proving the theorems, because only the routing function determines whether a routing algorithm is deadlock-free or not. Then, the selection function will only affect performance. Moreover, it is possible to extend the definition of the selection function by taking into account additional information, either local to the node or remote. We will comment on this in section 4.

Definition 5: A routing function R for a given interconnection network I is *connected* iff

$$\forall\, i, j \in N \quad i \neq j, \quad \exists\, c_1, c_2, \ldots c_k \in C \quad \ni$$

$$c_1 \in R(i, j) \quad \wedge \quad c_{m+1} \in R(d_m, j) \quad \forall\, m \in \{1, k-1\} \quad \wedge \quad d_k = j$$

In other words, it is possible to establish a path between i and j using channels belonging to the sets supplied by R. Notice that the interconnection network is strongly connected, but it does not imply that the routing function must be connected.

Definition 6: A *routing subfunction* R_1 for a given routing function R and channel subset $C_1 \subseteq C$, is a routing function

$$R_1: N \times N \rightarrow C_1^q, \quad 0 < q \leq p \quad \ni$$

$$R_1(i, j) = R(i, j) \cap C_1 \quad \forall\, i, j \in N$$

Definition 7: Given an interconnection network I, a routing function R and a pair of channels $c_i, c_j \in C$, there is a *direct dependency* from c_i to c_j iff

$$c_i \in R(s_i, n) \quad \wedge \quad c_j \in R(d_i, n) \quad \text{for some } n \in N$$

that is, c_j can be used immediately after c_i by messages destined to some node n.

Definition 8: Given an interconnection network I, a routing function R, a channel subset $C_1 \subset C$ which defines a routing subfunction R_1 and a pair of channels $c_i, c_j \in C_1$, there is an *indirect dependency* from c_i to c_j iff

$$\exists\, c_1, c_2, \ldots c_k \in C - C_1 \quad \ni$$

$$c_i \in R_1(s_i, n) \quad \wedge \quad c_1 \in R(d_i, n) \quad \wedge$$

$$c_{m+1} \in R(d_m, n) \quad \forall\, m \in \{1, k-1\} \quad \wedge$$

$$d_k = s_j \quad \wedge \quad c_j \in R_1(s_j, n) \quad \text{for some } n \in N$$

that is, it is possible to establish a path from s_i to d_j for messages destined to some node n. c_i and c_j are the first and last channels in that path and the only ones belonging to C_1. Then, c_j can be used after c_i by some messages. As c_i and c_j are not adjacent, some other channels belonging to $C - C_1$ are used between them. It must be noticed that, given three channels $c_i, c_k \in C_1$ and $c_j \in C - C_1$, the existence of direct dependencies between c_i, c_j and c_j, c_k, respectively, does not imply the existence of an indirect dependency between c_i, c_k.

Definition 9: A *channel dependency graph* D for a given interconnection network I and routing function R, is a directed graph, D = G(C, E). The vertices of D are the channels of I. The edges of D are the pairs of channels (c_i, c_j) such that there is a direct dependency from c_i to c_j. Notice that there are no 1-cycles in D, because channels are unidirectional.

Definition 10: An *extended channel dependency graph* D_E for a given interconnection network I and routing subfunction R_1 of a routing function R, is a directed graph, $D_E = G(C_1, E_E)$. The vertices of D_E are the channels that define the routing subfunction R_1. The edges of D_E are the pairs of channels (c_i, c_j) such that there is either a direct or an indirect dependency from c_i to c_j.

Definition 11: A *sink* channel for a given interconnection network I and routing function R is a channel c_i ∋

$$\forall\, j \in N, \quad c_i \in R(s_i, j) \quad \Rightarrow \quad j = d_i$$

In other words, all the flits that enter a sink channel reach their destination in a single hop. As a result, there are no outgoing arcs from a sink channel in any channel dependency graph, as can be easily seen from the definitions.

Definition 12: A *configuration* is an assignment of a set of flits to each queue, all of them belonging to the same message or packet (assumption 6). The number of flits in the queue for channel c_i will be denoted $size(c_i)$. The destination node for a flit f_j will be denoted $dest(f_j)$. If the first flit in the queue for channel c_i is a header flit destined for node n_d, then $head(c_i) = n_d$. If the first flit is not a header and the next channel reserved by its header is c_j, then $next(c_i) = c_j$, that is, each flit must follow the same path as its header. A configuration is *legal* iff

$$\forall\, c_i \in C, \quad size(c_i) \leq cap(c_i) \quad \wedge$$
$$c_i \in R(s_i, dest(f_j)) \quad \forall\, f_j \in queue(c_i)$$

that is, the queue capacity is not exceeded and all the flits stored in the queue have been sent there by the routing function.

Definition 13: A *deadlocked configuration* for a given interconnection network I and routing function R is a nonempty legal configuration verifying the following conditions:

1) $\forall\, c_i \in C \ni head(c_i) \in N \quad \Rightarrow \quad head(c_i) \neq d_i \wedge size(c_j) > 0 \quad \forall\, c_j \in R(d_i, head(c_i))$

2) $\forall\, c_i \in C \ni next(c_i) \in C \quad \Rightarrow \quad size(next(c_i)) = cap(next(c_i))$

The first condition refers to channels with a header flit at their queue head ($head(c_i) \in N$). The second condition refers to channels containing a data or tail flit at their queue head, not destined to d_i ($next(c_i) \in C$). No condition is imposed to empty channels. In a deadlocked configuration there is not any header flit one hop from its destination. Header flits cannot advance because the queues for all the alternative output channels supplied by the routing function are not empty (see assumption 6). As a particular case (for disconnected routing functions), the routing function may not supply any output channel. Data and tail flits cannot advance because the next channel reserved by their message header has a full queue. It must be noticed that a data flit can be blocked at a node even if there are free output channels to reach its destination. Also, in a deadlocked configuration, there is no message whose header flit has already arrived to its destination.

Definition 14: A routing function R for an interconnection network I is *deadlock-free* iff there is not any deadlocked configuration for that routing function on that network.

Two theorems are proposed. The first one is a straightforward extension of Dally's theorem for adaptive routing functions. The second one allows the design of adaptive routing functions with cyclic dependencies in their channel dependency graph. For each theorem, a sketch of the proof as well as the full proof are given.

Theorem 1: A connected and adaptive routing function R for an interconnection network I is deadlock-free if there are no cycles in the channel dependency graph D.

Proof sketch:

$\Leftarrow$ As the channel dependency graph for R is acyclic, it is possible to establish an order between the channels of C. As R is connected, the minimals of that order are also sinks. Suppose that there is a deadlocked configuration for R. Let c_i be a channel of C with a nonempty queue such that there are no channels less than c_i with a nonempty queue. If c_i is a minimal (that is, a sink) then the flits are not blocked and there is no deadlock. Otherwise, using the channels less than c_i, the flit at the queue head of c_i can advance and there is not a deadlock.•

Proof:

$\Leftarrow$ Suppose that there are no cycles in D. Then, one can assign an order to the channels of C so that if $(c_i, c_j) \in E$ then $c_i > c_j$. Consider the channel(s) c_i $\ni$

$\forall\, c_j \in C, \quad (c_i, c_j) \notin E$

Such a channel c_i is a minimal of the order. Let us prove that it is a sink. If it were not a sink, as the routing function is connected, for any legal configuration with a header flit stored in the queue head of c_i

$d_i \neq \text{head}(c_i) \quad \Rightarrow \quad \exists\, c_k \in C \quad \ni \quad c_k \in R(d_i, \text{head}(c_i))$

As the configuration is legal then

$c_i \in R(s_i, \text{head}(c_i)) \quad \Rightarrow \quad (c_i, c_k) \in E$

contrary to the assumption that c_i is a minimal. So, $d_i = \text{head}(c_i)$ and c_i is a sink of D.

Suppose that there is a deadlocked configuration for R. Let c_i be a channel with a nonempty queue such that there is not any channel less than c_i with a nonempty queue. If c_i is a minimal, it is also a sink and then, all the flits stored in its queue will be destined to d_i and the flit at the head of the queue for c_i is not blocked. If c_i is not a minimal then

$\text{size}(c_j) = 0 \quad \forall\, c_j \in C \quad \ni \quad c_i > c_j$

Thus, the flit at the head of the queue for c_i is not blocked, regardless it is a header or a data flit, and there is no deadlock.•

There are some interesting considerations:

1) The theorem gives a sufficient but not necessary condition for an adaptive routing function to be deadlock-free. As will be seen later, the existence of cycles in the channel dependency graph does not imply the existence of a deadlocked configuration.

2) For most networks and routing functions, even for static ones, only a partial order

between channels can be defined, based on the set E. In general, there will be more than a single sink in D.

3) As indicated above, in a legal configuration all the flits stored in a given queue have been sent there by the routing function. Otherwise, the theorem cannot be proved. Consider, for instance, a configuration in which the queues of all the sink channels in D are full of flits destined to nodes not directly connected to those channels.

Theorem 2: A connected and adaptive routing function R for an interconnection network I is deadlock-free if it exists a subset of channels $C_1 \subseteq C$ that defines a routing subfunction R_1 which is connected and has no cycles in its extended channel dependency graph D_E.

Proof sketch:

$\Leftarrow$ The case $C_1 = C$ is trivial. Otherwise $C_1 \subset C$. As the extended channel dependency graph for R_1 is acyclic, it is possible to establish an order between the channels of C_1. As R_1 is connected, the minimals of that order are also sinks. Suppose that there is a deadlocked configuration for R. There are two possible cases:

a) The queues for channels belonging to C_1 are empty. As R_1 is connected and the header flits are at queue heads, they can be routed using channels belonging to C_1 and there is no deadlock.

b) The queues for channels belonging to C_1 are not empty. Let c_i be a channel of C_1 with a nonempty queue such that there are no channels less than c_i with a nonempty queue. Again, there are two possible cases:

b1) If c_i is a minimal (sink) then the flits are not blocked and there is no deadlock.

b2) If c_i is not a minimal, all the channels of C_1 less than c_i will have empty queues, existing three possible cases:

b2.1) If c_i has a header at the queue head, it can be routed because R_1 is connected and there is no deadlock.

b2.2) If there is a data flit at the queue head of c_i and next(c_i) belongs to C_1, that flit can also advance.

b2.3) If next(c_i) belongs to $C - C_1$, we have to use the indirect dependencies in the extended channel dependency graph. Let c_k be the channel containing the header of the data flits contained in c_i. Then, it is possible to find a channel c_j belonging to C_1 to route that header, because R_1 is connected. In that case, there is an indirect dependency from c_i to c_j ($c_i > c_j$), implying that c_j is empty and there is no deadlock.•

Proof:

$\Leftarrow$ Suppose that R_1 is connected and there are no cycles in D_E. If $C_1 = C$ then $D_E = D$, because $C - C_1 = \emptyset$. Thus, there is not any cycle in D and R is deadlock-free by theorem 1. Otherwise $C_1 \subset C$. As there are no cycles in D_E, one can assign an order to the channels of C_1 so that if $(c_i, c_j) \in E_E$ then $c_i > c_j$. Similarly to theorem 1, it can be proved that the minimals of that order are also sinks.

Suppose that there is a deadlocked configuration for R. There are two possible cases:

a) The queues for channels belonging to C_1 are empty. Then, there will be channels belonging to $C - C_1$ with header flits at their queue heads. Let c_i be one of those channels. As R_1 is connected then

$$head(c_i) \neq d_i \Rightarrow \exists\, c_j \in C_1 \ni c_j \in R_1(d_i, head(c_i)) \Rightarrow$$
$$\exists\, c_j \in C \ni c_j \in R(d_i, head(c_i))$$

Also $size(c_j) = 0$ and R does not have a deadlock.

b) The queues for channels belonging to C_1 are not empty. Let c_i be a channel belonging to C_1 with a nonempty queue such that there are no channels less than c_i with a nonempty queue. Again, there are two possible cases:

b1) c_i is a minimal. As shown above, it is also a sink and then, all the flits stored in its queue will be destined to d_i and the flit at the head of the queue for c_i is not blocked.

b2) c_i is not a minimal. Then

$$size(c_j) = 0 \quad \forall\, c_j \in C_1 \ni c_i > c_j$$

existing three possible cases:

b2.1) c_i has a header at the queue head. Taking into account that R_1 is connected

$$head(c_i) \neq d_i \Rightarrow \exists\, c_j \in C_1 \ni c_j \in R_1(d_i, head(c_i)) \Rightarrow$$
$$\exists\, c_j \in C \ni c_j \in R(d_i, head(c_i))$$

Also

$$c_i > c_j \Rightarrow size(c_j) = 0$$

and R does not have a deadlock.

b2.2) c_i has a data flit at the queue head, not destined to d_i and $next(c_i)$ belongs to C_1. Then

$$c_i > next(c_i) \Rightarrow size(next(c_i)) = 0$$

and R does not have a deadlock.

b2.3) c_i has a data flit at the queue head, not destined to d_i and $next(c_i)$ belongs to $C - C_1$. Let $c_1, c_2, \ldots c_k \in C - C_1$ be the set of channels reserved by the message after reserving c_i, c_k containing the message header. Those channels belong to $C - C_1$, because there are no channels less than c_i with a nonempty queue.

$$\exists\, c_1, c_2, \ldots c_k \in C - C_1 \ni$$
$$c_i \in R_1(s_i, head(c_k)) \wedge c_1 \in R(d_i, head(c_k)) \wedge$$
$$c_{m+1} \in R(d_m, head(c_k)) \quad \forall\, m \in \{1, k-1\}$$

As R_1 is connected

$$head(c_k) \neq d_k \Rightarrow \exists\, c_j \in C_1 \ni c_j \in R_1(d_k, head(c_k)) \Rightarrow$$
$$\exists\, c_j \in C \ni c_j \in R(d_k, head(c_k))$$

Thus, there is an indirect dependency from c_i to c_j ($c_i > c_j$), implying that $size(c_j) = 0$. Then, the header at the queue head for c_k is not blocked and R does not have a deadlock.•

Again, there are some interesting considerations:

1) The basic idea behind theorem 2 is that one can have an adaptive routing function with cyclic dependencies between channels, provided that there are alternative paths without cyclic dependencies to send a given flit towards its destination. As messages are several flits long, the extended channel dependency graph must be used to take into account the indirect dependencies.

2) If it were not necessary emptying a queue before accepting the header of another message, then it would be no guarantee that header flits occupy the queue heads and the theorem would not be valid. Consider, for instance, a set of three or more channels with cyclic dependencies between them and a configuration in which the queues of those channels are full, each one containing the tail of a message followed by a fragment of another message destined two nodes away. The rest of that message occupies part of the next channel queue and so on. That configuration is deadlocked because the header flits do not occupy the queue heads and cannot be routed using the alternative paths offered by the routing function.

3) If the routing function were defined as $R: C \times N \rightarrow C^p$, then the theorem would not be valid. Consider, for instance, two subsets of C, namely, C_1 and $C - C_1$, and a routing function defined in such a way that all the messages arriving to a given node through a channel belonging to $C - C_1$ are routed through a channel belonging to the same subset. Suppose that there are cyclic dependencies between the channels belonging to $C - C_1$ and that C_1 defines a routing subfunction which is connected and has no cycles in its channel dependency graph. That routing function is not deadlock-free.

4) The routing subfunction R_1 is not necessarily static. It can be adaptive.

3. Definitions and theorems for store-and-forward routing

This section develops the theoretical background for the design of deadlock-free adaptive routing algorithms for networks using store-and-forward routing. This theory is almost directly derived from the theorems proposed in section 2 for the particular case of messages consisting of a single flit. Thus, this theory avoids deadlock by restricting routing instead of buffer allocation.

The basic assumptions are very similar to the ones proposed in section 2, except that store-and-forward routing is used and then, all the packets have routing information and full packets are stored in each queue buffer. The assumption 6 has no meaning here.

The definitions are also identical, except those ones referring to deadlocked configurations.

Definition 12: A *configuration* is an assignment of a list of nodes to each queue. The number of packets in the queue for channel c_i will be denoted $size(c_i)$. The destination node for a packet f_j will be denoted $dest(f_j)$. If the first packet in the queue for channel c_i is destined for node n_d, then $head(c_i) = n_d$. A configuration is *legal* iff

$$\forall\, c_i \in C, \quad size(c_i) \leq cap(c_i) \quad \wedge$$
$$c_i \in R(s_i, dest(f_j)) \quad \forall\, f_j \in queue(c_i)$$

that is, the queue capacity is not exceeded and all the packets stored in the queue have been sent there by the routing function.

Definition 13: A *deadlocked configuration* for a given interconnection network I and routing function R is a nonempty legal configuration verifying the following condition:

$$\forall\, c_i \in C \quad \exists \quad head(c_i) \in N \quad \Rightarrow \quad head(c_i) \neq d_i \quad \wedge$$
$$size(c_j) = cap(c_j) \quad \forall\, c_j \in R(d_i, head(c_i))$$

In a deadlocked configuration there is not any packet one hop from its destination. Packets cannot advance because the queues for all the alternative output channels supplied by the routing function are full. As a particular case (for disconnected routing functions), the routing function may not supply any output channel.

The theorem 1 proposed in section 2 is also valid for store-and-forward routing, the proof being almost identical. In this section, two theorems are proposed. The first one is similar to theorem 2 for wormhole routing, except that the channel dependency graph is used instead of the extended one. The second theorem makes the design of adaptive routing functions more flexible. For each theorem, a sketch of the proof is given.

Theorem 3: A connected and adaptive routing function R for an interconnection network I is deadlock-free if it exists a subset of channels $C_1 \subseteq C$ that defines a routing subfunction R_1 which is connected and has no cycles in its channel dependency graph D_1.

Proof sketch:

$\Leftarrow$ The proof is basically the same as in section 2, changing nonempty queue by full queue and empty queue by nonfull queue. Also, cases b2.2 and b2.3 do not exist. Then, only the channel dependency graph for R_1, instead of the extended one, is required to be acyclic.

Theorem 4: A connected and adaptive routing function R for an interconnection network I is deadlock-free if it exists a subset of channels $C_1 \subseteq C$ that defines a connected and deadlock-free routing subfunction R_1.

Proof sketch:

$\Leftarrow$ If $C_1 = C$ the proof is trivial. Otherwise $C_1 \subset C$. Suppose that there is a deadlocked configuration for R. There are two possible cases:

a) The queues for channels belonging to C_1 are empty. As R_1 is connected, a given packet can be routed using an empty channel belonging to C_1 and there is no deadlock.

b) The queues for channels belonging to C_1 are not empty. As R_1 is deadlock-free, one can find a channel $c_i \in C_1$ such that $c_j \in R_1(d_i, head(c_i))$ and $size(c_j) < cap(c_j)$. As $C_1 \subset C$ and $R_1 \subset R$ then there is not a deadlock, contrary to the initial assumption.

4. Design methodologies

In this section we propose two methodologies for the design of deadlock-free adaptive routing algorithms. Although the same methodologies can be applied for both, wormhole and store-and-forward routing, there are some differences that will be highlighted.

The generation of static deadlock-free routing algorithms requires to restrict routing by

removing edges from D to make it acyclic. If it is not possible to make D acyclic without disconnecting the routing function, edges can be added to D by splitting physical channels into a set of virtual channels, each one requiring its own buffer. This technique was introduced by Dally [9] to remove cycles from the channel dependency graph.

However, a physical channel can be split into more virtual channels than the ones strictly necessary to avoid deadlock [6,10]. In such a case, the router can choose among several channels to send a message, reducing channel contention and message delay. Alternatively, more physical channels can be added to each node, increasing the network bandwidth and allowing the design of fault-tolerant adaptive routing algorithms.

A design methodology must supply a way to add channels following a regular pattern, also deriving the new routing function from the old one. A design methodology based on theorem 1 has been presented in [10]. Although the algorithms designed with it behave better than the static ones, a higher degree of freedom can be obtained basing the design either on theorem 2 (wormhole) or on theorems 3 and 4 (store-and-forward). Here we will present some more general methodologies for the design of deadlock-free adaptive routing algorithms.

Methodology 1. This methodology is intended to increase the number of valid alternative paths to send a message towards its destination without increasing the number of physical channels. In general, it will reduce channel contention and message delay but it will not increase fault-tolerance significantly. The steps are the following:

1) Given an interconnection network I_1, define a minimal path connected static routing function R_1 for it, following Dally's methodology and splitting physical channels into virtual ones, if necessary, to guarantee that R_1 is deadlock-free. Alternatively, define a minimal path connected adaptive routing function R_1 and selection function S_1, verifying that R_1 is deadlock-free using either theorem 2 or theorem 3. Let C_1 be the set of channels at this point.

2) Split each physical channel into a set of additional virtual channels. Let C be the set of all the (virtual) channels in the network. Let C_{ij} be the set of output channels from node i belonging to a minimal path from i to j. Define the new routing function R as follows:

$$R(i, j) = R_1(i, j) \cup (C_{ij} \cap (C - C_1)) \quad \forall\, i, j \in N$$

that is, the new routing function can use any of the new channels belonging to a minimal path or, alternatively, the channels supplied by R_1. The selection function can be defined in any way.

3) For wormhole routing only, verify that the extended channel dependency graph for R_1 is acyclic.

Step 1 establishes the starting point. We can use either a static or adaptive routing function as the basic one. Dally's theorem and theorems 2 and 3 can be used to verify that the basic function is deadlock-free. Step 2 indicates how to add more (virtual) channels to the network and how to define a new adaptive routing function from the basic one. Step 3 allows us to verify that the new routing function is deadlock-free. This step is only required for wormhole routing.

For store-and-forward routing, it is easy to see that the proposed methodology directly supplies a deadlock-free routing function R, because

$R_1(i, j) = R(i, j) \cap C_1 \quad \forall\, i, j \in N$

So, it exists a routing subfunction R_1 of R, which is connected and deadlock-free. Then, by theorem 4, R is deadlock-free.

It must be noticed that the methodology can also be applied by adding physical channels instead of virtual ones. The resulting network will be faster and more expensive, but the effective fault-tolerance will not increase. The reason is that the new routing function relies on the set of channels C_1 to guarantee that it is deadlock-free.

Methodology 2. This methodology is intended to increase fault-tolerance in a network. It will add physical channels, instead of splitting channels into virtual ones. Of course, it will also reduce channel contention and message delay. The steps are the following:

1) Given an interconnection network I_1, define a static or adaptive connected routing function R_1 for it, following Dally's methodology, the above proposed methodology or verifying that R_1 is deadlock-free using either theorem 2 or theorem 3. Let C_1 be the set of channels at this point.

2) Duplicate each physical channel. If the original channel was split into several virtual channels, the duplicated channel will also be split into the same number of virtual channels. Let C_2 be the set of duplicated channels and C the set of all the channels. Let R_2 be a routing function identical to R_1, but defined using C_2 instead of C_1. Define the new routing function R as follows:

$R(i, j) = R_1(i, j) \cup R_2(i, j) \quad \forall\, i, j \in N$

that is, the new routing function can use any of the channels supplied by both, R_1 and R_2. Define the selection function giving to the channels belonging to C_1 and C_2 the same probability of use.

Again, step 1 supplies the basic routing function and step 2 adds alternative paths. As can be easily seen, R_2 does not add any cycle neither to the channel dependency graph of R_1 nor to the extended one. Then, R is deadlock-free.

The duplication of channels defines an interconnection network $I_2 = G(N, C_2)$, which is identical to I_1 and shares the same set of nodes N. However, C_1 and C_2 are disjointed sets. R_2 has the same properties as R_1. Also,

$R_1(i, j) = R(i, j) \cap C_1 \quad \forall\, i, j \in N$

$R_2(i, j) = R(i, j) \cap C_2 \quad \forall\, i, j \in N$

So, one can find, at least, two subfunctions of R, which allow us the application of the corresponding theorem to guarantee that R is deadlock-free. Then, the theorems can be applied even if we remove some channels either from C_1 or from C_2.

However, the set of nodes is the same for I_1 and I_2. It seems that the proposed methodology is not tolerant to node faults. But, provided that R_1 and R_2 are adaptive routing functions, in general there will be alternative paths to reach the destination node (assuming that it is not the faulty one). Of course, some mechanism is needed to identify faulty channels, marking them as busy, and faulty nodes, marking all the channels connected to them as busy and avoiding to send messages to them. It must be noticed that if there is not any faulty node, the information about faulty channels is recorded locally.

Finally, step 2 can be applied several times, duplicating each channel as many times as desired.

The proposed methodologies are very simple to apply. They illustrate the power of the theorems. More complex design methodologies can be defined based on the same theoretical background.

As an example, we will present a design based on the above proposed methodologies. Consider a binary n-cube. We will study three cases: a) applying methodology 1; b) applying methodology 2; c) applying methodologies 1 and 2.

a) For the step 1 we can use the conventional static routing algorithm for the binary n-cube. It forwards messages crossing the channels in order of decreasing dimensions. It is well known that this routing function is connected and deadlock-free.

For the step 2, consider that each physical channel c_i has been split into two virtual channels, namely, a_i and b_i. Let C_1 be the set of 'b' channels. The algorithm obtained applying the step 2 can be stated as follows: Route over any useful dimension using 'a' channels. Alternatively, route over the highest useful dimension using 'b' channels. A useful dimension is one that forwards a message nearer to its destination.

It can be seen that the extended channel dependency graph for R_1 is acyclic. Then, R is deadlock-free.

As virtual channels share a single physical channel, the former algorithm effectively allows messages to cross the physical channels corresponding to the n-cube dimensions in any order, increasing considerably the number of alternative paths and decreasing network contention. The simulation results for that algorithm are very promising.

b) Assume that step 1 is applied as in case a), obtaining the conventional static routing algorithm.

For the step 2, consider that each physical channel c_i has been duplicated, obtaining a new channel d_i. The algorithm obtained applying the step 2 can be stated as follows: Route over the highest useful dimension using 'c' channels. Alternatively, route over the highest useful dimension using 'd' channels.

That algorithm increases the tolerance to faulty channels, but it does not take advantage of alternative minimal paths.

c) Assume that we apply the methodology 1, obtaining the algorithm proposed in case a). That algorithm constitutes the step 1.

For the step 2, consider that each physical channel c_i has been duplicated, obtaining a new channel d_i, which is split into two virtual channels, namely, e_i and f_i. The algorithm obtained applying the step 2 can be stated as follows: Route over any useful dimension using either 'a' or 'e' channels. Alternatively, route over the highest useful dimension using either 'b' or 'f' channels.

That algorithm has the advantages of the previous ones at the cost of a slightly more complicated circuitry.

As stated in section 2, the selection function only affects performance. It is not necessary to give a higher priority to the channels in the acyclic dependency subgraph,

because when the remaining channels are busy those ones will be used. In general, a higher performance is achieved when the channels in the cyclic dependency subgraph are given a higher priority, because they usually offer a larger number of alternative paths.

Also, the selection function can be extended by including additional information in its domain. For instance, for the algorithm obtained in case a), it is possible to favour the 'a' channel connecting to the neighbour with a higher number of free channels in useful dimensions. This selection function is inspired in an algorithm proposed by Reeves et al. [22], the main difference being that our algorithm does not require a complex mechanism to abort messages because it is deadlock-free.

Another interesting extension of the selection function is taking into account the time a message is waiting in a given node. This information can be used to prevent channel multiplexing when the network traffic is low. That extension gives good results when added to the algorithm obtained in case a). The simulation results will be presented in another paper.

More examples could be presented for other topologies. However, the application of the above proposed methodologies is so easy that we consider that it is not necessary. The simulation under different load conditions will give some insight about the performance of the new family of adaptive algorithms.

5. Conclusions

The theoretical background for the development of deadlock-free adaptive routing algorithms has been proposed for both, store-and-forward and wormhole routing. Firstly, a straightforward extension of Dally's theorem has been presented, allowing the design of adaptive algorithms. However, the absence of cycles in the channel dependency graph is too restrictive.

For wormhole routing, theorem 2 gives a more flexible condition for the development of adaptive algorithms, by allowing the existence of cycles in the channel dependency graph. The only requirement is the existence of a channel subset which defines a connected routing subfunction with no cycles in its extended channel dependency graph.

For store-and-forward routing, theorem 3 develops a sufficient condition similar to theorem 2. Theorem 4 adds more flexibility. It simply requires the existence of a connected and deadlock-free routing subfunction. In turn, that subfunction can be proved to be deadlock-free using theorem 3.

To simplify the application of the theorems, two design methodologies have been proposed. The first one supplies adaptive algorithms with a high degree of freedom. The second one gives a way to design fault-tolerant routing algorithms. Both methodologies can be combined easily.

Finally, an example showing three alternative ways to apply the proposed design methodologies is presented.

References

[1] W.C. Athas and C.L. Seitz, Multicomputers: message-passing concurrent computers, *Computer,* Vol. 21, No. 8, pp. 9-24, August 1988.

[2] S. Borkar et al., iWarp: an integrated solution to high-speed parallel computing, *Supercomputing'88,* Kissimmee, Florida, November 1988.

[3] W. Chou, A.W. Bragg and A.A. Nilsson, The need for adaptive routing in the chaotic and unbalanced traffic environment, *IEEE Trans. Commun.,* Vol. COM-29, No. 4, pp. 481-490, April 1981.

[4] E. Chow, H. Madan, J. Peterson, D. Grunwald and D.A. Reed, Hyperswitch network for the hypercube computer, *Proc. 15th Int. Symp. Computer Architecture,* Honolulu, May-June 1988.

[5] W.J. Dally, *A VLSI architecture for concurrent data structures,* Kluwer Academic Publishers, 1987.

[6] W.J. Dally, Virtual-channel flow control, *Proc. 17th Int. Symp. Computer Architecture,* Seattle, Washington, May 1990.

[7] W.J. Dally, Performance analysis of k-ary n-cube interconnection networks, *IEEE Trans. Computers,* Vol. C-39, No. 6, pp. 775-785, June 1990.

[8] W.J. Dally and C.L. Seitz, The torus routing chip, *Distributed Computing,* Vol. 1, No. 3, pp. 187-196, October 1986.

[9] W.J. Dally and C.L. Seitz, Deadlock-free message routing in multiprocessor interconnection networks, *IEEE Trans. Computers,* Vol. C-36, No. 5, pp. 547-553, May 1987.

[10] J. Duato, Deadlock-free adaptive routing algorithms for multicomputers. Submitted to *Tech. et Sci. Informatiques.*

[11] J. Duato, On the design of deadlock-free adaptive routing algorithms for multicomputers: theoretical aspects, *Proc. 2nd European Distributed Memory Computing Conference,* Munich, April 1991.

[12] D. Gelernter, A DAG-based algorithm for prevention of store-and-forward deadlock in packet networks, *IEEE Trans. Computers,* Vol. C-30, pp. 709-715, October 1981.

[13] C. Germain-Renaud, *Etude des mécanismes de communication pour une machine massivement parallèle: MEGA,* Ph.D. dissertation, Université de Paris-Sud, Centre d'Orsay, 1989.

[14] K.D. Gunther, Prevention of deadlocks in packet-switched data transport systems, *IEEE Trans. Commun.,* Vol. COM-29, pp. 512-524, April 1981.

[15] W.D. Hillis, *The connection machine,* MIT Press, Cambridge, Mass., 1985.

[16] C.R. Jesshope, P.R. Miller and J.T. Yantchev, High performance communications in processor networks, *Proc. 16th Int. Symp. Computer Architecture,* Jerusalem, Israel, May-June 1989.

[17] P. Kermani and L. Kleinrock, Virtual cut-through: a new computer communication switching technique, *Computer Networks,* Vol. 3, pp. 267-286, 1979.

[18] C.K. Kim and D.A. Reed, Adaptive packet routing in a hypercube, *Proc. 3rd Conf. on Hypercube Concurrent Computers & Applications,* Pasadena, California, January 1988.

[19] H.T. Kung, Deadlock avoidance for systolic communication, *Proc. 15th Int. Symp. Computer Architecture,* Honolulu, May-June 1988.

[20] P.M. Merlin and P.J. Schweitzer, Deadlock avoidance in store-and-forward networks - I: Store-and-forward deadlock, *IEEE Trans. Commun.,* Vol. COM-28, pp. 345-354, March 1980.

[21] S. Ragupathy, M.R. Leutze and S.R. Schach, Message routing schemes in a hypercube machine, *Proc. 3rd Conf. on Hypercube Concurrent Computers & Applications,* Pasadena, California, January 1988.

[22] D.S. Reeves, E.F. Gehringer and A. Chandiramani, Adaptive routing and deadlock recovery: a simulation study, *Proc. 4th Conf. on Hypercube Concurrent Computers & Applications,* Monterey, California, March 1989.

A Toolkit for Debugging Parallel Lisp Programs

Hermann Ilmberger, Sabine Thürmel
*EDS Project**
Siemens AG, ZFE IS SOF 22
Otto-Hahn-Ring 6, D-8000 München 83, Germany
e-mail: thuermel@ztivax.uucp, hermann%moony@ztivax.uucp

Abstract

As part of the ESPRIT-II project EDS a toolkit (called Delphi) is under development for the debugging of Lisp programs with explicit parallelism being executed on a homogeneous distributed machine. It assists in the detection of functional and synchronisation errors. It also helps to detect unexpected nondeterminacy and sources of poor program performance. Specific mechanisms allow the user to effectively control several processes in a debugging session. The paper introduces the basic concepts behind the tools and how the user may benefit from them.

1 Motivation and Overview

Many experts consider parallel computation the only promising approach to enhancing the performance of computer systems. This is not only true for numerical algorithms. Significant increases in performance are also expected for data intensive applications using nontrivial algorithms such as the translation of natural languages [EDS89]. This motivates to combine parallel computation and symbolic languages. Such a combination is the goal of the ESPRIT-II project 2025 EDS (European Declarative System). The project will produce the prototype of a homogeneous distributed machine supporting among other things a parallel Lisp dialect (EDS Lisp). In EDS Lisp processes are spawned by the future construct. Communication may be realized through mailboxes. EDS Lisp is intended to support large knowledge based systems, for example for the translation of natural languages or VLSI chip design. As a consequence there is the need for a toolkit that assists users of EDS Lisp in debugging applications consisting of several concurrent processes.

This paper describes Delphi, a toolkit for the debugging and visualisation of EDS Lisp programs. Specifically Delphi addresses the problem of how the user

* The EDS-Project (European Declarative System) is partially sponsored by the European Community under ESPRIT-II 2025.

can keep an overview of the parallel program run and effectively control the debugging session.

The paper begins with a short introduction to EDS Lisp and the EDS Machine. Chapter 3 focuses on basic issues in debugging parallel Lisp programs. In chapter 4 an overview of Delphi is presented. Its tools for distributed symbolic debugging are described. It is outlined how Delphi supports user controlled program execution for debugging purposes. We also present a visualisation facility supporting the user in the granularity analysis and the detection of inadequate communication structures. Chapter 5 is devoted to additional, planned features. A comparison to other approaches follows in chapter 6.

2 EDS Lisp and the EDS Machine

EDS Lisp is a parallel Lisp dialect supporting explicit parallelism. It is an extension of Common Lisp [Steele 84]. Common Lisp has been chosen as the base language of EDS Lisp because one goal of the project is to port real life applications to the EDS system.

The following gives a brief overview of the main features of this language (for a detailed introduction see [HamHen90]):
For the spawning of processes EDS Lisp has a single concept, the future construct, known from other parallel Lisp extensions (e.g. [Halst85]). It allows concurrent function evaluation. After spawning a process the newly spawned process immediately returns a placeholder for the result. If there is an access to the result of a spawned process while the return value is not yet computed the spawning process is forced to wait by an implicit synchronisation mechanism.
One may also wait for the result of a process using the wait construct thus causing an explicit synchronisation. This construct can be used if one wants to wait for the sideeffects of a spawned process before resuming evaluation.
In addition EDS Lisp contains two other concepts for explicit synchronisation, critical sections and mailboxes. Critical sections are particularly well suited to synchronise accesses to shared global variables. Processes can communicate via mailboxes which provide asynchronous buffered message passing.

The EDS system is a homogeneous distributed multiprocessor. A prototype will consists of 63 processing elements and a diagnostic node. During the test phase the latter will serve as an interface between the 63 node EDS machine and the user's workstation. The Process and Store Model of EDS Lisp is described in detail in [HamHen90]. We won't refer to it in this paper because our debugging concepts are intentionally independent of the EDS system. So our ideas are also applicable to similar architectures.

3 Basic Issues in Debugging Parallel Lisp Programs

In this paper we want to address some of the basic issues in debugging parallel Lisp programs. They are listed below. The concepts we propose for their solution are introduced briefly. (Chapter 4 is devoted to their detailed description.)

Detecting functional errors is usually necessary for the detection of synchronisation errors

While debugging parallel Lisp programs the detection of functional errors (classically called bugs) cannot be neglected. For example, most synchronisation errors result from incorrect use of a combination of sequential and parallel constructs.
Therefore we propose a straightforward extension to classical symbolic debugging tools. This allows us to extend well known debugging strategies for the parallel case.

Concentrating on debugging one process at a time is not appropriate in debugging parallel programs

Symptoms of a problem may show up on another process rather than on the one where the problem actually occurred. Thus a single debugger examining one process at a time makes debugging a clumsy affair. Tools for distributed symbolic debugging are needed. To provide the user with a detailed view of the program state the tools have to provide a mechanism to compute nonlocal information.

Observing and controlling process states by the user is necessary

For the exact location and analysis of errors the programmer needs assistance in getting an overview of the general program state (e.g. which processes are active, what is their status). In addition the user would like to control the amount of trace information to be displayed (there will be times when the user simply wants to discard trace output). Also it is helpful if broken processes can be continued individually after being inspected. Thus a mechanism for the observation and control of processes is necessary.

Visualising the process history gives a comprehensive overview of the runtime behaviour

An overview focusing on a moment in time is not sufficient for the understanding of the possibly quite complex runtime behaviour of parallel programs. Graphical displays of the program execution have to be supported which allow getting an overview of the total program execution.We propose focusing on two essential aspects: the spawning/joining of processes and the

Supporting user controlled execution enhances the error detection

The debugging tools should enhance the detection of unwanted nondeterminacy. Nondeterminacy is not in itself an error and may be intentional. But careless unsynchronized access to global or dynamic variables may lead to nondeterminate results. Our clique model for the support of user controlled execution helps detecting sources of nondeterminacy.

4 Delphi: A Toolkit for Debugging Parallel Lisp Programs

The EDS Lisp debugger Delphi supports various debugging techniques. It consists of

* basic tools: stepper, tracer and break-mode,
* a mechanism allowing the computation of nonlocal information by communicating sequential debuggers,
* an interactive graphical process observation and control tool,
* a process history visualizer,
* the clique model for user controlled program execution.

Stepper, tracer and break-mode are common for sequential Lisp debuggers. These tools have been extended for use in the parallel environment of EDS Lisp. Special treatment is necessary if stepping/tracing/debugging takes place in several processes. In order not to confuse the programmer, the information is handled in different windows on the terminal, controlled by the user via the graphical representation of the Lisp process tree. Additionally, the debuggers can communicate to compute nonlocal information.

The visualizer offers two complementary views of a program execution, one focusing on the spawning of processes and one concentrating on the communication at mailboxes.

A clique is a group of Lisp processes running quasiparallel. Grouping and scheduling is user controlled. The clique model supports the programmer in detecting communication errors and nondeterminacy and can be used in the testing phase to simulate extreme scheduling behaviour.

4.1 Basic Tools

Stepper

The stepper interactively single-steps through the evaluation of a form. If the stepped form spawns a process, the user is asked whether the new process should be stepped, too. If yes, a new window is opened on the screen, and the user can single-step through both processes, thus controlling their relative

progress. As this method is very intrusive, it is only recommended when no timing conditions are involved.

Tracer

The user can specify a set of Lisp functions to be traced. Both Common Lisp functions and the new extensions for parallelism in EDS Lisp can be traced. Each program run logs a minimal trace. There is a separate window for the trace output of each process. To avoid a confusing display, the user can control the window action via a graphical representation of the process tree (ch. 4.3).

Break-Mode - the "Debugger"

According to the semantics of error and break in EDS Lisp, an error in one process of a program causes all processes of the program to be broken. There is a separate debugger for each broken process. Using the graphical representation of the process tree (ch. 4.3), the user can open a window for selected debuggers. Each user-opened debugger communicates with the user via its own window.

4.2 Communicating Sequential Debuggers to Gain Nonlocal Information

Since a program can consist of many processes, it is not convenient to open a window for all the corresponding debuggers. The programmer may want to handle all broken processes using only a few windows. Assume the following scenario:

When a whole EDS Lisp program is broken because of an error, the bug can be located

a) locally in the process A that raised the error, or

b) in one or more other processes which transferred erroneous data to process A.

In case b) the bug lies in other processes, but the error itself shows up in process A. So the programmer will probably first inspect process A, but will sooner or later realize that the real bug lies in some other process. If the process containing the bug is located, the programmer can simply open a window for the debugger of the process and continue debugging there.

If process A is a process that communicates frequently with other processes, it may be more difficult to determine the process(es) containing the bug. For such cases the EDS Lisp debuggers can communicate with each other using the "nonlocal" commands. Each debugger can be remote controlled from any other. The responses from remote debuggers are displayed in the controlling debugger's window. For the nonlocal commands the user can specify from which remote debuggers the information is requested.

Examples of nonlocal commands are:

* list the top-of-stack frames of all broken processes
* continue all processes with name foo
* list the value of the dynamic variable x in all processes (in EDS Lisp each process has its own view of the dynamic binding stack)

The above commands simply broadcast a request to the other debuggers and gather the answers. But there are also more powerful nonlocal commands:

* Which processes are executing the function foo? (requires searching through the backtrace of all processes)
* With which processes did a process communicate? (not directly visible in the program because processes send messages to mailboxes, not to processes)

Thus the nonlocal commands enable the user to get a global view of the intrinsics of a program. How are other existing parallel Lisp systems addressing the problem of debugging many processes?

The parallel Lisp system Mult-T [Kranz89] supports the concept of process groups. If one process in a group raises an error, all processes of this group are stopped. There is one window to debug the whole group. Debugger commands by default refer to the process in which the error occurred. The commands also allow referring to other processes or other stopped groups. But there are no nonlocal commands in our sense; debugger commands influence always only one process.

MultiScheme [Miller87] also stops all processes when an error occurs, but the user can debug only one process at all. Butterfly Lisp [Allen87] opens a separate window for each broken process.

The EDS Lisp Debugger supports this as an option. But often this is not desired. Therefore the opening of windows can be controlled by the user as described in the next section.

4.3 Graphical Process Observation & Control and Visualization of Runtime Behaviour

For locating and analysing errors, the programmer needs assistance in getting an overview of the current program state. But often the current state is not sufficient. An overview of the past program behaviour is also necessary for efficient debugging.

4.3.1 Graphical Process Observation and Control

The graphical observation and control tool gives a global view of the process dependencies and states. It is based on a graphical representation of the program's process tree. The behaviour of trace and debugger windows are controlled via the tree.

Each node in the tree represents a process. The children of a node are the directly spawned future-processes of the parent process, ordered by their time of creation. The root of the tree is the EDS Lisp program's toplevel.

The tree is kept in a separate window on the user's workstation screen. The pictorial representation of a node changes according to the state of the corresponding process. There are the following states:

a) process running
b) process wants to output trace information (but continues to run)
c) process is broken, its debugger is active
d) process completed

In case b) and c) the user can open a trace resp. debugger window by selecting a node. Thus, the user can manage the opening and closing of windows. There is no abundance of tracer and debugger windows to handle. Only the window of the process' debugger where the error occurred is automatically opened. The debuggers for other suspect processes can be opened at will. Processes that are not suspect can be continued, changing to state a).

There is no separate node state for active steppers, because stepper windows are automatically opened by the system. This makes sense since usually only one form is stepped at a time and the user is asked if both processes should be stepped after a future-command.

Opening a debugger or tracer window is one of many commands that may be performed on a node in the process tree display. Other commands are:

* break the process
* restart a broken process
* discard the trace output (if the user does not want to see the trace)
* list basic characteristics of the process (processing element, runtime, ...)
* visualize process history (see below)

Thus the process tree in combination with the node commands is a powerful tool to get a general overview of the current program state and to customise the window management.

4.3.2 Visualisation of Runtime Behaviour

Visualisation of the dynamic behaviour of EDS Lisp programs
* enhances the understanding of the possibly quite complex dynamic

behaviour of EDS Lisp programs,
* assists in finding errors,
* is the basis for granularity analysis performed by the user,
* supports the user in detecting inappropriate communication structures.

Alternative representations of the runtime behaviour include the one favored in literature (e.g. [GeKr86], [Bag89]). It represents the processes that existed during program execution as horizontal bands with arrows between the processes indicating interactions. It seems to be very helpful when one wants to concentrate on certain aspects of an execution. But even when such process graphs are hierarchically organised with adequate operators for hiding processes and their descendants, such systems are limited in the size of an application they can deal with.

Therefore we decided to offer two complementary views of program execution:
The first view focuses on the spawning of processes in an EDS Lisp program execution. It displays a general overview of the possibly quite complex runtime behaviour. It may serve as a basis for granularity analysis. This view is called a process history tree.
The second view depicts the use of EDS Lisp mailboxes. It assists in the detection of inappropriate communication structures in the EDS Lisp program.

Such displays may be generated during execution, displaying the behaviour up to that time, or may be generated post mortem. The process tree displayed in the tool for process observation and control is a snapshot of the program behaviour (i.e. long dead processes are not visible). The process history tree allows the inspection of the runtime behaviour up to the moment when the tree was built.

To produce these views, for every process a trace is generated consisting of basic events (as process creation and termination etc.) and user selected events during execution. At the user's workstation these traces are evaluated. Displays are generated upon request. Since the graphical representations are built offline, the delay of the program execution caused by the visualizer is minimal.

Process History Trees

A process tree for the execution of an EDS Lisp program shows the spawning structure of the EDS Lisp processes generated during the execution.This is exemplified in fig. 1.

To navigate these displays, horizontal and vertical scrollbars are provided allowing depth first and breadth first search. For the hiding/unhiding of processes and their descendents appropriate operators are supported.

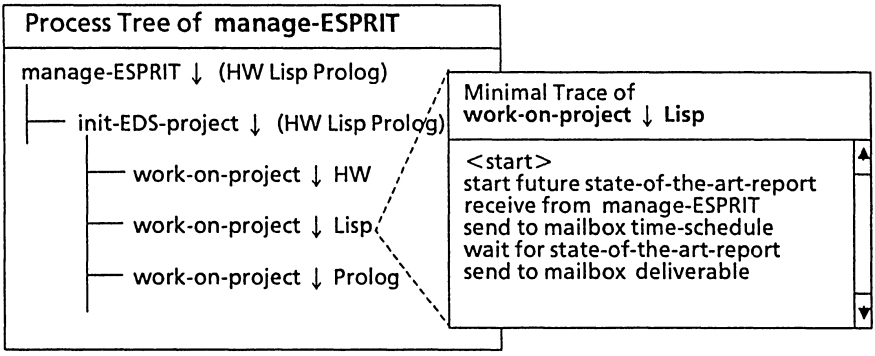


fig. 1: A sample process-tree and minimal trace.
legend: ↓ actual parameters of a process

For each process its process specific trace (process history) as well as some of its basic characteristics (like active and idle time) and number of remote accesses may be displayed upon request.

The trace allows inspecting process specific events. In combination with its basic (system) characteristics, granularity analysis may be performed by the user: An extremely short active time will indicate that the process should be evaluated inline rather than in parallel. Long idle times in combination with a large amount of remote accesses may be a hint to look for inappropriate variable bindings. Thus this display may help the programmer to tune the performance on the EDS Lisp level.

Display of Communication via Mailboxes

The displays of the communication via mailboxes assist the user in finding inappropriate communication structures. Similar to [BuMil88] we want to display all send and receives to and from any user selected mailbox. In concrete terms: The y-axis presents time. The x-axis holds the information on the send and receive operations as demonstrated in figure 2. Every send operation increases the access counter, while a receive operation reduces the access counter. So the region on the negative side of the chart gives an impression on blocked and waiting processes, while the positive side visualizes the number of stored messages. By clicking on a specific access counter value in the display the corresponding message resp. waiting queue is displayed (see fig. 2). For any message the following information is given: who sent the message and who, if any, received it and its identification on source code level (the latter is not represented in the example). Its contents are not displayed because it costs too much time recording it during runtime since any Lisp object may be sent. The waiting queue can be used to identify the processes waiting at that mailbox.

Comparing the waiting queues of different mailboxes when all processes are broken or killed makes deadlock detection feasible. Upon request the representations of the processes involved in the traffic at a specific mailbox will be highlighted in the corresponding process tree. After detecting a disadvantageous communication structure in the mailbox display, the process specific traces of the process tree will help to improve it.

4.4 Support of User Controlled Execution

An EDS Lisp program can spawn several future-processes at runtime. A major problem in debugging parallel programs is that there is more than one process running at the same time. How can the programmer keep the overview and how can s/he handle the parallel processes to find bugs?

One common method is to run the program and wait until it crashes. Using the state of the program (i.e. variable bindings, Lisp stacks, ...) at the moment of the crash, many errors can be detected post mortem. Support for this method was discussed in chapter 4.2 and 4.3.

Sometimes this kind of post mortem detection is not sufficient. If an error occurs in one process of a Lisp program, the semantics of EDS Lisp says that all processes of the program will be broken. Suppose a Lisp program consists of two processes A and B running on different processing elements. If an error occurs in process A, A is broken immediately. Process B can still change its state in the time until the break signal reaches B's processing element. B can for example change the value of a global variable. Post mortem analysis will be difficult if just the former value of this variable caused the error in process A.

We developped the clique model to address this problem.

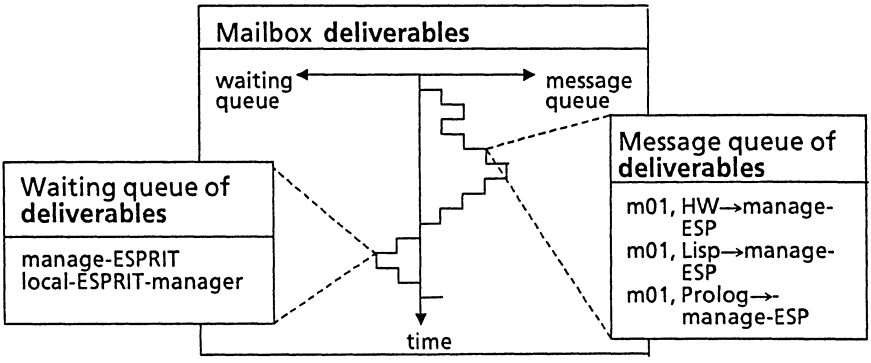


fig. 2: A sample communication at a mailbox

The Clique Model

The EDS Lisp debugger allows the user to control the sequence of execution by grouping processes together (into "cliques") and forcing them to run in a quasiparallel manner with a user defined Lisp level scheduling. If an error occurs in one of the clique processes, the other clique members are not allowed to continue and thus cannot change the program state.

Cliques support not only post mortem analysis but can be used even more profitably in the test phase. They can be used to test the program under unusual scheduling conditions (for example scheduling after each function call), and help programmers familiar with sequential languages get a better understanding how a parallel program works.

Membership

A "clique" is a group of processes which the programmer wants to serialise. Each Lisp process can become a member of (at most) one clique. A process which is not a member of a clique is "clique-free". (for an example see fig. 3)

Scheduling

At most one member of each clique is in the state 'computing' at any moment of time (quasiparallel run). Scheduling within a clique is controlled by the programmer, is visible at the source code level, and is repeatable. Timeslice scheduling is not reasonable, because the exact point of descheduling cannot be controled. Repeatability cannot be guaranteed either.

The straightforward kind of scheduling is function based. The programmer can supply a list of function names, and scheduling takes place before the call or

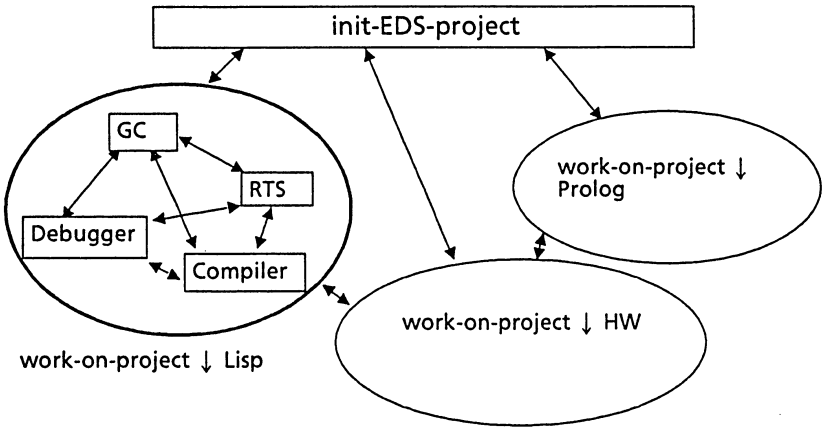


fig. 3: A sample of cliques
legend: ⟷ communication link via a mailbox

after the exit of each of these functions. Changing the list enforces a different scheduling. The scheduling can also be set to 'verbose'. In this mode each (de)scheduling action is printed out at runtime and the user can follow the program.

Application Areas

We see three main applications for cliques:

1. Tracing communication and synchronisation errors: the programmer conjectures that a mailbox communication or a critical section does not work well. Scheduling after the suspected command makes it possible to test whether it cooperates well with the other clique members.

2. Testing unusual scheduling: the programmer can simulate any possible scheduling in order to test highly communicating program parts and to detect unexpected nondeterminacy.

3. Programmers familiar with sequential languages often have difficulties programming parallel applications. Serialization of a program with verbose scheduling may help in better understanding what happens in a program.

Use

What strategy should be used to group processes together?

For programs with only a few processes where the programmer suspects nondeterminacies or communication errors, all processes can be grouped into one clique. There are two possibilities:

1. Shared mode: The clique runs on one processing element. Thus all processes have real shared memory. Errors caused by inconsistent use of distributed memory cannot occur. Furthermore all processes have the same global time, making it possible to totally order the events in the program.

2. Distributed mode: The processes are distributed to the processing elements "as in real life". This makes better use of the EDS machine's storage capacity. Now all the effects of distributed memory can occur. Total ordering is still possible since only one clique member runs at a time.

If the program gets bigger, a bottom up test strategy can be used: In a program there are often groups of processes which cooperate more with each other than with the rest of the program. These processes can form a clique. Several such cliques may be found. These cliques can now be analysed separately, and the other processes are perhaps replaced with stubs. Larger program systems can be tested incrementally. A process is already tested, can be made clique-free and run asynchronously.

5 Additional Features: Replay and Countercheck Sessions

In general, concurrent program runs cannot be repeated exactly, because there may be races between processes. There are two main aspects why a programmer wants to repeat a run:

a) After the occurence of an error it is very useful to be able to replay a program exactly in order to see how the error came into being: "replay".

b) A program shall be tested for nondeterminism. In this case a first run ("initial session") will be compared against a second, third etc. ("countercheck sessions") with same input data, the latter having for example different scheduling strategies and/or different machine load.

Replay

Replay techniques are an ideal approach to reproducing program execution([LeBl87], [Mill88]). Unfortunately they only allow reproducing such program executions where all shared data structures are known at compile time. This is usually not the case for programs written in parallel extensions to Common Lisp. Therefore we do not want to guarantee replay for unsynchronized global variable access (the effect of this is undefined in EDS Lisp anyway). We only want to guarantee replay for mailbox communication (mailboxes are created explicitely) and for access to critical sections (they are created explicitely, too). Inside the critical sections the programmer can access global variables in a well-defined manner. The replay does not need to know which variables are shared as long as all accesses happen within critical sections.

Countercheck Sessions

In a countercheck session the logged events of the initial session are compared to the current run. Any deviation signals a nondeterminism in the program. It is selfevident that countercheck sessions can only help to find nondeterminism, not to prove the absence of nondeterminism. Countercheck sessions seem to come almost for free with replay.

6 Other Approaches to Debugging Parallel Programs

Although there exists a variety of parallel Lisp dialects ([Tar89]) only very few pointers to debugging concepts for these languages can be found. We know of several people working in this field. For example, Bert Halstead is working on visualising Multilisp program execution based on [Bag89]. Later on, other debugging techniques will be integrated into this tool [Halst90].

Apart from work in progress there exists a traditional debugging toolkit

Lisp [TopL89]. It runs in a shared environment and supplies assistance for debugging several processes simultaneously in a traditional manner.

In contrast to the debugging of parallel Lisp dialects there exists a whole variety of debugging techniques for sideeffect free languages or for languages where shared data structures are known at compile time ([Jell90]). Behavioural abstraction techniques, replay mechanisms and static analysis tools are among the most well known.

Behavioural Abstraction ([Bate88], [Baia86]) allows comparing the expected and actual runtime behaviour using specifications based on predefined event classes and operators for their combination. Filter- and clustering techniques make it possible to abstract from unimportant details. But all possibly interesting events have to be specified before program execution.

To reproduce runtime behaviour replay techniques were developed. In [LeBl87] only accesses to shared data are logged. These protocols then control the reexecution. Additional techniques are necessary for the actual debugging. The same is true for [Mill88]. Here a (minimal) program graph is constructed during the (initial) execution. The thus obtained information can by extended by partial reexecution (incremental tracing). The adaption of this concept for EDS Lisp is briefly outlined in chapter 7.

Static analysis techniques were developed to parallelize programs automatically (e.g. [Harr89]). Other approaches allow the detection of potential nondeterminacy in procedural languages ([Emra88], [Bala88], [Call88]). Potential Nondeterminacy means that by solely using the proposed techniques of static analysis it is not decidable whether a certain statement may contribute to a nondeterminate program result. In such cases tracing is necessary. We are investigating how such techniques could be helpful for EDS Lisp. Although static techniques are only of very restricted use in a language such as Common Lisp which is list oriented and which allows dynamic function definition we think such techniques an elaborate pendant to the use e.g. of masterscope ([Inter85]) in sequential Lisp.

7 Conclusion

Delphi is a toolkit for debugging the parallel Lisp dialect EDS Lisp. Although being developed for the distributed EDS system, its debugging strategies are applicable as well to similar distributed systems and shared memory architectures.

Delphi contains tools for distributed symbolic debugging. In addition to classical debugging strategies, the tools allow the computation of nonlocal information. Thus a detailed view of the global state of the parallel program may be obtained in break-mode, whereas the visualizer of the process history and mailbox communication offers a global view of the process dependencies

their communication structure. The visualizer assists in the granularity analysis of parallel Lisp programs and allows the detection of inappropriate communication structures in the EDS Lisp program.Together these tools provide a comprehensive overview of the program.
The debugging session is controlled by the user: The tool for observation and process control makes it possible to inspect the state of the different processes and reducing the amount of information displayed. The developped clique model is a mechanism for user controlled execution. It enhances the detection of unexpected nondeterminacy.

Delphi contains flexible debugging tools that allow the programmer to focus on details as well as to get a general overview of a program execution.

References

[Allen87] D. Allen, S. Steinberg, L. Stabile
Recent developments in Butterfly Lisp, AAAI 87, Seattle, July 1987, pp. 2-6

[Bag89] Laura Bagnell
ParVis: A Program Visualization Tool for Multilisp, S.M. thesis, MIT E.E.C.S. Dept., Cambridge, Ma, Feb. 1989

[Baia86] Fabrizio Baiardi, Nicoletta De Francesco, Gigliola Vaglini
Development of a Debugger for a Concurrent Language, IEEE Transactions on Software Engineering, Vol.SE-12(4), April 1986, pp. 547-553

[Bala88] Vasanth Balasundaram, Donn Bäumgartner, David Callahan, Ken Kennedy, Jaspal Subhlok
PTOOL: A System for Static Analysis of Parallelism in Programs, Rice University, Computer Science Technical Report TR88-71, June, 1988

[Bate88] Peter Bates
Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison, WI, USA, May 5-6, 1988, pp. 11-22

[BuMil88] Helmar Burkhart, Roland Millen
Performance-Measurement Tools in a Multiprocessor Environment, IEEE Transactions onComputers, Vol.38,No. 5, May 1989

[Call88] David Callahan, Jaspal Subhlok
Static-Analysis of Low-level Synchronization, Proceedings of the

ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison, WI, USA, 1988, pp. 100-111

[EDS89] Carsten Hammer et al.
Volume 5 (Part 2) Language Subsystems The Lisp Subsystem, ESPRITII EP2025, Document:EDS.DD.5S.0001, Dez. 1989

[Emra88] Perry A. Emrath, David A. Padua
Automatic Detection of Nondeterminism in Parallel Programs, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison, WI, USA, May 5-6, 1988, pp. 89-99

[Fid88] C. J. Fidge
Partial Orders for Parallel Debugging, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison, WI, USA, May 5-6, 1988, pp. 183-194

[GeKr86] Patrick F. McGehearty, Edward J. Krall
Potentials for Parallel Execution of Common Lisp Programs, Proceedings of the 1986 International Conference on Parallel Processing, IEEE, pp.696-702

[Harr89] Williams Ludwell Harrison III
The Interprocedural Analysis and Automatic Parallelization of Scheme Programs, Lisp and Symbolic Computation, Vol.2 No3/4, Okt.1989,pp.185-391

[Halst85] R. Halstead
Multilisp: A Language for Concurrent symbolic Computation, ACM Transactions on Programming Languages and Systems, Okt. 1985

[Halst90] R. Halstead
private communication

[Inter85] Xerox Cooporation
Interlisp-D Reference Manual I-III, Okt.1985

[HamHen90] Carsten Hammer, Thomas Henties
Parallel Lisp for a Distributed memory Machine, Proc. of the EUROPAL workshop on "High Performance and Parallel Computing in Lisp", Nov. 1990, Twickenham, UK

[Jell90] Sylvia Jell
Parallel Debugging - State of the Art Report, ESPRIT-II EP2025, Document: EDS.WP.8S.0002, Mar. 90

[Kranz89] David A. Kranz, Robert H. Halstead Jr., Eric Mohr
Mult-T: A High-Performance Parallel Lisp, SIGPLAN 1989 Sympo-

sium on Programming Language Design and Implementation, Portland, Oregon, June 1989

[LeBl87] Thomas J. LeBlanc, John M. Mellor-Crummey
Debugging Parallel Programs with Instant Replay, IEEE Transactions on Computers, Vol.C-36(4), April 1987, pp. 471-482

[Mill88] Barton P. Miller, Jong-Deok Choi
A Mechanism for Efficient Debugging of Parallel Programs, Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, USA, June 22-24, 1988, pp. 135-144

[Miller87] J. Miller
MultiScheme: A Parallel Processing System Based on MIT Scheme, Ph.D. Thesis, M.I.T. E.E.C.S. Dept., Cambridge, Mass., August 1987

[Sto88] Janice Stone
A graphical representation of concurrent processes, Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison, WI, USA, May 5-6, 1988, pp. 226-235

[Stee84] Guy Steele
Common LISP: The Language, Digital Press, 1984

[Tar89] Jolan M. Targonski, Parallel Lisp Languages - the State of the Art, ESPRITII EP2025, Document: EDS.WP.5S.0001, Jul.1989

[ToLe90] Top Level, Inc.

Loosely-Coupled Processes (Preliminary Version)

Jayadev Misra*
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
(512) 471-9547
misra@cs.utexas.edu

1 Introduction

1.1 Message Communicating and Shared-Variable Systems

A system of processes in which the interactions are solely through messages is often called *loosely-coupled.* Such systems are attractive from a programming viewpoint. They are designed by decomposing a specification into its separable concerns, each of which could then be implemented by a process; the operation of the system can be understood by asserting properties of the message sequences transmitted among the component processes. A key attribute of loosely-coupled systems is a guarantee that a message that has been sent cannot be unsent. As a consequence, a process can commence its computation upon receiving a message, with the guarantee that no future message it receives will require it to undo its previous computations.

Processes that communicate through shared variables, where a shared variable may be read from/written to by an arbitrary number of processes, are often called *tightly-coupled.* In contrast to loosely-coupled systems, designs of tightly-coupled systems typically require deeper analysis. Since speeds of component processes are assumed to be nonzero and finite, but otherwise arbitrary, it is necessary to analyze all possible execution sequences,

*This work was partially supported by ONR Contract N00014-90-J-1640 and by Texas Advanced Research Program grant 003658-065.

however unlikely some of them may be, to guarantee the absence of "race conditions." Special protocols for mutual exclusion are often required for a process to access shared-variables in an exclusive manner. Yet, shared-variables often provide succinct, and even elegant, solutions; for instance, broadcasting a message can often be implemented by storing the message in a variable that can be read by every process.

1.2 Loosely-Coupled Processes

To motivate the discussion, we consider two examples of shared-variable systems. In the first case, we have two processes sharing an integer variable x; one process doubles x and the other process increments x by 1, from time to time; both processes read x and assign it to their local variables. We contend that these two processes are tightly-coupled by x; each process needs to know the exact value of x before it can complete its access—read or write—of x; no process can proceed with its computation with only a partial knowledge of the value of x. Contrast this situation with a system in which two processes share an integer variable y; the first process increments y by 1 from time to time and the second process, from time to time, decrements y by 1 provided y is positive, and then it proceeds with its computation. (The variable y implements a semaphore.) We contend that these two processes are loosely-coupled; the second process can continue with its computation knowing only that y is positive but without knowing the exact value of y. Similarly, the first process need not know the exact value of y; it may merely transmit the message that y should be incremented by 1, to the second process; the latter increments y upon receiving this message. The value of y at the second process is never more than the true value of y and it tends to "catch up" with the true value; hence the system will not be permanently blocked.

1.3 Contributions of This Paper

The notions of tightly-coupled and loosely-coupled processes depends on the way the shared-variables are accessed. We propose a definition of loosely-coupled processes and show that all point-to-point message passing systems are loosely-coupled under our definition. We argue that large-scale shared-variable programming is feasible only if processes are loosely-coupled. First, the action-sequences in different processes are then serializable

(Eswaran et al [1976]). Therefore, it may be imagined that accesses to shared variables are exclusive even though explicit mutual exclusion is not implemented. Second, an important class of progress properties—of the form, if p is *true* now then q is or will become *true*—holds in a loosely-coupled system if it is implemented in a "wait-free manner" by any component process of the system, and the post-condition q is falsified by no other process; this result does not hold for arbitrary—i.e., non-loosely-coupled—systems. Therefore, a possible programming methodology is to design a loosely-coupled system in which each property of the above form is implemented by a single process, with the restriction that no other process falsify the post-condition.

A problem in hardware design, called *cache-coherence* (Lenoski et al [May 1990]), has its roots in shared-variable programming. Suppose that several processes hold copies of a shared-variable in their local caches. If processes write into their local copies autonomously then these copies may become inconsistent (and then the processes may read different values for the shared-variable). The traditional solutions to cache-coherence restrict accesses to the caches at runtime so that the inconsistencies are avoided; one typical solution is to lock all caches prior to a write by any process, and then broadcast the new value to all processes after completion of the write.

The cache-coherence problem vanishes for loosely-coupled processes. Each process that accesses a shared variable, x, initially keeps a copy of x in its local cache. Reads and writes are performed on the local copies. Whenever a local copy of x is changed the new value is transmitted, asynchronously, to all processes that hold a copy of x. Each process, upon receiving notification of a change, updates its local copy. We prove that this implementation scheme is correct.

The suggested implementation allows the processes to compute asynchronously, using only a partial knowledge of the values of the shared variables. Unlike traditional multiple-copy systems the copies of a shared variable will, typically, have different values; furthermore, no permission is ever sought by a process from other processes nor are the caches locked before a local copy is changed. Therefore, the performance of the system will not degrade if a shared-variable is accessed by a large number of processes; see Lenoski et al [February 1990] for some of the issues in scaling up shared-memory multiprocessors. Since the implementation by local copies is transparent to the programmer, programming

loosely-coupled systems will retain much of the succinctness of expression while admitting efficient implementations.

1.4 Related Work

It is easy to see that read-only shared variables cause no race conditions; however, processes cannot communicate information about the progress of their computations through such variables. Several concurrent logic programming languages (Shapiro [1989]) support *logic variables* or write-once variables. A logic variable is initially undefined and it may be assigned a value at most once and by a single designated process, in any computation. Logic variables have been used in the place of message-communicating primitives (Arvind et al [1989], Vishnubhotla [1989], Chandy and Taylor [1990]). In particular, the programming notation PCN by Chandy and Taylor [1990] makes a distinction between ordinary program variables—called *mutables*—and logic variables—called *permanents*; concurrently executing processes (in one form of process composition) are prevented from changing the shared mutables, thus communicating through the permanents only. It can be shown that these processes are then loosely-coupled. A slightly general form of shared variables has been employed in Misra [1990]. A shared variable assumes an additional value, $\perp$, analogous to the undefined value. A value can be assigned to a shared variable, x, only if its current value is $\perp$; a value can be read from x only if its current value differs from $\perp$; the reading causes x to have the value $\perp$. The variable x may be viewed as a buffer of size 1 for which the writer,reader act as producer,consumer. Unlike logic variables, such shared variables may be assigned value more than once. Processes employing such shared variables where each variable is shared between two processes only can be shown to be loosely-coupled.

Gaifman, Maher and Shapiro [1990] have observed that by restricting the updates of the common store to be monotonic, nondeterministic computations can be replayed efficiently, for the purposes of debugging and recovery. They also propose an easy solution to the snapshot problem for such programs.

Steele [1990] has argued that asynchrony can be made safe by requiring that the "causally-unrelated" actions of various processes commute. He argues that such restricted systems where also each process is deterministic and the computation is terminating,

have some of the properties of SIMD systems (in which processes operate in lock-step) including determinacy. He shows how the violation of commutativity can be detected at run-time. In Chandy and Misra [1988] we had proposed a set of access conditions for shared variables that allows the shared variables to be implemented by messages; the present work simplifies and generalizes those conditions.

2 A Theory of Loosely-Coupled Processes

2.1 Processes

We consider a system consisting of a finite set of processes. Each process has a local store which only that process can access, and there is a global store which all processes can access. (The control point of a process, i.e., the point in the process-text where control resides at some point in the computation, is a part of its local store). Let A be the set of possible values that can be held in the global store, and L_i, the set of possible values for the local store of some specific process, P_i. The set of system states is the cartesian product $A \times_i L_i$, ranging over all process indices i.

The states of the stores are modified by the actions of the processes. An action in process P_i is given by a function, f,

$$f \ : \ A \times L_i \ \rightarrow \ A \times L_i$$

denoting that P_i may read only from the global store and its own local store, and may modify only those stores. A function may be partial; $dom.f$ is the subset of $A \times L_i$ where f is defined.

The effect of applying an action in a given system state is to evaluate the corresponding function, if it is defined, in that state and overwrite the appropriate part of the system state by the function value; if the function is not defined in a system state the action has no effect.

Each process consists of a set of actions. In each step, an action is chosen from some process and applied in the current state; both choices—the process and the action—are nondeterministic. The nondeterministic choice is constrained by the fairness rule that every action from every process is chosen eventually.

This process-model is from UNITY (Chandy and Misra [1988]). At first sight, it may seem restrictive because the traditional control structures—if-then-else, do-while, sequencing, etc.—have been eliminated. Yet, this model has all the generality of the traditional models (except process creation and deletion), and it provides a simpler basis for reasoning and constructing theories. To see how traditional control structures are encoded within this model, consider two actions, f and g, that are to be applied in sequence. Let the program-counter (pc) be 0 when f is to be applied and 1 when g is to be applied. In our model, pc is a part of the local store of the process. We define a function f' at all system states where pc is 0 *and* f is defined; the effect of applying f' is the same as applying f except that the former results in additionally setting pc to 1. Similarly, we define g' (for all states where g is defined and pc is 1). This scheme implements the desired sequencing implicitly.

The reason we have chosen to work with this process-model is that the desired theory can be developed simply, by ignoring a number of issues having to do with program structures.

Convention: If $f.y = y$, for some y, then the effect of applying the corresponding action in this state is a "skip"; the effect is the same as if $y \notin dom.f$. Therefore, we may remove y from $dom.f$ without changing the program semantics. Henceforth, we assume that $f.y \neq y$ for all y, i.e., all our functions are irreflexive.

Convention: A function f,

$$f \ : \ A \times L_i \ \rightarrow \ A \times L_i$$

has the same effect as a function f', that is defined on system states:

$$f' \ : \ A \times_i L_i \ \rightarrow \ A \times_i L_i$$

where f' simply does not "use" or "update" any L_j, $j \neq i$. Therefore, we view each function as a function from system states to system states.

2.2 Definition of Loosely-Coupling

A set of processes is *loosely-coupled* if for every pair of functions, f, g, whenever $f.x$ and $g.x$ are both defined, then so are $f.g.x$ and $g.f.x$, and they are equal in value. □

Observation: If the component processes are sequential, any two functions within a single process commute because there is no system state where both functions are defined (because the functions are defined at system states where the program counter, for this process, has different values). Now, consider two functions, f, g, from distinct processes. If variables accessed by *both* f, g are read-only variables, i.e., no variable written by an action is read or written by another action, then the functions commute. Therefore, the test of commutativity need be applied only to functions where one accesses (i.e., reads or writes) a global variable that the other writes into. □

2.3 Examples of Loosely-Coupled Processes

We give several examples of processes that are loosely-coupled. We adopt the UNITY syntax for writing the process-codes (though the results are entirely independent of the syntax).

counting

An integer variable, c, is shared among processes $P_0 \ldots P_N$. Process P_i, $1 \leq i \leq N$, accesses c in a statement,

$$\{P_i\} \quad c := c + d_i \qquad \text{if} \quad b_i$$

where b_i, d_i are local to P_i and $d_i > 0$. Process P_0 references c in

$$\{P_0\} \quad l := true \qquad \text{if} \quad c > 0$$

where l is a local variable of P_0.

The functions from P_i, P_j, $i \neq j$, $1 \leq i \leq N$, $1 \leq j \leq N$, clearly commute (because b_i, d_i are local to P_i and addition is commutative). The functions from P_0 and P_i, $1 \leq i \leq N$, commute because, since $d_i > 0$,

$$b_i \wedge c > 0 \;\Rightarrow\; c + d_i > 0 \quad \{P_0 \text{ can be applied after applying } P_i\} \wedge$$
$$b_i \qquad\qquad\quad \{P_i \text{ can be applied after applying } P_0\}$$

Also, $\quad (c', l') = (c + d_i,\ true)$

where c', l' are the values of c, l after applying both operations in either order when both are defined. □

parallel search

N processes, $N > 0$, carry out a search in parallel. Shared variables are P, r, where P is a set of process indices, 1 through N (P holds the indices of all processes that have yet to complete the search) and r is the index of the lowest numbered process that has succeeded in the search; if no process has yet succeeded in the search then $r > N$. Initially, $P = \{i | 1 \leq i \leq N\}$ and $r > N$. The code of process i, $1 \leq i \leq N$, is as follows:

$$\begin{array}{lll} \{f_i\} & P, r := P - \{i\}, r \min i & \text{if} \quad b_i \wedge i \in P \\ [\!] \ \{g_i\} & P \quad := P - \{i\} & \text{if} \quad c_i \wedge i \in P \end{array}$$

Here b_i, c_i are local to process i that denote successful or unsuccessful completion of the search. Process 0 accesses the shared variables, using

$$\{h\} \quad d := r \qquad \text{if} \quad P = \phi$$

where d is a local variable of this process.

To see that these processes are loosely-coupled:

- f_i, g_i commute because b_i and c_i cannot hold simultaneously, i.e., the search cannot be both successful and unsuccessful.
- f_i, f_j commute because removing two items from a given set in either order results in the same set and *min* is commutative and associative.
- f_i, g_j commute for similar reasons.
- g_i, g_j commute for similar reasons.
- h, f_i (or h, g_i) are not both defined in any state because $P = \phi \wedge i \in P$ never holds. Therefore, they commute trivially.

point-to-point communication

Two processes, a *sender* and a *receiver*, communicate using a FIFO channel, c. The sender sends messages; the messages are eventually delivered to the receiver in the order sent. It is not surprising that this pair of processes is loosely-coupled; we verify this, formally, below.

We regard the channel, c, as a variable—of type message sequence—shared between the sender and the receiver. The action at the sender is of the form:

$$\{send\}\ c\ :=\ c; m \qquad \text{if} \quad bs$$

Here ";" denotes concatenation and m, bs are local to the sender denoting, respectively, the next message being sent and the condition for sending m. The action at the receiver is of the form:

$$\{receive\}\ l, c\ :=\ \text{head}.c, \text{tail}.c \qquad \text{if} \quad br\ \wedge\ c \neq \langle\rangle$$

Here l, br are local to the receiver (the received message is copied into l; the condition for receiving a message is given by br) and "$\langle\rangle$" denotes the empty sequence.

The two corresponding functions commute in that they produce the same values for l, c when applied in either order.

$$\begin{array}{lll} c \neq \langle\rangle \ \Rightarrow & & \{\text{receive is defined only when } c \neq \langle\rangle\} \\ & (\text{head}(c; m),\ \text{tail}(c; m)) & \{\text{first send then receive}\} \\ = & (\text{head}(c),\ \text{tail}(c); m)) & \{\text{first receive then send}\} \end{array}$$

We leave it to the reader to show that if c is not FIFO, i.e., c is a bag into which the sender adds items and from which the receiver removes items, the processes are still loosely-coupled. □

multi-input channel

This is a variation of the previous example. Two senders use a single channel to send messages to a receiver.

The send actions are

$$\{S1\} \quad c := c; m1 \quad \text{if} \quad bs1$$
$$\text{and } \{S2\} \quad c := c; m2 \quad \text{if} \quad bs2$$

where $m1$, $bs1$ are local to Sender 1 and $m2$, $bs2$ to Sender 2. As before, the receiver action is

$$\{\text{receive}\}\ l, c := \text{head}.c, \text{tail}.c \quad \text{if} \quad br \wedge c \neq \langle\rangle$$

S1, S2 do not commute because, under $bs1 \wedge bs2$

$$(c; m1); m2 \neq (c; m2); m1$$

This reflects the possibility that relative speeds of the processes and the communication network could affect the outcome of the computation. If the multi-output channel is of type bag—the senders add items to the bag and the receiver removes some item from the bag—then the processes are loosely-coupled, because bag union is commutative.

Note: The reader can show that one or more senders sharing a bag with *multiple* receivers are *not* loosely-coupled; then receive actions in two different receivers do *not* commute.

□

broadcast

The value of a shared variable x is to be broadcast to processes $P_1 \ldots P_N$. Process P_0 stores a new value into x provided the previous value has been read by all processes. A process P_i, $1 \leq i \leq N$, records the value of x, in a local variable x_i, provided it is a new value. In order to determine if a value is new and also if the processes have read the previous value, we introduce boolean variables $b_0 \ldots b_N$, and the invariant: For any i, $1 \leq i \leq N$, P_i has read the current value of x iff $b_i = b_0$. Thus, if b_0 equals every other b_i then every P_i has read the value of x; a new value may then be stored in x and b_0 changed (so that it differs from every other b_i). Also, P_i reads x only if b_i differs from b_0; reading is accompanied by setting b_i to b_0. Specifically, the write action in P_0 is

$$\{P_0\} \quad x, b_0 := l, \neg b_0 \quad \text{if} \quad (\forall i : 1 \leq i \leq N \ :: \ b_0 = b_i) \wedge cw$$

where l, cw are local to P_0; variable l holds the next value to be broadcast and cw is the condition for broadcasting. The read action in P_i is

$$\{P_i\} \quad x_i, b_i \ := \ x, b_0 \qquad \text{if} \quad b_i \neq b_0 \ \wedge \ cr_i$$

where x_i, cr_i are local to P_i; the value read is stored in x_i and cr_i is the condition for reading.

The functions in P_0, P_i, $1 \leq i \leq N$, commute because both the corresponding actions cannot be performed in any state:

$$b_i \neq b_0 \ \wedge \ (\forall i \ : \ 1 \leq i \leq N \ \ :: \ \ b_0 = b_i)$$

is *false*.

The operations in P_i, P_j, $1 \leq i \leq N$, $1 \leq j \leq N$, commute because common variables accessed by these two actions are x and b_0, and both are only read by these two actions. Hence the processes are loosely-coupled.

As a consequence of loose-coupling, we can show that simultaneous access to all b_i's, as required in P_0's action, may be replaced by asynchronous accesses. □

2.4 Serializability

We show that a sequence of accesses to shared variables within a process (of a loosely-coupled system) may be regarded as non-preemptible or atomic. Thus, within a process it may be assumed that the process has exclusive access to all shared resources. Such exclusive accesses are typically implemented by explicit mutual exclusion algorithms. Hence, explicit mutual exclusion for exclusive access to shared variables is unnecessary for loosely-coupled systems. In particular, programming constructs such as monitors (Hoare [1974]) that enforce mutual exclusion are unnecessary; a monitor may be viewed as a mechanism to implement loose-coupling. Conversely, the mutual exclusion problem cannot be solved by loosely-coupled processes. It has been observed by many people—in particular, Lamport (in private conversation)—that the mutual exclusion paradigm and the producer-consumer paradigm (i.e., loose-coupling, in our terminology) are two distinct notions; our results seem to justify this observation.

We prove a number of results about loosely-coupled processes. In particular, if finite executions of two processes are defined in a particular system state then any interleaving

of these executions is defined in that state, and all interleavings result in the same state. As an example, let f, g be the functions from one process and h be a function from another process. Suppose $f.g.x$, $h.x$ are defined (where x is a system state). Then so are $f.g.h.x$, $f.h.g.x$ and $h.f.g.x$, and they all have the same value. Also, if $f.h.g.x$ and $h.f.g.x$ are both defined—$f.g.x$, $h.x$ may not be defined—then they have the same value. We show that the system state resulting from an interleaved execution, i.e., the value of $f.h.g.x$, say, can be computed in a lazy manner by applying the next function from either sequence—fg or h—if it is defined in the current state, and repeating this procedure. These theorems are used as the basis of design and implementation of loosely-coupled processes.

2.4.1 Properties of Commuting Functions

Henceforth, we deal with (partial) functions from D to D, for a fixed set D; in the context of loosely-coupled processes D is the set $A \times_i L_i$—the set of system states—and each function corresponds to an action in some process. Functions f, g *commute* iff for every x in D,

$$\begin{array}{ll} & f.x \text{ and } g.x \text{ defined} \\ \Rightarrow & f.g.x \text{ and } g.f.x \text{ defined, and } f.g.x = g.f.x \end{array}$$

Let α be a finite sequence of functions. If α is the empty sequence then $\alpha.x$ is defined for every x in D and $\alpha.x = x$. If α is the sequence $\alpha' f$, then $\alpha.x$ is defined iff $f.x$ is defined and $\alpha'.(f.x)$ is defined, and $\alpha.x = \alpha'.(f.x)$. Two sequences, α, β, of functions *commute* if every function from α commutes with every function from β.

Notational Convention: For expressions e, e' we write

$$e = e'$$

to denote that both e, e' are defined and they are equal. Thus, the commutativity condition for f, g can be written:

$$f.x \text{ and } g.x \text{ defined} \quad \Rightarrow \quad f.g.x = g.f.x$$

Lemma 1: Let g be a function that commutes with every function in sequence α. For any x in D,

$$g.x \text{ and } \alpha.x \text{ defined} \quad \Rightarrow \quad g.\alpha.x = \alpha.g.x$$

Proof: Proof is by induction on the length of α.

α is the empty sequence: Since $g.x$ is defined

$g.\alpha.x = g.x$ and $\alpha.g.x = g.x$

Hence, the result follows.

$\alpha = \alpha' f$:

	$f.x$ defined	, $\alpha.x$ defined
	$g.x$ defined	, given
(1)	$f.g.x = g.f.x$	, f, g commute
	$\alpha'.(f.x)$ defined	, $\alpha.x$ defined and $\alpha = \alpha' f$
	$g.(f.x)$ defined	, from (1)
	$\alpha'.g.(f.x) = g.\alpha'.(f.x)$	, using inductive hypothesis on the above two
	$\alpha'.f.g.x = g.\alpha'.f.x$	, replacing $g.f.x$ in the lhs of the above by $f.g.x$, using (1)
	$\alpha.g.x = g.\alpha.x$	, using $\alpha = \alpha' f$

□

Theorem 1: Suppose α, β commute and γ is an interleaving of α, β. For any x in D,

$$\alpha.x \text{ and } \beta.x \text{ defined} \quad \Rightarrow \quad \gamma.x \text{ defined}$$

Proof: Proof is by induction on the length of γ (i.e., on the combined lengths of α and β).

$|\gamma| = 0$: $\gamma.x$ is defined (and $\gamma.x = x$).

$|\gamma| > 0$: If $|\beta| = 0$ then $\gamma = \alpha$ and hence $\gamma.x$ is defined because $\alpha.x$ is defined. Therefore, let β be nonempty, say $\beta = \beta' g$. Since γ is an interleaving of α, β

$$\gamma = A\ g\ B$$

where A, B are sequences of functions and B is a suffix—possibly empty—of α.

Furthermore, the sequence AB is an interleaving of α, β'. For any x in D,

	$g.x$ defined	, $\beta.x$ defined and $\beta = \beta' g$
	$\alpha.x$ defined	, given
	$\alpha.g.x$ defined	, using Lemma 1 on the above two
	$\beta'.g.x$ defined	, $\beta.x = \beta'.g.x$
(1)	$(AB).(g.x)$ defined	, using inductive hypothesis, any interleaving of α, β'—in particular AB—is defined at $g.x$
	$B.x$ defined	, B is a suffix of α and $\alpha.x$ is defined
	$g.x$ defined	, $\beta.x$ defined and $\beta = \beta' g$
	$B.g.x = g.B.x$	, using Lemma 1 on the above two
	$A.B.g.x = A.g.B.x$	, from (1) and the above
	$A.B.g.x = \gamma.x$	, replacing γ by $A\ g\ B$ in the rhs of the above

Therefore, $\gamma.x$ is defined. □

It should be understood that if one of α, β is undefined at x, their interleaving γ may or may not be defined at x. For instance, consider the functions f, g over natural numbers.

$$f.x \equiv x + 1$$
$$g.x \equiv x - 1 \quad \text{if} \quad x > 0$$

f, g commute (note that both are defined at all positive numbers). However, $f.g.0$ is undefined whereas $g.f.0$ is defined. We show, below, that whenever two interleavings of α, β are defined at x they have the same value.

(Also, it is interesting to note that $\gamma.x$ may be defined even though neither $\alpha.x$ nor $\beta.x$ is defined. To see this consider the functions f, g given above and let

$$f'.x \equiv x - 2 \quad \text{if} \quad x \geq 2$$
$$g'.x \equiv x + 2$$

Let $\alpha = f'f$, $\beta = g'g$, $\gamma = f'g'gf$. It is easy to see that α, β commute, and that γ is an interleaving of α, β. Neither $\alpha.0$ nor $\beta.0$ is defined, though $\gamma.0$ is defined.)

Theorem 2: Suppose α, β commute. Let γ, δ be interleavings of α, β. For any x in D, $\gamma.x$ and $\delta.x$ defined $\Rightarrow$ $\gamma.x = \delta.x$.

Proof: Proof is by induction on $|\gamma|$ (note that $|\gamma| = |\delta|$).

$|\gamma| = 0$: Then, both γ, δ are empty sequences and the result holds, trivially.

$|\gamma| > 0$: Let $\gamma = \gamma' f$ and $\delta = \delta' g$.

case 1) $f = g$: Then,

$\gamma.x = \gamma'.(f.x)$ and $\delta.x = \delta'.(f.x)$.

Since γ', δ' are both defined at $f.x$ and they are both interleavings of some subsequences of α, β, applying induction hypothesis,

$\gamma'.(f.x) = \delta'.(f.x)$

i.e., $\gamma.x = \delta.x$

case 2) $f \neq g$: Suppose f is from α. Then g is from β, i.e., $\beta = \beta' g$. We can write $\gamma = A\ g\ B$ where B is a suffix of α. Note that AB and δ' are both interleavings of α, β'.

	$B.x$ defined	, $\gamma.x$ defined and $\gamma = A\ g\ B$
	$g.x$ defined	, $\delta.x$ defined and $\delta = \delta' g$
(1)	$g.B.x = B.g.x$	, using Lemma 1 (since g is from β and all functions in B are from α, g commutes with B)
	$\gamma.x = A.g.B.x$	, $\gamma = A\ g\ B$
	$\gamma.x = A.B.g.x$	, replacing $g.B.x$ by $B.g.x$ using (1)
	$(AB).(g.x) = \delta'.(g.x)$	, from the above, $(AB).(g.x)$ is defined. Also, $\delta'.(g.x)$ is defined, because it is $\delta.x$. Both AB and δ' are interleavings of α, β'. From the induction hypothesis.
	$\gamma.x = \delta.x$	, from the above two

□

Theorem 3: Let α, β commute. Let γ be an interleaving of α, β. Suppose $\beta = \beta' g$, for some β' and g.

$g.x$ and $\gamma.x$ defined $\Rightarrow$ $\gamma.x = \gamma'.g.x$

for some interleaving γ' of α, β'.

Proof: Let $\gamma = A\ g\ B$. Here, B is a suffix of α because g is the least element of β.

$g.x$ defined	, given
$B.x$ defined	, $\gamma.x$ defined and $\gamma = A\ g\ B$.
$g.B.x = B.g.x$	, g, B commute. Use Lemma 1.
$A.g.B.x = A.B.g.x$	, from the above and $\gamma.x = A.g.B.x$
$\gamma.x = (AB).g.x$	, from the above

This completes the proof with $\gamma' = AB$. □

Corollary 1: Let α, β commute. Let γ be an interleaving of α, β.

$$\beta.x \text{ and } \gamma.x \text{ defined} \quad \Rightarrow \quad \gamma.x = \alpha.\beta.x$$

Proof: Repeated use of Theorem 3 substituting each proper prefix of β, from the longest to the shortest, for β' in the above theorem. □

The next theorem says that given γ, an interleaving of α, β that commute, if $\beta.x$ is defined and $\gamma.x$ is undefined then γ is undefined because of a function from α, i.e., γ has a suffix $f\gamma'$ where $\gamma'.x$ is defined, $f.\gamma'.x$ is undefined and f is from α. The theorem below states the contrapositive of this result.

Theorem 4: Let α and $g\beta$ commute. Let γ be an interleaving of α, β.

$$g.\beta.x \text{ and } \gamma.x \text{ defined} \quad \Rightarrow \quad g.\gamma.x \text{ defined}$$

Proof: Since $\beta.x$ is defined (from $g.\beta.x$ is defined) and $\gamma.x$ is defined, we have $\gamma.x = \alpha.\beta.x$, from Theorem 3. Now,

$g.\beta.x$	defined	, from the antecedent
$\alpha.\beta.x$	defined	, from the above argument
$g.\alpha.\beta.x$	defined	, Lemma 1 applied to the above two
$g.\gamma.x$	defined	, $\gamma.x = \alpha.\beta.x$

□

Note: Our theorems readily generalize to interleavings of several sequences where each pair of sequences commute. In fact, we will be normally dealing with this more general form in all cases. □

2.5 Compositional Designs of Loosely-Coupled Systems

We show how properties of a system of loosely-coupled processes may be deduced from the properties of the component processes. These deduction rules can be used as a basis for system design.

We compose processes using the *union* operator of UNITY; for processes F, G their union, written as $F \parallel G$, is a process in which actions from F, G are executed concurrently and asynchronously. In each step of the execution of $F \parallel G$ an action from either F or G is chosen and executed; the choice is to be made *fairly*, in the sense that every action is chosen eventually for execution; see Chandy and Misra [1988] for details.

The two classes of program properties—*safety* and *progress*—are expressed using the operators, *unless* and *leads-to*. (UNITY employs another operator, *ensures*, as the basis for the inductive definition of *leads-to*; we won't be needing that operator for the theory developed in this paper.)

The basic safety property is of the form, p *unless* q, where p, q are predicates (defined on the state space of the program). The operational interpretation of p *unless* q is that once p is *true* it remains *true* as long as q is *false*. An important special case is p *unless false* which means that p remains *true* once it becomes *true*; we write p *stable* for p *unless false*. A predicate p is invariant in a program if p is initially *true* and p is stable.

The basic progress property is of the form $p \mapsto q$ (read p *leads-to* q); its operational interpretation is: Once p is *true*, q is or will become *true*. See Chandy and Misra [1988] for formal definitions of these operators.

union theorem

Safety properties of composite programs can be deduced from the safety properties of their component processes, as given below. This result applies to all processes, not just loosely-coupled processes.

Theorem 5 (for safety): (See Chandy and Misra [1988, Sec. 7.2.1])

$$p \text{ unless } q \text{ in } F \parallel G = p \text{ unless } q \text{ in } F \wedge p \text{ unless } q \text{ in } G \qquad \square$$

This theorem provides a basis for designing a composite system: A system satisfying *p unless q* can be designed as a union of two components in each of which *p unless q* holds.

There is no corresponding result for progress properties. Indeed, a progress property established by one component process may be affected by the operations of another process, as shown in the following example.

Example: Processes F, G share a variable x that can take three possible values—0, 1 or 2. Each process has a single action.

$$F :: x := (x+1) \bmod 3$$
$$G :: x := (x+2) \bmod 3$$

The initial value of x is irrelevant. It is easy to see that

$$x = 0 \mapsto x = 2 \quad \text{in } F$$
$$x = 0 \mapsto x = 2 \quad \text{in } G$$

However,

$$x = 0 \mapsto x = 2 \quad \text{in } F \,[\!]\, G$$

does not hold: To see this consider the repeated execution of F followed by G, starting in a state where $x = 0$. Note that F, G are loosely-coupled according to our definition. □

(A part of the union theorem can be used to deduce a special class of progress properties, of the form *p ensures q*; the theorem says that *p ensures q* holds in a composite program iff *p ensures q* holds in at least one component and *p unless q* holds in all other components. No such result holds for $p \mapsto q$.)

The lack of a theorem for progress, analogous to the union theorem for safety, is a serious drawback for concurrent program design. In order to design a system in which $p \mapsto q$ holds, we cannot easily partition the design into several processes each satisfying some safety and progress properties. Conversely, to assert a progress property for a given system, all its components have to be considered together. A major simplification,

proposed by Owicki and Gries [1976], is to ascertain that the proof constructed for a single process is not affected by the executions of the statements in the other processes; construction of such a proof is still an expensive procedure.

We show a theorem below which can serve as the basis for designing loosely-coupled systems. The essence of the theorem is: If process F establishes $p \mapsto q$ in a "wait-free" manner (defined below), and F is a component in a system of loosely-coupled processes where every other process preserves q, then $p \mapsto q$ holds in the entire system. Thus, a progress property, $p \mapsto q$, of a loosely-coupled system can be implemented by designing a single process to implement this property in a wait-free manner, requiring other processes to preserve q (i.e., having "q *stable*").

Process F is *wait-free* for (p, q) if once $p \wedge \neg q$ holds, every action of F is *effective* (i.e., execution of the action changes state) at least until q holds. This property can be established as follows. Let c_i be the condition under which the i^{th} action in F executes effectively. Then,

$$p \wedge \neg q \;\Rightarrow\; (\wedge\, i \;::\; c_i) \quad \{\text{all actions can be executed effectively under } p \wedge \neg q\}$$
$$(\wedge\, i \;::\; c_i)\ \textit{unless}\ q \quad \{\text{all actions can be executed effectively at least until } q \text{ holds}\}$$

The notion of wait-freedom captures our intuitive understanding that once p holds in F, process F can execute autonomously without waiting for any external signal.

Theorem 6: Let F, G be loosely-coupled.

$$\frac{\begin{array}{c} p \mapsto q \ \text{ in } F\,, \\ F \text{ is wait-free for } (p, q)\,, \\ q \textit{ stable in } G \end{array}}{p \mapsto q \ \text{ in } F \,[\!]\, G}$$

Proof (sketch): Consider an infinite execution, σ, of $F \,[\!]\, G$ starting in a state where p holds; call this starting state x. We show that σ has a finite prefix, τ, at the end of which q holds.

Let σ_F be the sequence obtained by retaining only and all the F-actions of σ. Since σ is a (fair) execution of $F \,[\!]\, G$, σ_F is a (fair) execution of F. From $p \mapsto q$ in F, we

know that every (fair) execution of F starting in a state where p holds has a prefix at the end of which q holds. Therefore, there is a prefix α of σ_F after which q holds. Since F is wait-free for (p, q), $\alpha.x$ defined.

Let τ be a prefix of σ such that $\tau_F = \alpha$. We deduce the system state by applying τ to state x, as follows. First, remove all actions from τ that are undefined in the corresponding state (those that behave as a *skip*); let the new sequence be τ'. Clearly, the system states by applying τ and τ' to x are identical; furthermore, $\tau'.x$ is defined. We show that q holds for $\tau'.x$.

Since τ is an interleaving of α and some sequence of G-actions, and $\alpha.x$ is defined, according to Theorem 4, the removed actions from τ are G-actions only. Therefore, τ' is an interleaving of α and a sequence, say β, of G-actions. Since F, G are loosely-coupled, α, β commute. We know that $\tau'.x$ and $\alpha.x$ are defined. From Theorem 3,

$$\tau'.x = \beta.\alpha.x$$

We know that q holds for $\alpha.x$ and q is not falsified by any action from β (because q is stable in G and β consists of G-actions). Hence, q holds for $\tau'.x$. □

A Programming Methodology for Loosely-Coupled Processes

Theorem 5 provides a basis for programming loosely-coupled processes. We implement a progress property $p \mapsto q$ by a single process in a wait-free manner; then we require that the other processes not falsify q. The methodology for constructing a program from its specification, consisting of safety and progress properties, is as follows:

Require that

- the component processes be loosely-coupled,
- each safety property hold for each component process, and
- each progress property, of the form $p \mapsto q$, hold in a specific process (in a wait-free manner) and q be stable in the other processes.

This design methodology is driven by consideration of the progress properties; the safety properties merely serve as the restrictions in designing the individual processes. Note that we may decompose a progress property $p \mapsto q$ into, say, $p \mapsto r$ and $r \mapsto q$ and have them be implemented by different processes.

3 Implementing Loosely-Coupled Processes: The Cache -Coherence Problem

We show in this section that the cache-coherence problem vanishes for loosely-coupled processes. An implementation can employ an *asynchronous* communication system to bring the caches into coherence, and it is not required for a process to lock other process caches or ask for permission in updating its own cache. The implication of this observation is that loosely-coupled processes can be implemented very efficiently in a distributed, message-passing type architecture. Consequently, such systems are highly scalable, a property that is not enjoyed by arbitrary shared-variable systems. Even when a system is not loosely-coupled, it pays to identify the processes that are loosely-coupled because their interactions can be implemented asynchronously, as described above. (In fact, with slight assistance from the programmer, compilers can generate code that minimizes cache-locking.)

The implementation of loosely-coupled processes employing a point-to-point communication network is as follows. Initially, each process holds the contents of the global store and the local stores of all processes in a local cache; hence, all caches are initially coherent. A process computes by reading values from its local cache and writing new values into its local cache. Whenever a process changes a value in its local cache it sends an *update* message to every other process informing them of the change. Upon receiving an *update* message a process updates its local cache appropriately (the exact mechanism of update is explained in the sequel).

We show that this implementation is correct. Observe that the caches may never be coherent beyond the initial state. Therefore, we cannot show that for every finite execution of the original system there is a finite execution in the implementation that has the same final state. Instead, we show that whatever could be proven in the original system can be proven in the implementation. More precisely, if A denotes the original system and B the implementation, we prove that

$$p \text{ unless } q \quad \text{in } A \quad \Rightarrow \quad p \text{ unless } q \quad \text{in } B$$

$$p \mapsto q \quad \text{in } A \quad \Rightarrow \quad p \mapsto q \quad \text{in } B$$

Let s denote the entire system state (the content of the global store and the local

stores) during the computation of A. An action in a process i is of the form,

$$s := f.s \qquad \text{if} \quad f.s \text{ is defined}$$

(Note that the action can only modify the contents of the global store and the local store of process i.) In the implementation, B, let s_i denote the local cache contents of process i. Initially, all s_i's are equal to s. The above action is implemented by

$$s_i := f.s_i \qquad \text{if} \quad f.s_i \text{ is defined}$$

and sending an update message "f" to all other processes.

Messages are delivered in FIFO order along a channel directed from one process to another. Process i removes the next message, g, from an incoming channel only if $g.s_i$ is defined; it then executes

$$s_i := g.s_i$$

We assume that if $g.s_i$ remains defined continuously it will be removed. If $g.s_i$ is undefined, the message is left in the channel.

The sequence of messages in the channel from process j to process i (sent by j and unremoved by i) is $ch(j,i)$. Initially, all $ch(j,i)$ are empty.

In order to prove the correctness of the implementation we consider yet another system, C, which is obtained by augmenting B with the auxiliary variable s denoting the "true" system state; the action in process i is replaced by

$$s, s_i := f.s, f.s_i \qquad \text{if} \quad f.s_i \text{ is defined}$$

(We will show that $f.s$ is defined.) Our correctness proof consists of showing that all properties of A hold in C. (Next, since s is auxiliary, it may be removed from C, to obtain B.) We show that at every point in the computation, s may be obtained by applying the update messages in the incoming channels of i to s_i. Specifically, the following is an invariant for C.

invariant For process i there is a sequence, γ, where γ is an interleaving of $ch(j,i)$, for all j, $j \neq i$ and

$$s = \gamma.s_i$$

Proof: Initially, the invariant holds with γ as the empty sequence. To prove that every change to s or s_i preserves the invariant, we observe that these variables may be changed by: applying an action in process i (which may change both s, s_i), applying an action in process j, $j \neq i$ (which may change s and $ch(j,i)$, but does not change s_i) or, process i receiving an update message, g (thus changing s_i but leaving s unchanged).

Action in process i: Executing

$$s, s_i \ := \ f.s, f.s_i \qquad \text{if} \quad f.s_i \text{ is defined}$$

preserves the invariant, trivially, if $f.s_i$ is undefined. Otherwise, prior to the execution

	$f.s_i$ defined	, assume
	$\gamma.s_i$ defined	, $s = \gamma.s_i$ from the invariant
(1)	$f.\gamma.s_i = \gamma.f.s_i$	, Lemma 1 (f commutes with γ)
	$f.s$ defined	, from the above using $s = \gamma.s_i$

Thus assigning $f.s$ to s is legal. Next, observe that

$$\begin{aligned} & f.s \\ = & \{s = \gamma.s_i\} \\ & f.\gamma.s_i \\ = & \{\text{from (1)}\} \\ & \gamma.f.s_i \end{aligned}$$

Hence, the invariant holds with $f.s_i$, $f.s$ in place of s_i, s; also $ch(j,i)$ does not change for any j, $j \neq i$.

Action in process j, $j \neq i$:

$$s, s_j \ := \ g.s, g.s_j \qquad \text{if} \quad g.s_j \text{ is defined}$$

has the effect that s may be changed; also $ch(j,i)$ is extended by appending "g" to it.

$$\begin{aligned} & g.s \\ = & \{s = \gamma.s_i\} \\ & g.\gamma.s_i \end{aligned}$$

Hence, the invariant is satisfied with the sequence $g\gamma$.

Receiving a message: Process i executes

$$s_i := g.s_i \qquad \text{if} \quad g.s_i \text{ is defined}$$

and removes g from a sequence $ch(j,i)$, $j \neq i$. Since $\gamma.s_i$ is defined prior to the execution, $ch(j,i)$ is of the form $\beta' g$ and $g.s_i$ is defined, then using Theorem 3 we have $\gamma.s_i = \gamma'.g.s_i$. Therefore, the above action preserves the invariant, by replacing s_i by $g.s_i$, and γ by γ'. □

Communication Axiom

There is another property of C that captures the essence of the underlying communication. Assume that the message network delivers every message eventually, and that process i eventually removes the head message g of an incoming channel if $g.s_i$ remains defined. Consider some point in the computation when the values of

$$s_i, s \text{ are } S_i, S$$

From the invariant, there is an interleaving, γ, of all incoming-channel contents such that

$$S = \gamma.s_i$$

In order to compute $\gamma.S_i$, we may start by applying *any* function g where g is the head message of a channel and $g.S_i$ is defined (from Theorem 3). It can be shown that our rule of removing messages from channels will eventually result in

$$s_i = \gamma'.S_i$$

where γ is a subsequence of γ'. Applying Theorem 3 (as in Corollary 1) we have,

$$\gamma'.S_i = \delta.\gamma.S_i$$

or, $s_i = \delta.S$

We observe the following fact about δ. For function f in process i,

- either $f.s_i$ remains undefined throughout until s_i becomes $\delta.S$. In that case, function f is never applied to s_i and hence, δ does not contain f.

- or $f.s_i$ becomes defined somewhere during this period.

Combining these observations, we have the communication axiom:

CA :: For each process i,

$$s = S \;\mapsto\; (s_i = \delta.S \;\wedge\; \delta \text{ does not contain } f) \;\vee\; f.s_i \text{ is defined}$$

Using the invariant and the communication axiom we can establish,

Theorem 7: p *unless* q in $A \;\Rightarrow\; p$ *unless* q in C

Theorem 8: $p \mapsto q$ in $A \;\Rightarrow\; p \mapsto q$ in C

4 Conclusion

The notion of loose-coupling promises to simplify programming and implementations of a class of shared-variable systems.

References

- Arvind, Nikil, R.S. and K. K. Pingali [1989]. "I-Structures: Data Structures for Parallel Computing," *ACM TOPLAS*, Vol 11, No. 4, October 1989, 598–632.
- Chandy, K. M., and J. Misra [1988]. *Parallel Program Design: A Foundation*, Reading, Massachusetts: Addison-Wesley, 1988.
- Chandy, K. Mani and Stephen Taylor, "A Primer for Program Composition Notation," *Caltech-CS-TR-90-10*, June 20, 1990.
- Eswaran, K. P., Gray, J. N., Lorie, R. A. and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *C. ACM* 19(11):624–633, November 1976.
- Gaifman, H., Maher, M. J., and E. Shapiro [1990. "Replay, Recovery, Replication and Snapshot of Nondeterministic Concurrent Programs," Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, 76100, Israel, July 1990.
- C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *C. ACM*, Vol. 17, No. 10, 549–557, October 1974.
- Lenoski, D., Gharachorloo, K., Laudon, J., Gupta, A., Hennessy, J., Horowitz, M. and Lam, M., "Design of Scalable Shared-Memory Multiprocessors: The DASH Approach," *Proc. ACM, Compcon*, February, 1990.

- Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A. and Hennessy, J., "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proc. IEEE, 17th Annual International Symposium on Computer Architecture*, 148–159, Seattle, WA, May, 1990.

- Misra, J. [1990]. "Specifying Concurrent Objects as Communicating Processes," *Science of Computer Programming 14* (1990), 159–184.

- Owicki, S., and D. Gries [1976]. "An Axiomatic Proof Technique for Parallel Programs I," *Acta Informatica*, 6:1, 1976, 319–340.

- Shapiro, E. [1989]. "The Family of Concurrent Logic Programming Languages," *ACM Computing Surveys*, 21:3, 412–510, 1989.

- Steele, Guy L. Jr., "Making Asynchronous Parallelism Safe for the World," *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 17–19, 1990, pp. 218-231.

- Vishnubhotla, Prasad [1989]. "Concurrency and Synchronization in the ALPS Programming Language," *TR56*, Ohio State University, 1989.

Rendez-Vous with Metric Semantics

J.W. de Bakker[1]
CWI, Postbus 4079, NL–1009 AB Amsterdam
& Vrije Universiteit

E.P. de Vink
Department of Mathematics and Computer Science, Vrije Universiteit
De Boelelaan 1081a, NL–1081 HV Amsterdam

Abstract

A comparative semantic study is made of an element of the family of concurrent object-oriented programming languages. Particular attention is paid to two notions: (i) dynamically evolving process structures, including a mechanism to name and refer to processes and a means to create new processes, and (ii) rendez-vous between processes involving the sending and answering of messages and the induced execution of method calls. The methodology of metric semantics is applied in the design of operational and denotational semantics, as well as in the proof of their equivalence. Both semantics employ domains which are determined as fixed points of a contracting functor in the category of complete metric spaces. Moreover, fruitful use is made of the technique of defining semantic meaning functions as fixed points of contracting higher-order mappings. Finally, syntactic and semantics continuations play a pervasive role.

1 Introduction

We shall present a comparative semantic study of a language of the COOP (concurrent object-oriented programming) variety. Particular attention will be paid to the following two phenomena

- dynamically evolving process structures, including a mechanism to name and refer to processes and a means to create new processes;
- a version of *rendez-vous* between processes involving the sending and answering of messages and the ensuing execution of method calls.

The language we consider is a slightly simplified version of the language POOL - the parallel object-oriented language designed by America [Ame89]. Several semantic investigations of this language have appeared already: operational semantics ([ABKR86]), denotational semantics ([ABKR89]), and a comparison of these two ([Rut90b]). Cf. also [AR89a] for a somewhat streamlined version of parts of [ABKR86, ABKR89, Rut90b] – excluding the more difficult sections of the comparison –, and [AR90], where an improvement of POOL's denotational semantics which is organized in three *layers* (for statements, objects and programs) is described. The latter paper is intended as well as a contribution to the issue of the *full abstractness* of the POOL semantics.

[1]Partially supported by ESPRIT Basic Research Action 3020: Integration

The treatments in [ABKR89, Rut90b] are rather complex and demand much from the uninitiated reader. The first aim of the present paper is to provide a more comprehensible version of these investigations, with special emphasis on the comparative issues. Partly, this is achieved by a presentation in two stages, both dealing with dynamically evolving processes, but only in the second one with a facility to name and refer to processes. Also, a careful tuning of the design of the operational and denotational definitions – in particular by the systematic use of so-called syntactic and semantic *continuations* – results in a transparent view of the relationship between the two models. Maybe more importantly, we propose a substantial simplification in the way the rendez-vous concept is handled. Firstly, the operational semantics rule for the rendez-vous is now appealingly simple and, secondly, some of the complexities in the denotational models of [ABKR89, Rut90b], in particular in the definition of the merge operator, are to a large extent avoided. Related to this we find that the equation determining the domain used in POOL's denotational semantics is essentially simplified in our approach. (In the domain equation $P = F(P)$, $F(P)$ has no more subterms of the form $(P \to \cdots)$. See Section 2 for background on this.) In addition, the somewhat extraneous use of the denotational meaning function $\mathcal{D}$ as part of the intermediate operational semantics in [Rut90b] is no more necessary.

The second aim of our paper is to provide a case study in the use of *metric* semantics. Let us first devote a few words to its basic principles. Consider two computations p_1, p_2. A natural *distance* $d(p_1, p_2)$ may be defined in terms of the notion of *initial segment* $p(k)$ of p – roughly, that part of p consisting of the first k steps (if present, otherwise p itself). Now we put $d(p_1, p_2) = 2^{-n}$, where n is the length of the longest common initial segment of p_1 and p_2 (i.e., $n = sup\{k \mid p_1(k) = p_2(k)\}$). Details vary with the form of the p_1, p_2. If computations are given as words (finite or infinite sequences of atomic actions), we take the standard notion of prefix; if p_1, p_2 are trees, we use truncation at depth k for $p(k)$. Other kinds of computations, e.g. involving function application, may be accommodated as well.

Complete metric spaces (*cms*'s) have the characteristic property that Cauchy sequences always have limits; this motivates their use for a smooth handling of infinite behaviour. In addition, each *contracting* function $f : (M, d) \to (M, d)$, for (M, d) a cms, has a *unique* fixed point (by Banach's theorem; see Section 2 for the definition of contracting). Uniqueness of fixed points may conveniently be exploited in a variety of situations:

Firstly, it has been shown that cms's may be used to solve *domain equations* of the form

$$P = F(P) \tag{1.1}$$

or, rather, $(P, d) \cong F(P, d)$, with (P, d) the cms to be determined, $\cong$ isometry, and F a mapping (functor) built from given cms's (A, d_A), the unknown (P, d), and composition rules such as $\cup$ (disjoint union), $\times$ (cartesian product), $\to$ (function space), and $\mathcal{P}_{cl}(\cdot)$, $\mathcal{P}_{co}(\cdot)$ (the power sets of all *closed* or *compact* subsets of $\cdot$). See [BZ82], [AR89b] for mathematical details. As an advantage over the more usual cpo framework when used to solve (1.1) we mention that the notions of closed and, especially, compact subset arise very naturally for (the meanings of) many programming constructs. In a cpo setting, one has to choose between the Plotkin-, Smyth-, and Hoare powerdomains (cf. [GS90] for definitions), and it may not be so readily seen how to motivate a choice among these on the basis of a programming (rather than a mathematical) intuition.

Secondly, both denotational ($\mathcal{D}$) and operational ($\mathcal{O}$) semantics may be obtained as fixed points of (contracting) higher-order mappings, say Ψ and Φ. For $\mathcal{D}$ this is fairly traditional: in fact, it subsumes the classical fixed point treatment of recursion. For $\mathcal{O}$ it is less standard: Starting from a transition system $\mathcal{T}$ in the familiar Plotkin SOS style, one may assemble all transitions for a given program π into a meaning $\mathcal{O}(\pi)$. Here the choice as to what kind of domain is used as range for $\mathcal{O}$ (e.g. *linear time*, *branching time* (cf. [BBKM84, BMOZ88]) or bisimulation, interleaving or *noninterleaving* ([BW90]), *failure set* semantics (cf. [Rut89])) is a separate decision, in most cases independent of the design of $\mathcal{T}$. Maybe the most important advantage of this way of defining $\mathcal{O}$ as $fix(\Phi)$ is that is suggests a quite natural method to establish (*) $\mathcal{O} = \mathcal{D}$, viz. by proving that $\mathcal{D} = \Phi(\mathcal{D})$, whence the desired result follows by Banach's theorem (this important proof method is due to [KR90], cf. also [BM88]). Elsewhere ([Rut89, HBR90]) it is discussed how (*) may be strengthened to certain full abstractness results. Recently, investigations have begun concerning the possibility of obtaining $\mathcal{D}$ 'automatically' from a given transition system $\mathcal{T}$. In restricted cases this is indeed possible ([Rut90a]), and it is an interesting problem how this idea may be generalized. We make one further remark on metric vs. cpo semantics: In the latter, one either uses least fixed points, and then has to impose additional conditions to cope with infinite behaviour (e.g. closedness and *boundedness* of [MV88]), or one resorts to greatest fixed points and then continuity may be problematic (see e.g. [Par81]). In the metric framework, once contractivity is satisfied - which is mostly the case - infinite behaviour fits in quite naturally.

Thirdly, unique fixed points may be used to define various semantic operators. In elementary settings, it is no problem to define e.g., sequential or parallel composition. However, if additional features such as infinite behaviour, possibly infinite alphabets, or rendez-vous as part of parallel composition are involved, it is non-trivial how to give rigorous definitions of such operations, and higher-order techniques again turn out to be quite useful.

The present investigation is, partly, a companion to [BV91]. Whereas in that paper we concentrate on so-called *uniform* language notions (the atomic actions are uninterpreted or schematic, and there are no individual variables), we here study a nonuniform (interpreted) language with full-fledged presence of individual variables and non-trivial expressions. The latter necessitate the use, besides of (syntactic and semantic) statement continuations, as well of (syntactic and semantic) expression continuations (in the form proposed in [AB88]). Since in the uniform case certain well-definedness arguments are more perspicuous, we shall occasionally refer below to [BV91] when in need of a justification of some well-definedness property. We refer as well to [BV91] for references to papers where we have used metric semantics for (parallel) *logic* programming (LP). In [Eli91], metric semantics have been described for a language which exhibits, besides the COOP notions studied here, as well LP-like notions such as clausal resolving of goals and backtracking.

We conclude this introduction with a brief overview of the contents of the paper. Section 2 is primarily devoted to a concise presentation of the main ideas concerning the solution of domain equations over (i.e., in the category of) complete metric spaces. In Section 3 we develop comparative semantics for a language ($\mathcal{L}_{pp}$) with 'parallel processes', here to be taken as a dynamically growing system of statements executing in parallel and communicating through (a skeleton version of) the rendez-vous concept. In Section 4 we add to these notions the facility to name and refer to processes, together

with certain refinements of the rendez-vous. The resulting language we call $\mathcal{L}_{po}$, a language with 'parallel objects'. Both for $\mathcal{L}_{pp}$ and $\mathcal{L}_{po}$ we exhibit operational and denotational semantics. We prove that $\mathcal{O} = \mathcal{D}$, for $\mathcal{L}_{pp}$ in some detail and for $\mathcal{L}_{po}$ in outline, in Section 5. Here we find the pay-off from our earlier efforts to obtain a transparent correspondence between the two models, in that the proof of $\mathcal{O} = \mathcal{D}$ is largely syntax-directed, and does not require particular ingenuity.

Acknowledgements We owe much to Pierre America, the designer of the POOL language, and to Jan Rutten who, jointly with Pierre, was responsible for laying its semantics foundations. Jan Rutten also pointed out the need for articulating the notion of resumption in the present paper. We are indebted to Joost Kok for his contributions to the semantic studies of POOL, and, in general, to the members of the Amsterdam Concurrency Group for providing an expert and stimulating forum for discussion on our ongoing research. We thank Franck van Breugel for detailed reading of an earlier version of our paper leading to various improvements.

2 Mathematical preliminaries

2.1 Notations

We use the phrase "let $(x \in)M$ be such that ..." to introduce a set M with variable x ranging over M such that ... . We use $\mathcal{P}_\pi(\cdot)$ for the collection of all subsets of $\cdot$ which have property π. We use $f : X \to Y$ to define a function f with domain X and range (or codomain) Y. If $X = Y$ and $x \in X$ is such that $f(x) = x$, we call x a fixed point of f. If f has a unique fixed point we denote it by $fix(f)$. For $(x \in)M$ any set, we use $\vec{x}$ as a notation for a list (or vector) over M, with $k \geq 1$ elements.

2.2 Domain equations

As mathematical domains for our semantics we use complete metric spaces satisfying a so-called *reflexive domain equation* of the following form:

$$P \cong F(P)$$

(The symbol $\cong$ is defined below; it says that there is a bijection from P to $F(P)$ that respects the metric defined on the spaces.) Here $F(P)$ is an expression built from P and a number of standard constructions on metric spaces (also to be formally introduced shortly). A few examples are

$$P \cong A \cup (B \times P) \tag{2.1}$$

$$P \cong A \cup \mathcal{P}_{co}(B \times P) \tag{2.2}$$

$$P \cong A \cup (B \to P) \tag{2.3}$$

where A and B are given fixed complete metric spaces. In [BZ82] it is first described how to solve these equations in a metric setting Roughly, the approach amounts to the following: In order to solve $P \cong F(P)$ they define a sequence of complete metric spaces $(P_n)_n$ by: $P_0 = A$ and $P_{n+1} = F(P_n)$, for $n>0$, such that $P_0 \subseteq P_1 \subseteq \cdots$. Then they take the *metric completion* of the union of these spaces P_n, say $\bar{P}$, and show: $\bar{P} \cong F(\bar{P})$. In this way they are able to solve equations (2.1), (2.2) and (2.3) above.

There is one type of equation for which this approach does not work, namely,

$$P \cong A \cup (P \xrightarrow{1} G(P)) \tag{2.4}$$

in which P occurs at the *left* side of a function space arrow, and $G(P)$ is an expression possibly containing P. This is due to the fact that it is not always the case that $P_n \subseteq F(P_n)$.

In [AR89b] the above approach is generalized in order to overcome this problem. The family of complete metric spaces is made into a *category* $\mathcal{C}$ by providing some additional structure. (For an extensive introduction to category theory we refer the reader to [Mac71].) Then the expression F is interpreted as a *functor* $F : \mathcal{C} \to \mathcal{C}$ which is (in a sense) *contracting*. It is proved that a generalized version of Banach's theorem (see below) holds, i.e., that contracting functors have a fixed point (up to isometry). Such a fixed point, satisfying $P \cong F(P)$, is a solution of the domain equation.

We shall now give a quick overview of these results, omitting many details and all proofs. For a full treatment we refer the reader to [AR89b]. We start by listing the basic definitions and facts of metric topology that we shall need. We assume the following notions to be known (the reader might consult [Dug66] or [Eng89]): metric space, ultra-metric space, complete (ultra-)metric space, continuous function, closed set, compact set. In our definition the distance between two elements of a metric space is always between 0 and 1, inclusive.

An arbitrary set A can be supplied with a metric d_A, called the *discrete* metric, defined by

$$d_A(x,y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$$

Now (A, d_A) is a metric space (it is even an ultra-metric space).
Let (M_1, d_1) and (M_2, d_2) be two complete metric spaces. A function $f : M_1 \to M_2$ is called *non-expansive* if for all $x, y \in M_1$

$$d_2(f(x), f(y)) \leq d_1(x,y)$$

A function $f : M_1 \to M_2$ is called *contracting* (or a *contraction*) if there exists an ϵ<1 such that for all $x, y \in M_1$

$$d_2(f(x), f(y)) \leq \epsilon \cdot d_1(x,y)$$

(Non-expansive functions and contractions are always continuous.)

The following fact is known as Banach's theorem: Let (M, d) be a complete metric space and $f : M \to M$ a contraction. Then f has a unique fixed point, that is, there exists a unique $x \in M$ such that $f(x) = x$. This x can be obtained by taking the limit of $f^n(x_0)$ for any arbitrary $x_0 \in M$ (where $f^0(y) = y$ and $f^{n+1}(y) = f(f^n(y))$).

We call M_1 and M_2 *isometric* (notation: $M_1 \cong M_2$) if there exists a bijective mapping $f : M_1 \to M_2$ such that for all $x, y \in M_1$

$$d_2(f(x), f(y)) = d_1(x,y)$$

Definition 2.1 Let (M, d), $(M_1, d_1), \ldots, (M_n, d_n)$ be metric spaces.

1. We define a metric d_F on the set $M_1 \to M_2$ of all functions from M_1 to M_2 as follows: For every $f_1, f_2 \in M_1 \to M_2$ we put

$$d_F(f_1, f_2) = \sup_{x \in M_1} \{d_2(f_1(x), f_2(x))\}$$

This supremum always exists since the values taken by our metrics are always between 0 and 1.

2. With $M_1 \cup \cdots \cup M_n$ we denote the *disjoint union* of $M_1, \ldots, M_n$, which can be defined as $\{1\} \times M_1 \cup \cdots \cup \{n\} \times M_n$. We define a metric d_U on $M_1 \cup \cdots \cup M_n$ as follows: For every $x, y \in M_1 \cup \cdots \cup M_n$,

$$d_U(x, y) = \begin{cases} d_j(x, y) & \text{if } x, y \in \{j\} \times M_j, 1 \leq j \leq n \\ 1 & \text{otherwise} \end{cases}$$

If no confusion is possible we often write $\cup$ rather than $\dot{\cup}$.

3. We define a metric d_P on the Cartesian product $M_1 \times \cdots \times M_n$ by the following clause: For every $(x_1, \ldots, x_n), (y_1, \ldots, y_n) \in M_1 \times \cdots \times M_n$,

$$d_P((x_1, \ldots, x_n), (y_1, \ldots, y_n)) = \max_i \{d_i(x_i, y_i)\}$$

4. Let $\mathcal{P}_{cl}(M) = \{X : X \subseteq M \wedge X \text{ is closed}\}$. We define a metric d_H on $\mathcal{P}_{cl}(M)$, called the *Hausdorff distance*, as follows: For every $X, Y \in \mathcal{P}_{cl}(M)$,

$$d_H(X, Y) = \max\{\sup_{x \in X}\{d(x, Y)\}, \sup_{y \in Y}\{d(y, X)\}\}$$

where $d(x, Z) = \inf_{z \in Z}\{d(x, z)\}$ for every $Z \subseteq M$, $x \in M$. (We use the convention that $\sup \emptyset = 0$ and $\inf \emptyset = 1$.) The spaces $\mathcal{P}_{co}(M) = \{X \subseteq M \wedge X \text{ is compact}\}$ and $\mathcal{P}_{nc}(M) = \{X \subseteq M \wedge X \text{ is non-empty and compact}\}$ are supplied with a metric by taking the restriction of d_H.

5. For any real number ϵ with $0 < \epsilon \leq 1$ we define

$$\mathrm{id}_\epsilon((M, d)) = (M, d')$$

where $d'(x, y) = \epsilon \cdot d(x, y)$, for every x and y in M.

Proposition 2.2 Let (M, d), $(M_1, d_1), \ldots, (M_n, d_n)$, d_F, d_U, d_P and d_H be as in Definition 2.1 and suppose that (M, d), $(M_1, d_1), \ldots, (M_n, d_n)$ are complete. Then

$$(M_1 \to M_2, d_F) \quad \text{(a)}$$

$$(M_1 \cup \cdots \cup M_n, d_U) \quad \text{(b)}$$

$$(M_1 \times \cdots \times M_n, d_P) \quad \text{(c)}$$

$$(\mathcal{P}_{cl}(M), d_H), \quad (\mathcal{P}_{co}(M), d_H), \quad (\mathcal{P}_{nc}(M), d_H) \quad \text{(d)}$$

$$\mathrm{id}_\epsilon((M, d)) \quad \text{(e)}$$

are complete metric spaces. If (M,d) and (M_i,d_i) are all ultra-metric spaces, then so are these composed spaces. (Strictly speaking, for the completeness of $M_1 \to M_2$ we do not need the completeness of M_1. The same holds for the ultra-metric property.)

Whenever in the sequel we write $M_1 \to M_2$, $M_1 \cup \cdots \cup M_n$, $M_1 \times \cdots \times M_n$, $\mathcal{P}_{cl}(M)$, $\mathcal{P}_{co}(M)$, $\mathcal{P}_{nc}(M)$, or $\mathrm{id}_\epsilon(M)$, we mean the metric space with the metric defined above.

The proofs of Proposition 2(a), (b), (c), and (c) are straightforward. Part (d) is more complex. It can be proved with the help of the following characterization of the completeness of $(\mathcal{P}_{cl}(M),d_H)$.

Proposition 2.3 Let $(\mathcal{P}_{cl}(M),d_H)$ be as in Definition 1. Let $(X_i)_i$ be a Cauchy sequence in $\mathcal{P}_{cl}(M)$. We have

$$\lim_{i\to\infty} X_i = \{ \lim_{i\to\infty} x_i : x_i \in X_i, (x_i)_i \text{ a Cauchy sequence in } M \}$$

Proofs of Propositions 2.2(d) and 2.3 can be found in, for instance, [Dug66] and [Eng89]. The proofs are also repeated in [BZ82]. The completeness of $\mathcal{P}_{co}(M)$ is proved in [Kur56].

We proceed by introducing a category of complete metric spaces and some basic definitions, after which a categorical fixed point theorem will be formulated.

Definition 2.4 Let $\mathcal{C}$ denote the category that has complete metric spaces for its objects. The arrows ι in $\mathcal{C}$ are defined as follows: Let M_1, M_2 be complete metric spaces. Then $M_1 \to' M_2$ denotes a pair of maps $M_1 \rightleftarrows_j^i M_2$, satisfying the following properties:

1. i is an isometric embedding,
2. j is non-expansive,
3. $j \circ i = \mathrm{id}_{M_1}$.

(We sometimes write $[i,j]$ for ι.) Composition of the arrows is defined in the obvious way.

We can consider M_1 as an approximation to M_2: In a sense, the set M_2 contains more information than M_1, because M_1 can be isometrically embedded into M_2. Elements in M_2 are approximated by elements in M_1. For an element $m_2 \in M_2$ its (best) approximation in M_1 is given by $j(m_2)$. Clause 3 states that M_2 is a consistent extension of M_1.

Definition 2.5 For every arrow $M_1 \to' M_2$ in $\mathcal{C}$ with $\iota = [i,j]$ we define

$$\delta(\iota) = d_{M_2 \to M_1}(i \circ j, \mathrm{id}_{M_2}) \quad (= \sup_{m_2 \in M_2} \{d_{M_2}(i \circ j(m_2), m_2)\})$$

This number can be regarded as a measure of the quality with which M_2 is approximated by M_1: the smaller $\delta(\iota)$, the better M_2 is approximated by M_1.

As a category-theoretic equivalent of a contracting function on a metric space, we have the following notion of a *contracting functor* on $\mathcal{C}$.

Definition 2.6 We call a functor $F : \mathcal{C} \to \mathcal{C}$ contracting whenever the following holds: There exists an ϵ, with $0 \leq \epsilon < 1$, such that, for all $D \to' E \in \mathcal{C}$,

$$\delta(F(\iota)) \leq \epsilon \cdot \delta(\iota)$$

We can now state the analogue of Banach's theorem. (Cf. [Mac71] for the notions of convergence and direct limit:

Theorem 2.7 Let F be a contracting functor $F : \mathcal{C} \to \mathcal{C}$ and let $D_0 \to^{\iota_0} F(D_0) \in \mathcal{C}$. Let the sequence $(D_n, \iota_n)_n$ be defined by $D_{n+1} = F(D_n)$ and $\iota_{n+1} = F(\iota_n)$ for all $n \geq 0$. This sequence is converging, so it has a direct limit $(D, (\gamma_n)_n)$. We have $D \cong F(D)$.

Let us now indicate how this theorem can be used to solve Equations (2.1) to (2.4) above. We define

$$F_1(P) = A \cup \mathrm{id}_{1/2}(B \times P) \tag{2.5}$$
$$F_2(P) = A \cup \mathcal{P}_{co}(B \times \mathrm{id}_{1/2}(P)) \tag{2.6}$$
$$F_3(P) = A \cup (B \to \mathrm{id}_{1/2}(P)) \tag{2.7}$$

If the expression $G(P)$ in Equation (2.4) is, for example, equal to P, then we define F_4 by

$$F_4(P) = A \cup \mathrm{id}_{1/2}(P \xrightarrow{1} P) \tag{2.8}$$

Note that the definitions of these functors specify, for each metric space (P, d_P), the metric on $F(P)$ *implicitly* (see Definition 2.1).

Now it is easily verified that F_1, F_2, F_3, and F_4 are contracting functors on $\mathcal{C}$. Intuitively, this is a consequence of the fact that in the definitions above each occurrence of P is preceded by a factor $\mathrm{id}_{1/2}$. Thus these functors have a fixed point, according to Theorem 2.7, which is a solution for the corresponding equation. (We often omit the factor $\mathrm{id}_{1/2}$ in the reflexive domain equations, assuming that the reader will be able to fill in the details.)

In [AR89b] it is shown that functors like F_1 to F_4 have *unique* fixed points (up to isometry). The results above hold for complete *ultra-metric* spaces too, which can be easily verified.

3 Parallel Processes

3.1 Introduction

We study the language $\mathcal{L}_{pp}$ of 'parallel processes', with particular attention for the programming notions of process creation and rendez-vous. In Section 4, we shall extend $\mathcal{L}_{pp}$ to the language $\mathcal{L}_{po}$ of 'parallel objects', the essence of the extension being the ability to name and refer to processes.

In $\mathcal{L}_{pp}$ we firstly find several conventional and simple programming constructs: assignments, sequential composition, conditionals, and the while statement. Also, a simple block construct introducing initialized (for convenience) local variables is included. Moreover, simple expressions (terms over some signature) appear. Three more advanced notions are furthermore considered:

- Process creation: Assuming that already $n(\geq 0)$ processes are active (i.e. executing in parallel), the effect of the statement **new**(s) will be to create an $n+1$-st process, with body s, to be executed in parallel to the n already active processes. (Note

that no other form of parallel execution, in particular no form of syntactic '||', is present in $\mathcal{L}_{pp}$.)

- Rendez-vous: This appears in the following 'skeleton' version: We introduce so called *methods* m, $\bar{m}$ (with $\bar{\bar{m}} = m$), together with an accompanying declaration d which assigns to each m a statement $d(m) = s$. Synchronized execution of m and $\bar{m}$ in two parallel processes results firstly in the execution of s, and, thereafter, in the resumption (in parallel) of the two remaining statements ('continuations') following m and $\bar{m}$, respectively. (The effect of $m|\bar{m} = s(= d(m))$ should be compared with similar rules $c|\bar{c} = \tau$ (in CCS) or $a|b = c$ (in ACP), the essential difference being that, contrary to s, τ or c are atomic.) In Section 4, we shall dress up this skeleton with some further notions: transmitting parameters, returning a resulting value, and identifying, by the sender, of the receiving component.

- Expressions with side-effects: We introduce here a simple version of side-effects, in order to motivate the mechanism of (syntactic and semantic) expression continuations. Again, a more interesting setting will be provided in Section 4.

3.2 Syntax

Throughout our paper, we use a self-explanatory BNF-like notation for syntactic definitions. We start with the introduction of four basic sets

- $(x \in) IVar$, a countable set of *individual variables*
- $(\alpha, \beta \in) Cons$, a countable set of *constants*
- $(\phi \in) Func$, a countable set of *function symbols* (each with some arity ≥ 1)
- $(m \in) M$, a *finite* set of *method names*. On M, a mapping $\bar{} : M \to M$, satisfying $\bar{\bar{m}} = m$, is given. (Since it is customary to consider only finite systems of declarations, d's domain M is assumed to be finite. Mathematically, there are no obstacles to dealing with infinite M.)

A program $\pi = (d, s)$ in the language $\mathcal{L}_{pp}$ consists of a declaration d in $Decl_{pp}$ and a statement s in $Stat_{pp}$. A declaration is a mapping from M to $Stat_{pp}$. Statements are conventional (see above), or have the form of the process creation **new**(s) or of a method call m. Expressions (in Exp_{pp}) are conventional, or exhibit a side-effect, in the form of $(s; e)$: an expression which first executes the statement s, and then executes e.

Definition 3.1 (syntax for $\mathcal{L}_{pp}$).

a. $s(\in Stat_{pp}) \quad ::= \quad x := e \mid m \mid (s_1; s_2 \mid \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \mid$
$\textbf{while } e \textbf{ do } s \textbf{ od} \mid \textbf{new}(s) \mid \textbf{begin var } x := e; s \textbf{ end}$

b. $e(\in Exp_{pp}) \quad ::= \quad \alpha \mid x \mid \phi(\vec{e}) \mid (s; e)$

c. $(d \in) Decl_{pp} = M \to Stat_{pp}$

d. $\pi(\in \mathcal{L}_{pp}) \quad ::= \quad (d, s)$

3.3 Operational semantics

The operational semantics for $\mathcal{L}_{pp}$ is derived from a *transition system* $\mathcal{T}_{pp}$. Transitions are built using so-called *syntactic continuations,* which we use in two varieties:

- $(r \in) SySCo$, the syntactic statement continuations
- $(g \in) SyECo$, the syntactic expression continuations.

The design of these two classes has been motivated partly by our wish to obtain a smooth operational semantics for $\mathcal{L}_{pp}$, partly by the desire to obtain a tractable link with the *semantic continuations* which play a key role in the denotational semantics.

Definition 3.2 (syntactic continuations) Let E be a new symbol, standing for 'termination'.

a. $r(\in SySCo) ::= \mathrm{E} \mid (s : r) \mid (e : g) \mid (r_1, r_2) \mid r\langle\alpha/x\rangle \mid$
$\quad$ **if** β **then** r_1 **else** r_2 **fi** $\mid g(\alpha)$

b. $g(\in SyECo) ::= \lambda\alpha.r$

The continuations $(s : r)$ and $(e : g)$ are of a sequential nature. They should be read as 'execute s and continue with r', or 'evaluate e, pass its value to g, and continue with the result', respectively. Next, (r_1, r_2) denotes (interleaved) parallel execution of r_1 and r_2. The **if** $-$ **then** $-$ **else** $-$ **fi** construct and $g(\alpha)$ should be clear. The construct $r\langle\alpha/x\rangle$ will play a role in elaborating an assignment. Syntactic expression continuations were first used in this way in [AB88].

For the definition of $\mathcal{T}_{pp}$, we need the following basic definitions:

Definition 3.3

a. Let $(\alpha, \beta \in) V = Z \cup \{tt, ff\} \cup \cdots$ be the set of *basic values.* V is assumed to include at least the integers and the truth values tt, ff. Other basic values may be added, if desired. We find it convenient to use the same variables to range over V and over the set of constants C.

b. Let, for ϕ a function symbol with arity k, $\bar{\phi}$ be some element of $V^k \to V$.

c. Let $(\sigma \in) \Sigma = IVar \to V$ denote the set of *states.*

d. Let the auxiliary set $(\tau \in) \mathrm{T}$ be defined as $\mathrm{T} = \Sigma \cup M$.

e. Let $r[\alpha/\beta]$ denote the result of syntactically substituting the constant α for the constant β in r.

f. Let $\sigma[\alpha/x]$ denote the state which satisfies

$$\sigma[\alpha/x](y) = \begin{cases} \alpha & \text{if } x \equiv y \\ \sigma(y) & \text{if } x \not\equiv y \end{cases} .$$

We are now ready for

Definition 3.4 (transitions and transition systems)

a. A *transition* is a five-tuple

$$<r_1, \sigma, d, r_2, \tau> \tag{3.1}$$

in $SySCo \times \Sigma \times Decl_{pp} \times SySCo \times \mathrm{T}$. For (3.1) we usually write

$$<r_1, \sigma> \rightarrow_d <r_2, \tau>$$

b. A *transition system* $\mathcal{T}$ is a finite set of rules of the form

$$\frac{<r_1, \sigma_1> \rightarrow_d <r_1', \tau_1>, \cdots, <r_n, \sigma_n> \rightarrow_d <r_n', \tau_n>}{<r, \sigma> \rightarrow_d <r', \tau>}$$

for some $n \geq 0$. Such a rule should be read as: if we can establish (using $\mathcal{T}$) that the n premises are satisfied, we may infer that the conclusion holds. If $n = 0$, we have an *axiom*, written simply as $<r, \sigma> \rightarrow_d <r', \tau>$.

c. Rules which share the same (list of) premise(s) may be combined into one rule (with more than one consequence).

d. In a transition $<r, \sigma> \rightarrow_d <r', \tau>$ we shall usually suppress mentioning the d. No confusion will arise, since transitions are always to be taken with respect to one fixed d.

e. A rule of the form

$$\frac{<r_1, \sigma> \rightarrow_d <r, \tau>}{<r_2, \sigma> \rightarrow_d <r, \tau>}$$

will be abbreviated to $<r_2, \sigma> \rightarrow_0 <r_1, \sigma>$ or even to $r_2 \rightarrow_0 r_1$. (Read: in order to execute r_2, find out how to execute r_1. The '0' expresses that this requires zero 'steps'.)

f. Each transition system $\mathcal{T}$ determines a relation $\mathcal{R}$ which is defined as the least relation (here subset of $SySCo \times \Sigma \times Decl_{pp} \times SySCo \times \mathrm{T}$) satisfying the given axioms and rules.

Next, we give the definition of the transition system $\mathcal{T}_{pp}$ which will be used to obtain the operational semantics $\mathcal{O}$ for $\mathcal{L}_{pp}$.

Definition 3.5 (transition system $\mathcal{T}_{pp}$ for $\mathcal{L}_{pp}$) The rules in $\mathcal{T}_{pp}$ are organized in groups, for easier structuring. This grouping is not part of the formal system itself.

s-rules

- zero-step

$$\begin{aligned}
(x := e) : r &\rightarrow_0 e : \lambda\alpha.(r<\alpha/x>) \\
(s_1; s_2) : r &\rightarrow_0 s_1 : (s_2 : r) \\
\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} : r &\rightarrow_0 e : \lambda\beta.\textbf{if } \beta \textbf{ then } s_1 : r \textbf{ else } s_2 : r \textbf{ fi} \\
\textbf{new}(s) : r &\rightarrow_0 (s : \mathrm{E}, r) \\
<\textbf{begin var } x := e; s \textbf{ end} : r, \sigma> &\rightarrow_0 <(x := e; s) : r<\sigma(x)/x>, \sigma>
\end{aligned}$$

- axioms

$$\begin{aligned} <m : r, \sigma> &\rightarrow <r, m> \\ <\textbf{while } e \textbf{ do } s \textbf{ od} : r, \sigma> &\rightarrow \\ &<e : \lambda\beta.\textbf{if } \beta \textbf{ then } (s; \textbf{while } e \textbf{ do } s \textbf{ od}) : r \textbf{ else } r \textbf{ fi}, \sigma> \end{aligned}$$

e-rules

- zero-step

$$\begin{aligned} \alpha : \lambda\beta.r &\rightarrow_0 r[\alpha/\beta] \\ \phi(e_1, \cdot\cdot, e_k) : g &\rightarrow_0 e_1 : \lambda\beta_1.(e_2 : \cdots (e_k : \lambda\beta_k.\bar{\phi}(\beta_1, \cdot\cdot, \beta_k) : g) \cdots) \\ (s; e) : g &\rightarrow_0 s : (e : g) \end{aligned}$$

- axioms

$$<x : g, \sigma> \rightarrow <\sigma(x) : g, \sigma>$$

r-rules

- zero-step

$$\begin{aligned} \textbf{if } tt \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi} &\rightarrow_0 r_1 \\ \textbf{if } ff \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi} &\rightarrow_0 r_2 \end{aligned}$$

- axioms

$$<r<\alpha/x>, \sigma> \rightarrow <r, \sigma[\alpha/x]>$$

- rules for parallel execution

(interleaving)

$$\frac{<r, \sigma> \rightarrow <r', \tau>}{\begin{array}{l} <(r, \bar{r}), \sigma> \rightarrow <(r', \bar{r}), \tau> \\ <(\bar{r}, r), \sigma> \rightarrow <(\bar{r}, r'), \tau> \end{array}}$$

(rendez-vous)

$$\frac{<r_1, \sigma> \rightarrow <r', m>, <r_2, \sigma> \rightarrow <r'', \bar{m}>}{<(r_1, r_2), \sigma> \rightarrow <s : (r', r''), \sigma>}, \; d(m) = s$$

Explanation

(assignment): evaluating $x := e$ amounts to first evaluating e, and transmitting the result α to the continuation which will eventually arrange that x is set to α.

(new): the body s is supplied with the termination continuation E, and set in parallel to r (which itself may consist of several continuations in parallel)

(begin .. end): evaluate $x := e; s$, and next reset x to the value ($\sigma(x)$) it had upon block entrance

(m): the method m is stored, available for subsequent use in in the rendez-vous rule

($\phi(\vec{e})$): the arguments $e_1, .., e_k$ are evaluated from left to right, yielding $\beta_1, .., \beta_k$; the interpretation $\bar{\phi}$ of ϕ is then applied to these $\beta_1, .., \beta_k$

($r<\alpha/x>$): this handles the assignment of α to x, resulting in $\sigma[\alpha/x]$

(interleaving): the usual interleaving rule for parallel composition

(rendez-vous): in case r_1 and r_2 can make an m and $\bar{m}$-step, respectively, the rendez-vous succeeds, s is executed, and the execution continues with that of (r', r''). (See also the remark at the end of Section 3 for a possible refinement of the rule.)

We next discuss how to assemble all successive steps prescribed by $\mathcal{T}_{pp}$ for some program (d, s) into one result $\mathcal{O}(d, s)$. Crucial here is the definition of the range P of the mapping $\mathcal{O}: \mathcal{L}_{pp} \to P$. We shall determine P as solution of a domain equation (in the category of complete metric spaces, cf. Section 2), viz.

$$P = \{p_0\} \cup (\Sigma \to \mathcal{P}_{co}((\Sigma \cup M) \times P)) \tag{3.2}$$

Equation (3.2) may be understood as follows: Each element p in P (to be called a *process* as well, but now a mathematical, and not a programming, entity) is either the nil-process p_0, or it is a function in $\Sigma \to \mathcal{P}_{co}(\cdot)$ which, when supplied with a state σ as argument, yields an element X of $\mathcal{P}_{co}(\cdot)$, i.e. a compact subset of $(\Sigma \cup M) \times P$. Thus, the elements of $p(\sigma) = X$ are of the form $<\sigma', p'>$ or $<m, p'>$. The first possibility delivers a next state σ', together with a so-called *resumption* p'. This resumption tells us what to do next: In the operational or denotational setting this will be determined by the syntactic or semantic continuation, respectively. A second possibility for an element X is a pair $<m, p'>$; here m results from a method call, and p' is as before. The rendez-vous rule *resolves* synchronized method calls. However, one-sided method calls which have not synchronized with their partner will leave such a pair $<m, p'>$ as a trace in the result.

The domain P is used in the next definition which introduces (the intermediate) $\mathcal{O}_d$ as fixed point of a contracting higher-order mapping (of meaning functions to meaning functions) Φ_d. To understand the structure of the definition, the reader should look at Lemma 3.7.c. This is the result in the form which is most intuitive, and to justify it we employ the Φ_d-mapping.

Definition 3.6 Let $F \in SySCo \to P$.

a. We define $\Phi_d : (SySCo \to P) \to (SySCo \to P)$ by putting

$$\begin{aligned} \Phi_d(F)(\mathrm{E}) &= p_0 \\ \Phi_d(F)(r) &= \lambda\sigma.\{<\tau, F(r')> \mid <r, \sigma> \to <r', \tau>\}, \quad \text{for } r \neq \mathrm{E} \end{aligned}$$

where $\to$ is determined by $\mathcal{T}_{pp}$.

b. $\mathcal{O}_d = fix(\Phi_d)$, $\mathcal{O}(d, s) = \mathcal{O}_d(s : \mathrm{E})$.

We have

Lemma 3.7

a. $\Phi_d(F)(r) \in P$ for each F, r.

b. Φ_d is contracting in F.

c. $\mathcal{O}_d(\mathrm{E}) = p_0$
$\mathcal{O}_d(r) = \lambda\sigma.\{<\tau, \mathcal{O}_d(r')> \mid <r,\sigma> \rightarrow <r',\tau>\}$, for $r \neq \mathrm{E}$

Proof

a. Follows from the fact that $\mathcal{T}_{pp}$ is *finitely branching*, i.e. for each r , σ, we have $|\{(r',\tau) \mid <r,\sigma> \rightarrow <r',\tau>\}| < \infty$.

b. Clear by the definition of $\Phi_d(F)$, in particular by the $<\tau, ..>$-step in its definition.

c. Immediate by the definitions of Φ_d and $\mathcal{O}_d$. □

Remark The domain P has rather more structure than is usual for an operational semantics. We use the same P for our denotational definitions in the next subsection; the proof that $\mathcal{O} = \mathcal{D}$ (in Section 5) will considerably profit from it. On the other hand, it is not difficult to use the same $\mathcal{T}_{pp}$ to obtain a much simpler (i.e., less structured) operational meaning, say $\mathcal{O}^\star : \mathcal{L}_{pp} \rightarrow P^\star$. Let δ be a new symbol (standing for deadlock), and let $\Sigma_\delta^\infty = \Sigma^* \cup \Sigma^\omega \cup \Sigma^* \cdot \{\delta\}$, i.e., the set of all finite sequences over Σ, possibly postfixed by δ, and all infinite sequences over Σ. We put

$$P^\star = \Sigma \rightarrow (\{\{\epsilon\}\} \cup \mathcal{P}_{nc}(\Sigma_\delta^\infty))$$

and define $\mathcal{O}_d^\star$ to satisfy

$$\mathcal{O}_d^\star(\mathrm{E}) = \lambda\sigma.\{\epsilon\}$$

$$\mathcal{O}_d^\star(r) = \begin{cases} \lambda\sigma.\bigcup\{\sigma'.\mathcal{O}_d^\star(r')(\sigma') \mid <r,\sigma> \rightarrow <r',\sigma'>\} \\ \qquad \text{if the above set } \{\cdot\} \neq \emptyset \\ \{\delta\} \quad \text{otherwise} \end{cases} \qquad \text{for } r \neq \mathrm{E}.$$

$\mathcal{O}_d^\star(r)$ exhibits three essential differences with $\mathcal{O}_d(r)$. Firstly, it has lost the branching structure of the latter. Next, steps $<r,\sigma> \rightarrow <r',m>$ do not contribute to the result (whence the possibillity that the set $\{\cdot\}$ might be empty). Thirdly, the resumptions have disappeared (instead of $<\sigma',p'>$ we now simply employ $p'(\sigma')$). As a consequence, $\mathcal{O}_d^\star$ is not compositional. In particular, no relationship of the form $\mathcal{O}_d^\star(r_1, r_2) = \mathcal{O}_d^\star(r_1) \| \mathcal{O}_d^\star(r_2)$ holds.

3.4 Denotational semantics

We shall define the denotational semantics $\mathcal{D}$ for $\mathcal{L}_{pp}$ in terms of the auxiliary semantic mappings $\mathcal{I}_d$ and $\mathcal{E}_d$:

$$\mathcal{I}_d : Stat_{pp} \rightarrow SeSCo \rightarrow P$$
$$\mathcal{E}_d : Exp_{pp} \rightarrow SeECo \rightarrow P$$

Here $(p \in)P$ is as in Section 3.3, $SeSCo =_{df} P$ is the set of semantic statement continuations, and $SeECo =_{df} (f \in)V \to P$ is the set of semantic expression continuations. The definition of the semantic parallel composition operator '$\|$' will be supplied in Definition 3.9.

Definition 3.8 (denotational semantics for $\mathcal{L}_{pp}$)

a.
$$\begin{aligned}
\mathcal{I}_d(x := e)(p) &= \mathcal{E}_d(e)(\lambda\alpha.\lambda\sigma.\{<\sigma[\alpha/x], p>\}) \\
\mathcal{I}_d(m)(p) &= \lambda\sigma.\{<m, p>\} \\
\mathcal{I}_d(s_1; s_2)(p) &= \mathcal{I}_d(s_1)(\mathcal{I}_d(s_2)(p)) \\
\mathcal{I}_d(\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi})(p) &= \mathcal{E}_d(e)(\lambda\beta.\textbf{if } \beta \textbf{ then } \mathcal{I}_d(s_1)(p) \textbf{ else } \mathcal{I}_d(s_2)(p) \textbf{ fi}) \\
\mathcal{I}_d(\textbf{while } e \textbf{ do } s_1 \textbf{ od})(p) &= \\
&\lambda\sigma.\{<\sigma, \mathcal{E}_d(e)(\lambda\beta.\textbf{if } \beta \textbf{ then } \mathcal{I}_d(s_1)(\mathcal{I}_d(\textbf{while } e \textbf{ do } s_1 \textbf{ od})(p)) \textbf{ else } p \textbf{ fi})\} \\
\mathcal{I}_d(\textbf{new}(s_1))(p) &= \mathcal{I}_d(s_1)(p_0) \,\|\, p \\
\mathcal{I}_d(\textbf{begin var } x := e; s_1 \textbf{ end})(p) & \\
&= \lambda\sigma.\mathcal{I}_d(x := e; s_1)(\lambda\bar{\sigma}.\{<\bar{\sigma}[\sigma(x)/x], p>\})(\sigma)
\end{aligned}$$

b.
$$\begin{aligned}
\mathcal{E}_d(\alpha)(f) &= f(\alpha) \\
\mathcal{E}_d(x)(f) &= \lambda\sigma.\{<\sigma, f(\sigma(x))>\} \\
\mathcal{E}_d(\phi(e_1, .., e_k))(f) & \\
&= \mathcal{E}_d(e_1)(\lambda\beta_1.\cdots\mathcal{E}_d(e_k)(\lambda\beta_k.f(\bar{\phi}(\beta_1, .., \beta_k)))\cdots) \\
\mathcal{E}_d(s; e)(f) &= \mathcal{I}_d(s)(\mathcal{E}_d(e)(f))
\end{aligned}$$

Some explanations may help.

- $s \equiv x := e$: e is evaluated, the result is passed on to the expression continuation $f = \lambda\alpha.\cdots$, and eventually a change of state – setting x to α – is performed, and f then continues (resumes) with p
- $s \equiv m$: the pair $<m, p>$ will play a role in the definition of $\|$.
- $s \equiv$ **while** e **do** s_1 **od**: A (silent) step is performed, leaving σ unchanged, and then e is evaluated and the usual conditional for the while statement is given. Note that $\mathcal{I}_d$(**while**...**od**) returns on the right-hand side. To turn this into a well-defined formula, we should in fact define $\mathcal{I}_d$ as a (unique) fixed point of some higher-order contraction Ψ_d. (Details of a related case can be found in [BV91].)
- $s \equiv$ **new**(s_1): $\mathcal{I}_d(s_1)$ is supplied with the nil-continuation p_0, and executed in parallel with the already present continuation p. Note that $\mathcal{I}_d$ uses $\|$; below, we shall see that $\|$ uses $\mathcal{I}_d$. A comment on this follows later.
- $s \equiv$ **begin**...**end**: this amounts to executing the assignment and then the statement s_1, and after that resetting x to the value $\sigma(x)$ it had upon entrance of the block. Thus, it mimicks the operational rule.
- $e \equiv x$: As in $\mathcal{T}_{pp}$, a silent step is performed, and then the value $\sigma(x)$ is passed on to the expression continuation f.

We proceed with the definition of the parallel composition operator. Let X, Y range over $\mathcal{P}_{co}(\mathrm{T} \times P)$.

Definition 3.9 Let $p_1, p_2 \in P$.

a. $p_1 \parallel p_2 = \lambda\sigma.((p_1(\sigma) \between p_2) \cup (p_2(\sigma) \between p_1) \cup (p_1(\sigma) \mid_\sigma p_2(\sigma))$

b. $X \between p = \{<\tau, p' \parallel p> \mid <\tau, p> \in X\}$

c. $X \mid_\sigma Y = \{<\sigma, \mathcal{I}_d(s)(p' \parallel p'')> \mid <m, p'> \in X, <\bar{m}, p''> \in Y \text{ and } d(m) = s\}$

In executing $p_1 \parallel p_2$ for argument σ, one either makes a simple step from the left- or right operand (this yields interleaved execution), or the two outcomes $X = p_1(\sigma)$ and $Y = p_2(\sigma)$ communicate (in $X \mid_\sigma Y$) by a rendez-vous of the two steps $<m, p'>$ in X and $<\bar{m}, p''>$ in Y. This leads to the evaluation of $\mathcal{I}_d(s)$, for $s = d(m)$, with continuation $p' \parallel p''$. The circularity in this definition, viz. $\parallel$ defined in terms of ($\between$ and $\mid_\sigma$ defined in terms of) $\parallel$ and $\mathcal{I}_d$, and $\mathcal{I}_d$ defined in terms of $\parallel$, may be circumvented by using a simultaneous higher-order mapping (in two arguments), and defining $<\mathcal{I}_d, \parallel>$ as unique fixed point of this mapping. Considerable detail about this approach is supplied in [BV91]; therefore, we omit this here.

Finally, we put

Definition 3.10 $\mathcal{D}(d, s) = \mathcal{I}_d(s)(p_0)$.

In Section 5 we shall prove

First Main Theorem For each π in $\mathcal{L}_{pp}$, $\mathcal{O}(\pi) = \mathcal{D}(\pi)$.

Remark Though the rendez-vous rule (and the corresponding denotational definitions) yield precisely all successful computations, one might argue that it induces too many deadlock possibilities: Consider, e.g., the situation that $d(m) = m'$, and that $r_1 = (m : \mathrm{E}, \bar{m}' : \mathrm{E})$, $r_2 = \bar{m} : \mathrm{E}$. Since $<r_1, \sigma> \rightarrow <(\mathrm{E}, \bar{m}' : \mathrm{E}), m>$ and $<r_2, \sigma> \rightarrow <\mathrm{E}, \bar{m}>$, we may infer that $<(r_1, r_2), \sigma> \rightarrow <m' : (\mathrm{E}, \bar{m}' : \mathrm{E}), \sigma>$. As a consequence, in the result $<m' : (\mathrm{E}, \bar{m}' : \mathrm{E}), \sigma>$, a rendez-vous between m' and $\bar{m}'$ is no longer possible (since m''s partner $\bar{m}'$ is not accessible in a parallel component, but has been 'hidden' to occur *after* m'). Thus, an extra deadlock possibility has arisen which should have been avoided. A way out of this problem is the introduction (taken from [ABKR89]) of a separation between so-called *dependent* and *independent* resumptions. This works as follows: Right-hand sides of transitions are now of the form $<r', \sigma'>$ or $<r', <r'', m>>$. Here r' is the independent resumption which may continue independently of the success of the rendez-vous involving m, and r'' is the dependent resumption which may resume only *after* the rendez-vous for m has taken place. The induced modifications in $\mathcal{T}_{pp}$ are

- $<m : r, \sigma> \rightarrow <\mathrm{E}, <r, m>>$
- (revised rendez-vous rule)

$$\frac{<r_1, \sigma> \rightarrow <r_1', <r_1'', m>> \quad <r_2, \sigma> \rightarrow <r_2', <r_2'', \bar{m}>>}{<(r_1, r_2), \sigma> \rightarrow <(s : (r_1'', r_2''), (r_1', r_2')), \sigma>} \quad , \quad d(m) = s$$

Also, in the interleaving rule we now take $\tau \in \Sigma \cup (SySCo \times M)$. As a consequence, only the independent resumption (r') in $<r', <r'', m>>$ is involved in interleaving steps. Next, in the definition of P we replace the $M \times P$ term by $M \times P \times P$. Finally, we change the definition of $\Phi_d(F)(r)$, for $r \neq \mathrm{E}$, to read

$$\Phi_d(F)(r) = \lambda\sigma.\ \{<\sigma', F(r')> \mid <r, \sigma> \rightarrow <r', \sigma'>\} \ \cup \\ \{<m, F(r''), F(r')> \mid <r, \sigma> \rightarrow <r', <r'', m>>\},$$

with '$\rightarrow$' with respect to the amended $\mathcal{T}_{pp}$.
As to the denotational definitions, we impose the following changes:

- change in P as just given
- change in definition of $\mathcal{I}_d$: $\mathcal{I}_d(m)(p) = \lambda\sigma\{<m, p, p_0>\}$
- change in definition of $\|$:

$$X \,\|\!\!\|\, p = \{<\sigma, p' \,\|\, p> \mid <\sigma, p'> \in X\} \cup \{<m, p'', p' \,\|\, p> \mid <m, p'', p'> \in X\}$$
$$X \mid_\sigma Y = \{<\sigma, \mathcal{I}_d(s)(p_1'' \,\|\, p_2'') \,\|\, p_1' \,\|\, p_2'> \mid <m, p_1'', p_1'> \in X, <\bar{m}, p_2'', p_2'> \in Y, d(m) = s\}$$

We leave to the reader to work out the required modifications in the equivalence proof of Section 5.

4 Parallel objects

4.1 Introduction

The language $\mathcal{L}_{po}$ extends $\mathcal{L}_{pp}$ with a mechanism to *name* and *refer to* processes. Such a named process will from now on be called an *object*. It includes an 'active' part - comparable to the s in the **new**(s) construct of Section 3 - and a declarative part. In the declarative part we find the information on how a method name m is to be supplied with a method body μ, here taken in the form of a parametrized expression $\lambda\vec{x}.e$. Individual variables may now refer not only to values such as integers or truth values (together called V in Section 3), but as well to (the names of) objects. To be precise we replace V by

$$(\alpha, \beta, \gamma \in) Obj = SObj \cup ObjN$$

where $SObj$, the set of *standard objects*, takes over the role of V, and $ObjN$ is the set of *object names*. Objects are created as instances of a *class*: the relevant information about a class c is contained in the declaration $d(c)$. This is a pair $<d(c)_1, d(c)_2>$, where $d(c)_1 \in M \rightarrow Meth$ tells us how each method name m is provided with a method $\mu \in Meth$ as its body (i.e., $d(c)_1(m) = \mu$), and $d(c)_2 \in Stat_{po}$ is the statement (the 'process' of Section 3) execution of which is initiated (in parallel to the already existing objects) at the moment **new**(c) is executed. In other words, each execution of **new**(c) results in the creation of one more object as instance of class c, and all these objects are executing the (same) body s (determined by c's declaration) in parallel. The execution of **new**(c) furthermore involves the creation of a new name, say α', which is used to identify the newly created object (instance of c). Normally, this name will be stored in some individual variable (occurring in the creat*ing* object), for later reference.

The snapshot in Figure 1 of a creat*ing* α and creat*ed* α' may help (see next page). This picture assumes that $d(c)_1(m') = \mu'$, ..., and that $d(c)_2 = s'$. Details on how the new name α' is to be determined follow in Section 4.3. The picture also reflects that individual variables (from now on rather called *instance* variables) are 'private' to the objects. Private variables are not accessible from other objects. In fact, the only way in which objects may interact is by the sending and receiving of *messages*. This takes place by an extended version of the rendez-vous mechanism. Instead of the earlier synchronized execution of m and $\bar{m}$ occurring in two parallel processes (leading to the execution of

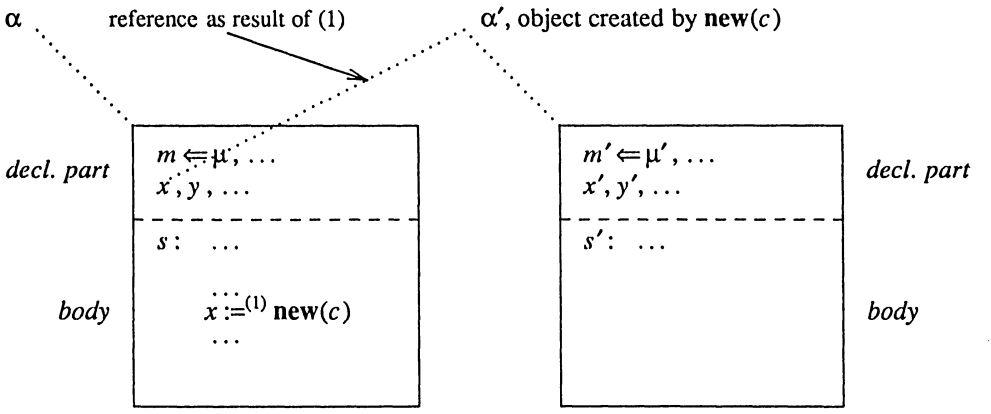


Figure 1: Two Objects

parallel processes (leading to the execution of the body $s = d(m)$ associated with m), we now have the following concept, execution of which is described in a number of steps:

1. a statement **answer**(m), when occurring in the body of an object (named, say, by object name α) indicates willingness to execute the method μ (associated with the method name m in the declaration of the class of which α is an instance) upon request;

2. a so-called send-expression $e!m(\vec{e})$, when occurring in the body of an object (named, say, by object name β) is executed as follows:

   - the value of the expression e is determined, resulting in the object γ; next
   - the values of the expressions $e_1, .., e_k$ are determined from left to right, resulting in $\gamma_1, .., \gamma_k$;
   - a request for execution of the method associated with method name m by the object name γ is issued

   [Step 2 takes place in parallel to Step 1];

3. in case the issue of this request synchronizes with the execution of the answer statement **answer**(m) as meant under 1 (implying that $\alpha = \gamma$), and assuming that $\mu = \lambda\vec{x}.e'$, next

4. the values $\gamma_1, .., \gamma_k$ are assigned to the (formal parameters, i.e., the) instance variables $x_1, .., x_k$, the expression e' is evaluated, the x_i are reset to their earlier values (which they had just before the assignment), and the result $\bar{\alpha}$ is returned to that position in object β where the value of $e!m(\vec{e})$ is required;

5. execution is resumed with the parallel execution (in α) of the statement following **answer**(m) and (in β) with the construct following $e!m(\vec{e})$.

All through the execution of 1. to 5., further parallel objects (different from α or β) will continue independently with their own activities. The only 'waiting' involved is (in α) for completing the evaluation of the method μ , and (in β) for the returning of the value $\bar{\alpha}$.

This brief sketch of the informal semantic of $\mathcal{L}_{po}$ should suffice here. More extensive explanations are contained in various studies on POOL semantics ([ABKR86, ABKR89, Rut90b, AR89a, AR90]. We have aimed at including, in $\mathcal{L}_{po}$, of all essential features of POOL. Concepts not treated are

- temporary variables (in addition to instance variables) and the object **nil**;
- the conditional answer statement, and an answer statement of the form **answer**$(m_1, .., m_k)$, $k \geq 1$;
- the method call (not as part of a rendez-vous);
- a few special cases of expressions;
- (a full treatment of) the standard objects.

Apart from the last item, the missing features can be dealt with without undue effort, by small extensions of the present definitions. Standard objects are more difficult since they are not, by nature, compact (cf. [Rut90b] for more information on this).

4.2 Syntax

The syntax for $\mathcal{L}_{po}$ may be inferred from that of $\mathcal{L}_{pp}$, as amended in the light of the extensions outlined above. Note that the new- and block constructs have been moved from the class of statements to that of expressions.
The following basic sets are used

- $(x \in) IVar$, a countable set of (individual or) instance variables
- $(m \in) M$, a finite set of method names
- $(\alpha, \beta, \gamma \in) SObj$, the syntactic set of *standard objects* (to be identified later with the semantic set of standard objects including the integers, truth values, and maybe more)
- $(c \in) Class$, a finite alphabet of *class names*.

We have no more use for the set *Func* . Finiteness of M and *Class* is, as before, postulated in order to avoid declarations with infinite domain.

Definition 4.1 (syntax for $\mathcal{L}_{po}$)

a. $s(\in Stat_{po}) \quad ::= \quad x := e \mid \textbf{answer}(m \mid (s_1; s_2) \mid \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \mid \textbf{while } e \textbf{ do } s \textbf{ od}$

b. $e(\in Exp_{po}) \quad ::= \quad \alpha \mid x \mid e!m(\vec{e}) \mid \textbf{new}(c) \mid (s; e) \mid \textbf{begin var } \vec{x} := \vec{e}; e \textbf{ end}$

c. $(d \in) Decl_{po} = Class \rightarrow ((M \rightarrow Meth) \times Stat_{po})$

d. $\mu(\in Meth) \quad ::= \quad \lambda \vec{x}.e$

e. $\pi(\in \mathcal{L}_{po}) \quad ::= \quad (d, \mathbf{new}(c))$

In clause e., we see that the execution of a program starts with the creating of a first object as instance of some (initial) class c.

4.3 Operational semantics

As before, we base the operational semantics on a transition system, now named $\mathcal{T}_{po}$. This will involve a somewhat extended notion of state, as well as an adapted notion of a, possibly labeled, syntactic continuation.

We begin with the introduction of the sets of objects and states.

Definition 4.2

a. $(\alpha, \beta, \gamma \in) Obj = SObj \cup ObjN$
Here $SObj$ is the set of standard objects, and $ObjN$ is a (not further specified) set of object names.

b. $(\sigma \in)\Sigma = (IVar \rightarrow Obj \rightarrow Obj) \times \mathcal{P}_{fin}(Obj)$.

c. The functions $new : \mathcal{P}_{fin}(Obj) \times Class \rightarrow ObjN$ and $class : ObjN \rightarrow Class$ will be introduced below.

d. The notation $\sigma[\beta/x, \alpha]$ abbreviates $<\sigma_1[\sigma_1(x)[\beta/\alpha]/x], \sigma_2>$: σ is changed such that $\sigma[\beta/x, \alpha](x)(\alpha)$ now equals β; elsewhere σ is not changed.

A state is a pair $\sigma = <\sigma_1, \sigma_2>$. For a given instance variable x and object name α, $\sigma_1(x)(\alpha)$ tells us the current value of x (in object α). Note that the 'same' x will have, in general, a different value $\sigma_1(x)(\bar{\alpha})$ in some other object $\bar{\alpha}$. Furthermore, $\sigma_2 \in \mathcal{P}_{fin}(Obj)$ consists of a *finite* subset of Obj which may be read as the collection of all objects currently active. (If one so desires, one may consider some or all of the standard objects (for integers, truth values and the like) as already active and supplied with suitable standard methods. These issues are dealt with at length in [ABKR89], [Rut90b], and are not further treated here.) The function *new* delivers, for a current set of active objects $\xi(\in \mathcal{P}_{fin}(Obj))$ and class c, a new name $new(\xi, c)$ not in ξ, which may be used to name a new instance of class c. The function *class* determines, for each object name α, the class $c = class(\alpha)$ of which α is an instance.
We proceed with the definition of the various continuations.

Definition 4.3

a. $(r \in) SySCo$ is the set of syntactic statement continuations given by

$$r \quad ::= \quad \mathrm{E} \mid (s : r) \mid (e : g) \mid r{<}\alpha/x{>} \mid g(\alpha) \mid \mathbf{if}\ \beta\ \mathbf{then}\ r_1\ \mathbf{else}\ r_2\ \mathbf{fi} \mid {<}\beta, m, \vec{\beta}{>} : g.$$

b. $(g \in) SyECo$ is the set of syntactic expression continuations given by

$$g \quad ::= \quad \lambda\alpha.r \mid g{<}\alpha/x{>} \mid \chi$$

c. $(\rho \in) LSySCo$ is the set of *labeled* syntactic statement continuations given by

$$\rho \quad ::= \quad <\alpha, r> \mid (\rho_1, \rho_2) \mid \alpha : \chi \mid <\beta, \rho>$$

d. $(\chi \in) LSyECo$ is the set of *labeled* syntactic expression continuations given by

$$\chi \quad ::= \quad <\alpha, g> \mid (\chi, \rho) \mid (\rho, \chi)$$

Anticipating the denotational semantics, we already mention that each ρ will correspond to some (mathematical) process in P, and each χ to some function in $Obj \rightarrow P$. Whereas $<\alpha, r>$ should be read as: have r executed by object α, the construct $\alpha : \chi$ has as intended meaning that the object α is passed as argument to (the function which is the meaning of) χ. The construct $e : \chi$ (special case of $e : g$) is normally evaluated by some object, say β. The value of the expression e is determined (with respect to β); eventually, its value, say γ, is passed on to χ (which itself may be a labeled construct, e.g., $<\alpha, g>$). The construct $<\beta, \rho>$ is auxiliary; the role of β is (eventually) no more than to be thrown away.

Below, we shall make extensive use of pairs $<\rho, \sigma>$ - to be read as: execute the labeled continuation ρ with state σ as argument. We adopt the convention that, in such a pair, ρ is always *consistent* with respect to σ. This requires, by definition, that all α appearing as labels in ρ are element of σ_2 (the set of currently active object names). Here we say that

- α appears as label in $<\alpha, r>$ or $<\alpha, g>$
- if α appears as label in ρ, ρ_1, ρ_2 or g, then α appears as label in (ρ_1, ρ_2), (ρ, χ) , (χ, ρ), $e : g$ or $<\beta, \rho>$.

A *transition* is a five-tuple (written in the arrow notation of Section 3) of one of three forms

- $<\rho, \sigma> \rightarrow_d <\rho', \sigma'>$,
- $<\rho, \sigma> \rightarrow_d <\rho', <\alpha, m>>$,
- $<\rho, \sigma> \rightarrow_d <\chi, <\beta, m, \vec{\beta}>>$.

The first possibility reflects a 'normal' step, the second results from an answer statement: $<\alpha, m>$ indicates that object α is willing to execute the method named by m, and the third results from a send expression, asking object β to execute m with parameters $\vec{\beta}$, with a result to be returned, upon completion of the method execution, to χ. A transition rule has the general form as described in Section 3. Rules of the form

$$\frac{<\rho_1, \sigma> \rightarrow_d \cdots}{<\rho_2, \sigma> \rightarrow_d \cdots}$$

with $\cdots$ standing for the *same* pair, will again be abbreviated to $<\rho_2, \sigma> \rightarrow_0 <\rho_1, \sigma>$, or even to $\rho_2 \rightarrow_0 \rho_1$. If ρ_2, ρ_1 are of the form $<\alpha, r_2>$, $<\alpha, r_1>$, respectively, we further simplify the notation to read $r_2 \rightarrow_0 r_1$.

We next present

Definition 4.4 (transition system $\mathcal{T}_{po}$ for $\mathcal{L}_{po}$)

s-rules

- zero-step

$$\begin{aligned}(x := e) : r &\rightarrow_0 e : \lambda\alpha.(r<\alpha/x>)\\ (s_1; s_2) : r &\rightarrow_0 s_1 : (s_2 : r)\\ \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} : r &\rightarrow_0 e : \lambda\beta.\textbf{if } \beta \textbf{ then } s_1 : r \textbf{ else } s_2 : r \textbf{ fi}\end{aligned}$$

- axioms

$$\begin{aligned}&<<\alpha, \textbf{answer}(m) : r>, \sigma> \rightarrow <<\alpha, r>, <\alpha, m>>\\ &<<\alpha, \textbf{while } e \textbf{ do } s \textbf{ od}, \sigma> \rightarrow\\ &\qquad <<\alpha, e : \lambda\beta.\textbf{if } \beta \textbf{ then } s : \textbf{while } e \textbf{ do } s \textbf{ od} : r \textbf{ else } r \textbf{ fi}>, \sigma>\end{aligned}$$

e-rules

- zero-step

$$\begin{aligned}\alpha : g &\rightarrow_0 g(\alpha)\\ (s; e) : g &\rightarrow_0 s : (e : g)\\ e!m(e_1, .., e_k) : g &\rightarrow_0 e : \lambda\beta.(e_1 : \lambda\beta_1.(\cdots\\ &\qquad e_k : \lambda\beta_k.(<\beta, m, \beta_1, .., \beta_k>: g)\cdots))\\ <<\alpha, \textbf{begin var } \vec{x} := \vec{e}; e \textbf{ end} : g &\rightarrow_0\\ <<\alpha, ((x_1 := e_1; ..; x_k := e_k); e) : g<\sigma_1(x_1)(\alpha)/x_1>..<\sigma_1(x_k)(\alpha)/x_k>>, \sigma>&\end{aligned}$$

- axioms

$$\begin{aligned}<<\alpha, x : g>, \sigma> &\rightarrow <<\alpha, \sigma_1(x)(\alpha) : g>, \sigma>\\ <<\alpha, \textbf{new}(c) : g>, \sigma> &\rightarrow <(<\alpha, \beta : g>, <\beta, s : \mathrm{E}>), \sigma'>\end{aligned}$$

where $\beta = new(\sigma_2, c)$, $\sigma' = <\sigma_1, \sigma_2 \cup \{\beta\}>$, and $d(c)_2 = s$

r, ρ, χ-rules

- zero-step

$$\begin{aligned}\textbf{if } tt \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi} &\rightarrow_0 r_1\\ \textbf{if } ff \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi} &\rightarrow_0 r_2\end{aligned}$$

$$\begin{aligned}(\lambda\alpha.r)(\beta) &\rightarrow_0 r[\beta/\alpha]\\ g<\alpha/x>(\beta) &\rightarrow_0 g(\beta)<\alpha/x>\\ <\alpha, g>(\beta) &\rightarrow_0 <\alpha, g(\beta)>\\ (\chi, \rho)(\beta) &\rightarrow_0 (\chi(\beta)), \rho)\\ (\rho, \chi)(\beta) &\rightarrow_0 (\rho, \chi(\beta))\\ <\beta, \rho> &\rightarrow_0 \rho\end{aligned}$$

- axioms

$$\begin{aligned}<<\alpha, <\beta, m, \vec{\beta}>: g>, \sigma> &\rightarrow <<\alpha, g>, <\beta, m, \vec{\beta}>>\\ <<\alpha, r<\beta/x>>, \sigma> &\rightarrow <<\alpha, r>, \sigma[\beta/x, \alpha]>\end{aligned}$$

- rules for parallel execution

(interleaving)

$$\frac{<\rho,\sigma>\rightarrow <\rho',\sigma'>}{\begin{array}{c}<(\rho,\bar{\rho}),\sigma>\rightarrow <(\rho',\bar{\rho}),\sigma'> \\ <(\bar{\rho},\rho),\sigma>\rightarrow <(\bar{\rho},\rho'),\sigma'>\end{array}}$$

and similar rules with $<\alpha,m>$ replacing σ', or with χ' replacing ρ' and $<\beta,m,\vec{\beta}>$ replacing σ'

(rendez-vous)

$$\frac{<\rho_1,\sigma>\rightarrow <\chi',<\beta,m,\vec{\beta}>>,<\rho_2,\sigma>\rightarrow <\rho'',<\beta,m>>}{<(\rho_1,\rho_2),\sigma>\rightarrow <\beta,\textbf{begin var } \vec{x} := \vec{\beta}; e \textbf{ end} : (\chi',\rho'')>,\sigma>}$$

where $d(class(\beta))_1(m) = \lambda\vec{x}.e$

Explanation. Most of the rules should be clear as refinement of those of $\mathcal{T}_{pp}$. We emphasize that (even when no object label α is explicitly written) all calculations take place as part of named objects: eventually, all access to variables is through the function application $\sigma_1(x)(\alpha)$ in the axiom for $<<\alpha,x:g>,\sigma>$. The answer statement executed in α determines a step $<\alpha,m>$; the send expression $e!m(\vec{e})$ evaluates e and $e_1,..,e_k$ from left to right, and makes a step involving the outcome $<\beta,m,<\beta_1,..,\beta_k>>$. The **new**$(c)$ expression determines a new object name β (on the basis of the current set of names σ_2 and the class name c), and initiates execution of $<\beta,s:\mathrm{E}>$, where s, the body of class c, is retrieved from $d(c)_2$. In the rendez-vous of ρ_1, ρ_2, where ρ_1 may make a send-step $<\chi',<\beta,m,\vec{\beta}>>$ and ρ_2 a (corresponding) answer step $<\rho'',<\beta,m>>$, the body of the method $\mu = \lambda\vec{x}.e$ associated with m in the declaration is, after appropriate initialization with the parameters $\vec{\beta}$, executed, and the result is eventually passed back to χ'. (If desired, one may refine the rendez-vous rule by the introduction of dependent and independent resumptions, cf. the remark at the end of Section 3.)
We next discuss how to use $\mathcal{T}_{po}$ to determine $\mathcal{O}$ for $\mathcal{L}_{po}$. First, we introduce the domain P which serves as range for $\mathcal{O}$. Corresponding to the three kinds of right-hand sides of a transition (viz. $<\rho',\sigma'>$, $<\rho',<\alpha,m>>$, $<\chi,<\beta,m,\vec{\beta}>>$), it is natural to define P as solution of the equation

$$\begin{aligned}P = \{p_0\} \cup (\Sigma \rightarrow \mathcal{P}_{co}(\Sigma \times P \quad \cup \\ Obj \times M \times P \cup Obj \times M \times Obj^* \times (Obj \rightarrow P)))\end{aligned}$$

Using this P, we define $\mathcal{O}$ as fixed point of a contracting higher order operator Φ_d based on $\mathcal{T}_{po}$. Since we now deal with transitions yielding both $<\rho',..>$ and $<\chi',..>$ results, we introduce Φ_d as an operator on *pairs* of meaning functions $F = <F_1,F_2>$:

Definition 4.5

a. Let $F_1 \in LSySCo \rightarrow P$, $F_2 \in LSyECo \rightarrow Obj \rightarrow P$, and let Φ_d have the type
$\Phi_d : (LSySCo \rightarrow P) \times (LSyECo \rightarrow Obj \rightarrow P) \rightarrow$
$(LSySCo \rightarrow P) \times (LSyECo \rightarrow Obj \rightarrow P)$,
where $\Phi_d(F_1,F_2) =_{df} <\hat{F}_1,\hat{F}_2>$ is given as

$\hat{F}_1(\rho) = p_0$ if all r occurring in ρ are equal to E, and otherwise

$$\begin{aligned}\hat{F}_1(\rho) = \lambda\sigma.\ & \{<\sigma', F_1(\rho')> \mid <\rho,\sigma> \rightarrow <\rho',\sigma'>\} \ \cup \\ & \{<<\beta,m>, F_1(\rho')> \mid <\rho,\sigma> \rightarrow <\rho', <\beta,m>>\} \ \cup \\ & \{<<\beta,m,\vec{\beta}>, F_2(\chi')> \mid <\rho,\sigma> \rightarrow <\chi', <\beta,m,\vec{\beta}>>\}\end{aligned}$$

and

$\hat{F}_2(\chi) = \lambda\alpha.p_0$ if all r occurring in χ are equal to E, and otherwise

$$\begin{aligned}\hat{F}_2(\chi) = \lambda\alpha.\lambda\sigma.\ & \{<\sigma', F_1(\rho')> \mid <\alpha:\chi,\sigma> \rightarrow <\rho',\sigma'>\} \ \cup \\ & \{<<\beta,m>, F_1(\rho')> \mid <\alpha:\chi,\sigma> \rightarrow <\rho', <\beta,m>>\} \ \cup \\ & \{<<\beta,m,\vec{\beta}>, F_2(\chi')> \mid <\alpha:\chi,\sigma> \rightarrow <\chi', <\beta,m,\vec{\beta}>>\}\end{aligned}$$

b. $\mathcal{O}_d = fix(\Phi_d)_1$, $\mathcal{O}(d, \mathbf{new}(c)) = \mathcal{O}_d(<\alpha, s : \mathrm{E}>)$, where $\alpha = \mathbf{new}(\emptyset, c)$, and $d(c)_2 = s$.

Thus, in order to execute $(d, \mathbf{new}(c))$, the first instance of c is named by α – obtained when the set of active objects is still empty – and execution of the body of this object (given in the declaration of c) is initiated.

4.4 Denotational semantics

Similar to what we did in Section 3, we define the intermediate denotational mappings

$$\begin{aligned}\mathcal{I}_d &: Stat_{po} \rightarrow Obj \rightarrow P \rightarrow P, \\ \mathcal{E}_d &: Exp_{po} \rightarrow Obj \rightarrow (Obj \rightarrow P) \rightarrow P.\end{aligned}$$

Let f range over $Obj \rightarrow P$.

Definition 4.6 (denotational semantics for $\mathcal{L}_{pp}$)

- statements

$$\begin{aligned}\mathcal{I}_d(x := e)(\alpha)(p) &= \mathcal{E}_d(e)(\alpha)(\lambda\beta.\lambda\sigma.\{<\sigma[\beta/x,\alpha], p>\}) \\ \mathcal{I}_d(m)(\alpha)(p) &= \lambda\sigma.\{<<\alpha,m>, p>\} \\ \mathcal{I}_d(s_1; s_2)(\alpha)(p) &= \mathcal{I}_d(s_1)(\alpha)(\mathcal{I}_d(s_2)(\alpha)(p)) \\ \mathcal{I}_d(&\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2\ \mathbf{fi})(\alpha)(p) \\ &= \mathcal{E}_d(e)(\lambda\beta.\mathbf{if}\ \beta\ \mathbf{then}\ \mathcal{I}_d(s_1)(\alpha)(p)\ \mathbf{else}\ \mathcal{I}_d(s_2)(\alpha)(p)\ \mathbf{fi}) \\ \mathcal{I}_d(&\mathbf{while}\ e\ \mathbf{do}\ s\ \mathbf{od})(\alpha)(p) = \lambda\sigma.\{<\sigma, \mathcal{E}_d(e)(\alpha)(\lambda\beta. \\ &\mathbf{if}\ \beta\ \mathbf{then}\ \mathcal{I}_d(s)(\alpha)(\mathcal{I}_d(\mathbf{while}\ e\ \mathbf{do}\ s\ \mathbf{od})(\alpha)(p))\ \mathbf{else}\ p\ \mathbf{fi})>\}\end{aligned}$$

- expressions

$$\begin{aligned}\mathcal{E}_d(\beta)(\alpha)(f) &= f(\beta) \\ \mathcal{E}_d(x)(\alpha)(f) &= \lambda\sigma.\{<\sigma, f(\sigma_1(x)(\alpha))>\} \\ \mathcal{E}_d(s; e)(\alpha)(f) &= \mathcal{I}_d(s)(\alpha)(\mathcal{E}_d(e)(\alpha)(f)) \\ \mathcal{E}_d(e!m(\vec{e}))(\alpha)(f) &= \mathcal{E}_d(e)(\alpha)(\lambda\beta.(\mathcal{E}_d(e_1)(\alpha)(\lambda\beta_1.(\cdots \\ &\qquad \mathcal{E}_d(e_k)(\alpha)(\lambda\beta_k.\lambda\sigma.\{<<\beta,m,\vec{\beta}>, f>\})\cdots)))) \\ \mathcal{E}_d(\mathbf{new}(c))(\alpha)(f) &= \lambda\sigma.\{<\sigma', f(\beta) \parallel \mathcal{I}_d(s)(\beta)(p_0)>\} \\ &\quad \text{where } \beta = new(\sigma_2, c),\ \sigma' = <\sigma_1, \sigma_2 \cup \{\beta\}> \text{ and } d(c)_2 = s \\ \mathcal{E}_d(&\mathbf{begin\ var}\ \vec{x} := \vec{e}; e\ \mathbf{end})(\alpha)(f) = \\ &\lambda\sigma.\mathcal{E}_d(\vec{x} := \vec{e}; e)(\alpha)(\lambda\beta.\lambda\bar{\sigma}.\{<\bar{\sigma}[\sigma_1(x)(\alpha)/x_1]\cdots[\sigma_1(x_k)(\alpha)/x_k], f(\beta)>\})(\sigma)\end{aligned}$$

The '||'-operator used in the clause for new(c) is defined in

Definition 4.7 Let $p_1, p_2 \in P$, $X, Y \in \mathcal{P}_{co}(\cdot)$. We put

$$p_1 \parallel p_2 = \lambda\sigma.((p_1(\sigma) \between p_2) \cup (p_2(\sigma) \between p_1) \cup (p_1(\sigma) \mid_\sigma p_2(\sigma)))$$

where

$$\begin{aligned} X \parallel p = \; & \{<\sigma, p' \parallel p> \mid <\sigma, p'> \in X\} \quad \cup \\ & \{<<\alpha, m>, p' \parallel p> \mid <<\alpha, m>, p'> \in X\} \quad \cup \\ & \{<<\beta, m, \vec{\beta}>, f \parallel p> \mid <<\beta, m, \vec{\beta}>, f> \in X\} \end{aligned}$$

$$f \parallel p = \lambda\alpha.(f(\alpha) \parallel p)$$

$$\begin{aligned} X \mid_\sigma Y = \{ \; & <\sigma, \mathcal{E}_d(\textbf{begin var } \vec{x} := \vec{\beta}; e \textbf{ end})(\beta)(f \parallel p') \mid \\ & <<\beta, m, \vec{\beta}>, f> \in X, <\beta, m, p'> \in Y \\ & \text{or vice versa, and } d(class(\beta))_1(m) = \lambda\vec{x}.e\} \end{aligned}$$

As in Section 3, the above definitions are circular in that $\mathcal{E}_d$ depends on the definition of $\between$, and $\parallel$ depends on the definition of $\mathcal{E}_d$. We again refer to [BV91] for a rigorous definition of a comparable problem. (In the present setting, contractivity of the relevant higher-order operator follows easily from the $<\sigma', \ldots>$ step in the clause for $\mathcal{E}_d$(new(c)) and the $<\sigma, \ldots>$ step in the clause for $X \mid_\sigma Y$.) Also, the definition of $\mathcal{I}_d$ is not well-formed since it is circular in the case of the while statement. This problem as well may be dealt with by the familiar argument.

We are, at last, ready for the final

Definition 4.8 The denotational meaning $\mathcal{D} : \mathcal{L}_{po} \to P$ is given by

$$\mathcal{D}(d, \text{new}(c)) = \mathcal{I}_d(s)(\alpha)(p_0)\,,$$

where $\alpha = new(\emptyset, c)$ and $s = d(c)_2$.

In Section 5, we shall sketch the proof of the

Second Main Theorem For each $\pi \in \mathcal{L}_{po}$, $\mathcal{O}(\pi) = \mathcal{D}(\pi)$.

5 Equivalence of $\mathcal{O}$ and $\mathcal{D}$

We shall provide a detailed presentation of the proof that $\mathcal{O}$ and $\mathcal{D}$ coincide on $\mathcal{L}_{pp}$. For $\mathcal{L}_{po}$, we shall only outline the main ideas.

We start with the equivalence proof for $\mathcal{L}_{pp}$. We assume the various definitions from Section 3; in addition we give several further definitions which will link the syntactic continuations r to their denotations involving semantics continuations.

Definition 5.1 The mappings

$$\begin{aligned} \mathcal{R}_d &: SySCo \to P \\ \mathcal{G}_d &: SyECo \to V \to P \end{aligned}$$

are given as follows

a.

$$\begin{aligned}
\mathcal{R}_d(\mathrm{E}) &= p_0 \\
\mathcal{R}_d(s:r) &= \mathcal{I}_d(s)(\mathcal{R}_d(r)) \\
\mathcal{R}_d(e:g) &= \mathcal{E}_d(e)(\mathcal{G}_d(g)) \\
\mathcal{R}_d(r_1, r_2) &= \mathcal{R}_d(r_1) \parallel \mathcal{R}_d(r_2) \\
\mathcal{R}_d(\textbf{if } \beta \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi}) &= \textbf{if } \beta \textbf{ then } \mathcal{R}_d(r_1) \textbf{ else } \mathcal{R}_d(r_2) \textbf{ fi} \\
\mathcal{R}_d(r\langle\alpha/x\rangle) &= \lambda\sigma.\{\langle\sigma[\alpha/x], \mathcal{R}_d(r)\rangle\} \\
\mathcal{R}_d(g(\alpha)) &= \mathcal{G}_d(g)(\alpha)
\end{aligned}$$

b.

$$\mathcal{G}_d(\lambda\alpha.r) = \lambda\beta.\mathcal{R}_d(r[\beta/\alpha]),\ \beta \text{ fresh}$$

We now state a central lemma relating the transition system $\mathcal{T}_{pp}$ and the $\mathcal{R}_d$-function:

Lemma 5.2 If $r_1 \rightarrow_0 r_2$ then $\mathcal{R}_d(r_1) = \mathcal{R}_d(r_2)$.

Proof In all the cases this is immediate by the definitions of $\mathcal{T}_{pp}$ and of $\mathcal{R}_d$. □

Next, we introduce complexity measures on *SySCo* and *SyECo* (and on $Stat_{pp}$, Exp_{pp}), which will play a role in an inductive argument in the proof of the key theorem below.

Definition 5.3 The mappings $\| \cdot \|_r : SySCo \rightarrow N$ (and analogously $\| \cdot \|_g$, $\| \cdot \|_s$, $\| \cdot \|_e$) are defined by

a. $\| \mathrm{E} \|_r = 0$, $\| s:r \|_r = \| s \|_s + \| r \|_r$, $\| e:g \|_r = \| e \|_e + \| g \|_g$, $\| (r_1, r_2) \|_r = \| r_1 \|_r + \| r_2 \|_r$, $\| r\langle\alpha/x\rangle \|_r = \| r \|_r$, $\| g(\alpha) \|_r = \| g \|_g + \| \alpha \|_e$,
$\| \textbf{if } \beta \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi} \|_r = \max(\| r_1 \|_r, \| r_2 \|_r) + 1$.

b. $\| \lambda\alpha.r \|_g = \| r \|_r$.

c. $\| x := e \|_s = \| x \|_e + \| e \|_e + 1$, $\| m \|_s = 1$, $\| s_1; s_2 \|_s = \| s_1 \|_s + \| s_2 \|_s + 1$,
$\| \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \|_s = \| e \|_e + (\max(\| s_1 \|_s, \| s_2 \|_s) + 1) + 1$,
$\| \textbf{while } e \textbf{ do } s \textbf{ od} \|_s = \| e \|_e + \| s \|_s + 1$, $\| \textbf{new}(s) \|_s = \| s \|_s + 1$,
$\| \textbf{begin var } x := e; s \textbf{ end} \|_s = \| x := e; s \|_s + 1$.

d. $\| \alpha \|_e = \| x \|_e = 1$, $\| \phi(e_1, .., e_k) \|_e = 1 + \| e_1 \|_e + \cdots + \| e_k \|_e + 1$, $\| s; e \|_e = \| s \|_s + \| e \|_e + 1$.

It is not difficult to verify that

Lemma 5.4 If $r_1 \rightarrow_0 r_2$ then $\| r_1 \|_r > \| r_2 \|_r$.

Proof By the various definitions. Note, e.g., that $\| \phi(e_1, .., e_k) \|_e = 1 + \sum_{i=1}^{k} \| e_i \|_e + 1$, but $\| \bar{\phi}(\alpha_1, .., \alpha_k) \|_e = 1$, since $\bar{\phi}(\alpha_1, .., \alpha_k)$ is an element of V. □

The main step leading to the proof that $\mathcal{O} = \mathcal{D}$ on $\mathcal{L}_{pp}$ now follows. The key idea is to show (following a method from [KR90]) that the denotational mapping $\mathcal{R}_d$ is a fixed point of the contracting higher-order operator Φ_d which we used earlier to *define* $\mathcal{O}_d$. This then implies that $\mathcal{R}_d = \mathcal{O}_d$, from which $\mathcal{O} = \mathcal{D}$ is immediate.

Theorem 5.5 $\Phi_d(\mathcal{R}_d)(r) = \mathcal{R}_d(r)$, for all $r \in SySCo$.

Proof Induction on $\| r \|_r$. If $r = \mathrm{E}$, the result is clear. We now discuss a selection of subcases for r, leaving the most difficult case that $r = (r_1, r_2)$ to the last.

- $r \equiv (x := e) : r_1$

$$
\begin{aligned}
&\Phi_d(\mathcal{R}_d)((x := e) : r_1) \\
&\quad = \text{ def. } \Phi_d, \text{ def. } \mathcal{T}_{pp} \\
&\Phi_d(\mathcal{R}_d)(e : \lambda\alpha.r_1{<}\alpha/x{>}) \\
&\quad = \text{ ind. hyp.} \\
&\mathcal{R}_d(e : \lambda\alpha.r_1{<}\alpha/x{>}) \\
&\quad = \text{ Lemma 5.2} \\
&\mathcal{R}_d((x := e) : r_1).
\end{aligned}
$$

- $r \equiv (s_1; s_2) : r_1$

$$
\begin{aligned}
&\Phi_d(\mathcal{R}_d)((s_1; s_2) : r_1) \\
&\quad = \text{ def. } \Phi_d, \text{ def. } \mathcal{T}_{pp} \\
&\Phi_d(\mathcal{R}_d)(s_1 : (s_2 : r_1)) \\
&\quad = \text{ ind. hyp.} \\
&\mathcal{R}_d(s_1 : (s_2 : r_1)) \\
&\quad = \text{ Lemma 5.2} \\
&\mathcal{R}_d((s_1; s_2) : r_1)
\end{aligned}
$$

- $r \equiv \textbf{while } e \textbf{ do } s \textbf{ od} : r_1$

$$
\begin{aligned}
&\Phi_d(\mathcal{R}_d)(\textbf{while } e \textbf{ do } s \textbf{ od} : r_1) \\
&\quad = \text{ def. } \Phi_d, \text{ def. } \mathcal{T}_{pp} \\
&\lambda\sigma.\{{<}\sigma, \mathcal{R}_d(e : \lambda\beta.\textbf{if } \beta \textbf{ then } (s; \textbf{while } e \textbf{ do } s \textbf{ od}) : r_1 \textbf{ else } r_1 \textbf{ fi}){>}\} \\
&\quad = \text{ def. } \mathcal{R}_d \\
&\lambda\sigma\{{<}\sigma, \mathcal{E}_d(e)(\lambda\beta.\textbf{if } \beta \textbf{ then } \mathcal{I}_d(s)(\mathcal{I}_d(\textbf{while } e \textbf{ do } s \textbf{ od})(\mathcal{R}_d(r_1))) \textbf{ else } \mathcal{R}_d(r_1) \textbf{ fi}){>}\} \\
&\quad = \text{ def. } \mathcal{I}_d \\
&\mathcal{I}_d(\textbf{while } e \textbf{ do } s \textbf{ od})(\mathcal{R}_d(r_1)) \\
&\quad = \text{ def. } \mathcal{R}_d \\
&\mathcal{R}_d(\textbf{while } e \textbf{ do } s \textbf{ od} : r_1)
\end{aligned}
$$

$r \equiv (x : g)$

$$
\begin{aligned}
&\Phi_d(\mathcal{R}_d)(x : g) \\
&\quad = \text{ def. } \Phi_d, \ \mathcal{T}_{pp} \\
&\lambda\sigma.\{{<}\sigma, \mathcal{R}_d(\sigma(x) : g){>}\} \\
&\quad = \text{ def. } \mathcal{R}_d \\
&\lambda\sigma.\{{<}\sigma, \mathcal{E}_d(\sigma(x))(\mathcal{G}_d(g)){>}\} \\
&\quad = \text{ def. } \mathcal{E}_d \\
&\lambda\sigma.\{{<}\sigma, \mathcal{G}_d(g)(\sigma(x)){>}\} \\
&\quad = \text{ def. } \mathcal{E}_d \\
&\mathcal{E}_d(x)(\mathcal{G}_d(g)) \\
&\quad = \text{ def. } \mathcal{R}_d \\
&\mathcal{R}_d(x : g)
\end{aligned}
$$

- $r \equiv (r_1, r_2)$ Take any σ.

$$
\begin{array}{l}
\Phi_d(\mathcal{R}_d)(r_1,r_2)(\sigma) \\
\quad = \text{ def. } \Phi_d \\
\{<\tau, \mathcal{R}_d(\bar{r})> \mid <(r_1,r_2),\sigma> \rightarrow <\bar{r},\tau>\} \\
\quad = \text{ def. } \mathcal{T}_{pp} \\
\{<\tau', \mathcal{R}_d(r',r_2)> \mid <r_1,\sigma> \rightarrow <r',\tau'>\} \quad \cup \\
\{<\tau'', \mathcal{R}_d(r_1,r'')> \mid <r_2,\sigma> \rightarrow <r'',\tau''>\} \quad \cup \\
\{<\sigma, \mathcal{R}_d(s:(r',r''))> \mid <r_1,\sigma> \rightarrow <r',m>, <r_2,\sigma> \rightarrow <r'',\bar{m}>, d(m) = s\} \\
\quad = \\
\{<\tau', \mathcal{R}_d(r')> \mid <r_1,\sigma> \rightarrow <r',\tau'>\} \between \mathcal{R}_d(r_2) \quad \cup \\
\{<\tau'', \mathcal{R}_d(r'')> \mid <r_2,\sigma> \rightarrow <r'',\tau''>\} \between \mathcal{R}_d(r_1) \quad \cup \\
\{<\sigma, \mathcal{I}_d(s)(\mathcal{R}_d(r') \parallel \mathcal{R}_d(r''))> \mid <r_1,\sigma> \rightarrow <r',m>, <r_2,\sigma> \rightarrow <r'',\bar{m}>, d(m) = s\} \\
\quad = \text{ see below for (*)} \\
\Phi_d(\mathcal{R}_d)(r_1)(\sigma) \between \mathcal{R}_d(r_2) \cup \Phi_d(\mathcal{R}_d)(r_2)(\sigma) \between \mathcal{R}_d(r_1) \cup \\
\quad (*)\Phi_d(\mathcal{R}_d)(r_1)(\sigma) \mid_\sigma \Phi_d(\mathcal{R}_d)(r_2)(\sigma) \\
\quad = \text{ ind. hyp.} \\
\mathcal{R}_d(r_1)(\sigma) \between \mathcal{R}_d(r_2) \quad \cup \quad \mathcal{R}_d(r_2)(\sigma) \between \mathcal{R}_d(r_1) \quad \cup \quad \mathcal{R}_d(r_1)(\sigma) \mid_\sigma \mathcal{R}_d(r_2)(\sigma) \\
\quad = \\
(\mathcal{R}_d(r_1) \parallel \mathcal{R}_d(r_2))(\sigma) \\
\quad = \\
\mathcal{R}_d(r_1,r_2)(\sigma)
\end{array}
$$

where the step leading to (*) is justified as follows:

$$
\begin{array}{l}
\Phi_d(\mathcal{R}_d)(r_1)(\sigma) \mid_\sigma \Phi_d(\mathcal{R}_d)(r_2)(\sigma) \\
\quad = \\
\{<\sigma, \mathcal{I}_d(s)(p' \parallel p'')> \mid <m,p'> \in \Phi_d(\mathcal{R}_d)(r_1)(\sigma), \\
\qquad\qquad <\bar{m},p''> \in \Phi_d(\mathcal{R}_d)(r_2)(\sigma), d(m) = s\} \\
\quad = \\
\{<\sigma, \mathcal{I}_d(s)(p' \parallel p'')> \mid \ <m,p'> \in \{<\tau', \mathcal{R}_d(r')> \mid <r_1,\sigma> \rightarrow <r',\tau'>\}, \\
\qquad\qquad <\bar{m},p''> \in \{<\tau'', \mathcal{R}_d(r'')> \mid <r_2,\sigma> \rightarrow <r'',\tau''>\}, d(m) = s\} \\
\quad = \\
\{<\sigma, \mathcal{I}_d(s)(\mathcal{R}_d(r') \parallel \mathcal{R}_d(r''))> \mid <r_1,\sigma> \rightarrow <r',m>, <r_2,\sigma> \rightarrow <r'',\bar{m}>, d(m) = s\}
\end{array}
$$

□

Finally, we can prove

First Main Theorem For $\pi \in \mathcal{L}_{pp}$, $\mathcal{O}(\pi) = \mathcal{D}(\pi)$.

Proof $\mathcal{O}(d,s) = \mathcal{O}_d(s:\mathrm{E}) = \mathcal{R}_d(s:\mathrm{E}) = \mathcal{I}_d(s)(p_0) = \mathcal{D}(d,s)$ □

Remark The above proof suggests that, once $\mathcal{T}_{pp}$ is in the 'right' form, and $\mathcal{D}$ and the semantic operators follow the structure of $\mathcal{T}_{pp}$, then the proof that $\mathcal{O} = \mathcal{D}$ follows more or less 'automatically', i.e., it may be completely syntax driven without an appeal to additional arguments. In [Rut90a], it has been established that this is indeed the case for transition systems (and associated $\mathcal{D}$) of a restricted format. We conjecture that the approach of [Rut90a] may be generalized to cover the present case as well. This

would require, more specifically, a better understanding of how continuations might be incorporated in the method of [Rut90a].

We next outline how the proof that $\mathcal{O} = \mathcal{D}$ on $\mathcal{L}_{po}$ may be structured extending the above approach. We first provide the counterpart of Definition 5.1.

Definition 5.6

a. The mapping $\mathcal{R}_d : SySCo \rightarrow Obj \rightarrow P$ is given by

$$\begin{aligned}
\mathcal{R}_d(\mathrm{E}) &= p_0 \\
\mathcal{R}_d(s : r)(\alpha) &= \mathcal{I}_d(s)(\alpha)(\mathcal{R}_d(r)(\alpha)) \\
\mathcal{R}_d(e : g)(\alpha) &= \mathcal{E}_d(e)(\alpha)(\mathcal{G}_d(g)(\alpha)) \\
\mathcal{R}_d(\textbf{if } \beta \textbf{ then } r_1 \textbf{ else } r_2 \textbf{ fi})(\alpha) &= \textbf{if } \beta \textbf{ then } \mathcal{R}_d(r_1)(\alpha) \textbf{ else } \mathcal{R}_d(r_2)(\alpha) \textbf{ fi} \\
\mathcal{R}_d(r{<}\beta/x{>})(\alpha) &= \lambda\sigma.\{{<}\sigma[\beta/x, \alpha], \mathcal{R}_d(r)(\alpha){>}\} \\
\mathcal{R}_d(g(\beta))(\alpha) &= \mathcal{G}_d(g)(\alpha)(\beta) \\
\mathcal{R}_d({<}\beta, m, \vec{\beta}{>} : g)(\alpha) &= \lambda\sigma.\{{<}{<}\beta, m, \vec{\beta}{>}, \mathcal{G}_d(g)(\alpha){>}\}
\end{aligned}$$

b. The mapping $\mathcal{G}_d : SyECo \rightarrow Obj \rightarrow Obj \rightarrow P$ is given by

$$\begin{aligned}
\mathcal{G}_d(\lambda\beta.r)(\alpha) &= \lambda\gamma.\mathcal{R}_d(r[\gamma/\beta])(\alpha), \gamma \text{ fresh} \\
\mathcal{G}_d(g{<}\beta/x{>})(\alpha) &= \lambda\gamma.\lambda\sigma.\{{<}\sigma[\beta/x, \alpha], \mathcal{G}_d(g)(\alpha)(\gamma){>}\}, \gamma \text{ fresh} \\
\mathcal{G}_d(\chi)(\alpha) &= \mathrm{R}_d(\chi)
\end{aligned}$$

c. The mapping $\mathrm{R}_d : LSySCo \rightarrow P$ is given by

$$\begin{aligned}
\mathrm{R}_d({<}\alpha, r{>}) &= \mathcal{R}_d(r)(\alpha) \\
\mathrm{R}_d(\rho_1, \rho_2) &= \mathrm{R}_d(\rho_1) \parallel \mathrm{R}_d(\rho_2) \\
\mathrm{R}_d({<}\beta, \rho{>}) &= \mathrm{R}_d(\rho) \\
\mathrm{R}_d(\alpha : \chi) &= \mathrm{X}_d(\chi)(\alpha)
\end{aligned}$$

d. The mapping $\mathrm{X}_d : LSyECo \rightarrow Obj \rightarrow P$ is given by

$$\begin{aligned}
\mathrm{X}_d({<}\alpha, g{>}) &= \mathcal{G}_d(g)(\alpha) \\
\mathrm{X}_d(\chi, \rho) &= \mathrm{X}_d(\chi) \parallel \mathrm{R}_d(\rho) \\
\mathrm{X}_d(\rho, \chi) &= \mathrm{R}_d(\rho) \parallel \mathrm{X}_d(\chi)
\end{aligned}$$

Again we have

Lemma 5.7 If $r_1 \rightarrow_0 r_2$ (with respect to $\mathcal{T}_{po}$), then $\mathcal{R}_d(r_1) = \mathcal{R}_d(r_2)$. □

Similar to the proof of Theorem 5.5 (assuming an appropriate generalization of the complexity measures $\| \cdot \|$), we can now prove

Theorem 5.8 $\Phi_d(R_d, \mathrm{X}_d)(\rho, \chi) = {<}R_d(\rho), \mathrm{X}_d(\chi){>}$. □

From this, the second main theorem follows directly:

Second Main Theorem For $\pi \in \mathcal{L}_{po}$, $\mathcal{O}(\pi) = \mathcal{D}(\pi)$. □

References

[AB88] P. America and J.W. de Bakker. Designing equivalent semantic models for process creation. *Theoretical Computer Science*, 60:109–176, 1988.

[ABKR86] P. America, J.W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. Operational semantics of a parallel object-oriented language. In *Proc. POPL'86*, pages 194–208, St. Petersburg, Florida, 1986.

[ABKR89] P. America, J.W. de Bakker, J.N. Kok, and J.J.M.M. Rutten. Denotational semantics of a parallel object-oriented language. *Information and Computation*, 83:152–205, 1989.

[Ame89] P. America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1:366–411, 1989.

[AR89a] P. America and J.J.M.M. Rutten. A parallel object-oriented language: design and semantic foundations. In J.W. de Bakker, editor, *Languages for Parallel Architectures: Design, Semantics, Implementation Models*, Wiley Series in Parallel Computing, pages 1–49. Wiley, 1989.

[AR89b] P. America and J.J.M.M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. *Journal of Computer and System Sciences*, 39:343–375, 1989.

[AR90] P. America and J.J.M.M. Rutten. A layered semantics for a parallel object-oriented language. Technical Report CS-R9052, CWI, Amsterdam, 1990. To appear in "Foundations of Object-Oriented Languages", LNCS 489, Springer, 1991.

[BBKM84] J.W. de Bakker, J.A. Bergstra, J.W. Klop, and J.-J.Ch. Meyer. Linear time and branching time semantics for recursion with merge. *Theoretical Computer Science*, 34:135–156, 1984.

[BV91] J.W. de Bakker and E.P. de Vink. CCS for OO and LP. In *Proc. TAPSOFT'91*. LNCS, Springer, 1991. To appear.

[BM88] J.W. de Bakker and J.-J.Ch. Meyer. Metric semantics for concurrency. *BIT*, 28:504–529, 1988.

[BMOZ88] J.W. de Bakker, J.-J.Ch. Meyer, E.-R. Olderog, and J.I. Zucker. Transition systems, metric spaces and ready sets in the semantics of uniform concurrency. *Journal of Computer and System Sciences*, 36:158–224, 1988.

[BW90] J.W. de Bakker and J.H.A. Warmerdam. Metric pomset semantics for a concurrent language with recursion. In I. Guessarian, editor, *Proc. 18ème Ecole de Printemps d'Informatique, Semantique du Parallelisme*, pages 21–49. LNCS 469, Springer, 1990.

[BZ82] J.W. de Bakker and J.I. Zucker. Processes and the denotational semantics of concurrency. *Information and Control*, 54:70–120, 1982.

[Dug66] J. Dugundji. *Topology.* Allyn and Bacon, 1966.

[Eli91] A. Eliëns. *DLP - a Language for Distributed Logic Programming.* PhD thesis, Universiteit van Amsterdam, 1991.

[Eng89] R. Engelking. *General Topology*, volume 6 of *Sigma Series in Pure Mathematics.* Heldermann, revised and completed edition, 1989.

[GS90] C.A. Gunter and D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 633–674. North-Holland, 1990.

[HBR90] E. Horita, J.W. de Bakker, and J.J.M.M. Rutten. Fully abstract denotational semantics for nonuniform concurrent languages. Technical Report CS-R9027, CWI, Amsterdam, 1990.

[KR90] J.N. Kok and J.J.M.M. Rutten. Contractions in comparing concurrency semantics. *Theoretical Computer Science*, 76:180–222, 1990.

[Kur56] K. Kuratowski. Sur une méthode de métrisation complète des certains espaces d'ensembles compacts. *Fundamenta Mathematicae*, 42:114–138, 1956.

[Mac71] S. MacLane. *Categories for the working mathematician*, volume 5 of *Graduate texts in mathematics.* Springer, 1971.

[MV88] J.-J.Ch. Meyer and E.P. de Vink. Applications of compactness in the Smyth powerdomain of streams. *Theoretical Computer Science*, 57:251–382, 1988.

[Par81] D. Park. Concurrency and automata on infinite sequences. In P. Deussen, editor, *Proc 5th GI Conference*, pages 167–183. LNCS 104, Springer, 1981.

[Rut89] J.J.M.M. Rutten. Correctness and full abstraction of metric semantics for concurrency. In J.W. de Bakker, W.P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 628–659. LNCS 354, Springer, 1989.

[Rut90a] J.J.M.M. Rutten. Deriving metric models for bisimulation from transition system specifications. In M. Broy and C.B. Jones, editors, *Proc. IFIP TC2 Working conference on programming concepts and methods*, pages 155–177. North-Holland, 1990.

[Rut90b] J.J.M.M. Rutten. Semantic correctness for a parallel object-oriented language. *SIAM Journal on Computing*, 19:341–383, 1990.

Embeddings Among Concurrent Programming Languages

Ehud Shapiro
Department of Applied Mathematics and Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel

Abstract

In a previous paper [1] we developed an algebraic framework for language comparison based on language embeddings, which are mappings that preserve certain aspects of both the syntactic and the semantic structure of the language. We provided separation results by demonstrating the nonexistence of various kinds of embeddings among languages. In this talk we survey the framework and complement the separation results with positive results, i.e., demonstrate embeddings among various well-known concurrent languages. Together the positive and negative results induce a preordering on the family of concurrent programming languages that quite often coincides with previous intuitions on the "expressive power" of these languages.

[1] Shapiro, E., Separating Concurrent Languages with Categories of Language Embeddings, in *Proceedings of 23rd Annual ACM Symposium on Theory of Computing*, ACM, 1991.

Invariants and Paradigms of Concurrency Theory

Ryszard Janicki
Department of Computer Science and Systems
McMaster University
Hamilton, Ontario, Canada, L8S 4K1

Maciej Koutny
Computing Laboratory
The University of Newcastle upon Tyne
Newcastle upon Tyne NE1 7RU, U.K.

Abstract

We introduce a new invariant semantics of concurrent systems which is a direct generalisation of the causal partial order semantics. Our new semantics overcomes some of the problems encountered when one uses causal partial orders alone. We discuss various aspects of the new invariant model. In particular, we outline how the new invariants can be generated by 1-safe inhibitor Petri nets.

1 Introduction

In the development of mathematical models of concurrent behaviours, the concept of partial and total order undoubtedly occupies a central position. Interleaving models use total orders of event occurrences, while so-called 'true concurrency' models use step sequences or causal partial orders (comp. [BD87,Mi80,Ho85,Pr86]). Even more complex structures, such as failures [Ho85] or event-structures [Wi82], are in principle based on the concept of total or partial order. While interleavings and step sequences usually represent executions or observations, the causality relation represents a set of executions or observations. The lack of order between two event occurrences in the case of step sequence is interpreted as simultaneity, while in the case of causality relation is interpreted as independency. Both interleaving and true concurrency models have been developed to a high degree of sophistication providing a framework for specification and verification of concurrent systems. However, some of the behavioural aspects of concurrent behaviour are difficult to tackle in the interleaving or partial order based setting. For instance, the specification of priorities using partial orders alone is rather problematic, in particular, if the events are not instantaneous (see [La85,Ja87,JL88,BK91]). Another example are inhibitor nets (see [Pe81]) which are virtually admired by practitioners and almost completely rejected by theoreticians, in our opinion mainly because their concurrent behaviour cannot be properly defined in terms of causality based structures. We believe that problems of this kind follow from an implicit assumption that all behavioural properties of concurrent systems can be adequately modelled in terms of causality based structures. We claim that the structure of concurrency phenome-

non is richer, with causality being only one of several fundamental invariants generated by sets of equivalent executions or observations. In this paper we will show how such invariants can be defined and constructed.

2 MOTIVATION

We start by discussing two specific situations which we believe identify an inherent inability of the causal partial order semantics to properly cope with some of the aspects of the non-sequential behaviour. We will use Petri nets [Pe81,Re85] as the system model, however this does not mean that our approach is restricted to Petri nets. COSY with priorities, or TCSP with priorities could be used as well (comp. [JL88]).

The first example closely follows the discussion in [Ja87,JL88]. We consider a concurrent system *Con* comprising two sequential subsystems A and B such that: (1) A can engage in event a and after that in event b; (2) B can engage either in event b or in event c; (3) the two sequential subsystems synchronise by means of the handshake communication; and (4) the specification of *Con* includes a priority constraint stating that whenever it is possible to execute b then c must not be executed.
The priority Petri net in Figure 1 illustrates this example. We now observe that causal partial orders cannot provide a satisfactory semantical model of *Con*. We first note that in the initial state both events a and c are enabled and can be executed simultaneously (note that the priority constrained is not violated since b is not enabled in the initial state). Thus in any causality based model *Con* generates a causal partial order with one occurrence of a and one occurrence of c such that there is no causal relationship between the two event occurrences. Now, since a and c are independent, it should be possible to execute c followed by a, and a followed by c. Whereas the former execution sequence does not violate the priority constraint, the latter does as after executing a event b becomes enabled and c must not be executed. Note that in [BK91] it was observed that whether the simultaneous execution of a and c should be allowed is related to whether or not one can regard a as an event taking some time. If a is instantaneous then the step $\{a,c\}$ should not be allowed, and then a causal partial order semantics of *Con* can be constructed along the lines described in [BK91]. If, however, a cannot be regarded as instantaneous (possibly because it is a compound event) then one should

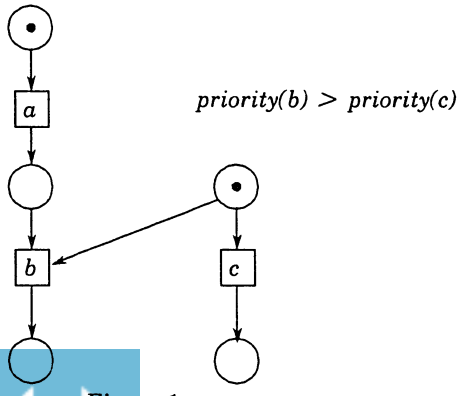


Figure 1

look for an invariant model more expressive than causal partial orders to capture the behaviour of *Con*.

As the second example we consider a system which supports an error recovery mechanism. That mechanism is invoked by an occurrence of a special signalling event, *err*, which may occur simultaneously with any other event in the system. The result of an occurrence of *err* is that: (1) the error recovery procedure is called and its successful completion it signified by an occurrence of a special event *rcv*; and (2) during the error recovery no event in the system is allowed to be executed.

We again observe that the causal partial orders do not provide a satisfactory model of the system's behaviour. For it is possible to execute *err* simultaneously with some other event, say a, and then after the termination of the error recovery procedure to execute event *rcv*. In any causal partial order which might underlay such a system history, the occurrences of *err* and a must be independent, and the occurrence of *rcv* must not precede the occurrence of a. This, however, means that it is possible to execute *err* followed by a and *rcv*, violating (2).

The above two examples show that causal partial orders are not expressive enough to satisfactorily model the invariant properties of certain kinds of concurrent systems. In the rest of this paper we will outline an alternative invariant semantics which overcomes the problems highlighted in the above two examples.

The overall goal of this paper might be explained in the following way. Consider the nets of Figure 2. Two of them, PN_4 and PN_5, are nets employing inhibitor arcs. (An inhibitor arc between place p and transition t means that t can be enabled only if p is unmarked [Pe81].) We want to define an invariant semantics of these nets in such a way that the following would hold (below by a 'complete' history or execution of net P_i we mean one which involves exactly one occurrence of a and one occurrence of b).

(1) Different nets generate different complete concurrent histories.
(2) Each net except PN_3 generates one complete concurrent history.
(3) In each case a concurrent history is defined on the same level of abstraction as the causal partial order.

Taking into account only complete executions (or observations) expressed in terms of step sequences, we might define the semantics of the nets in the following way. Let $\sigma_1,\sigma_2,\sigma_3$ be the

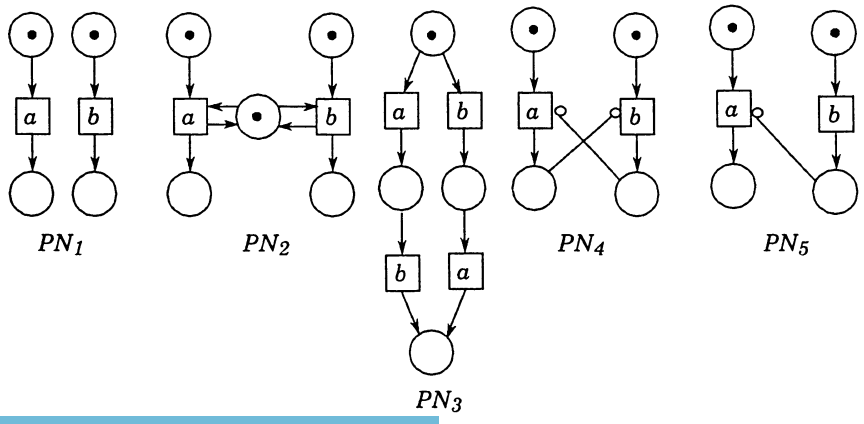


Figure 2

step sequences $\sigma_1 = \{a\}\{b\}$, $\sigma_2 = \{b\}\{a\}$ and $\sigma_3 = \{a,b\}$. Then:

$Steps(PN_1) = \{\sigma_1,\sigma_2,\sigma_3\}$
$Steps(PN_2) = \{\sigma_1,\sigma_2\}$
$Steps(PN_3) = \{\sigma_1,\sigma_2\}$
$Steps(PN_4) = \{\sigma_3\}$
$Steps(PN_5) = \{\sigma_1,\sigma_3\}$.

Step sequences cannot distinguish between PN_2 and PN_3, and do not tell us that each of PN_1, PN_2, PN_4, PN_5 generates in fact only one complete concurrent history. That each of PN_1, PN_4 and PN_5 generates only one complete concurrent history is intuitively obvious (no conflict occurs in these nets). However, this may be not so clear in the case of PN_2. Moreover, one might ask why at all should we distinguish between PN_2 and PN_3. To show that making such a distinction may in some cases be appropriate we consider a program statement:

$a: x:=x+1 \quad \& \quad b: x:=x+3$

where "&" denotes the *commutativity* operator (see [LH82]) which means that the instructions a and b may be performed in any order but never simultaneously. We believe that this statement should generate *one* concurrent history comprising two essentially equivalent executions, σ_1 and σ_2, rather than two different concurrent histories, one comprising σ_1, the other σ_2. Thus, since PN_2 seems to be a natural implementation of the commutativity operator, it should also generate one complete concurrent history. On the other hand, PN_3 is clearly a net generating two different complete histories. Thus each PN_i $(i=1,2,4,5)$ should generate exactly one complete concurrent history H_i, where: $H_1 = \{\sigma_1,\sigma_2,\sigma_3\}$, $H_2 = \{\sigma_1,\sigma_2\}$, $H_4 = \{\sigma_3\}$ and $H_5 = \{\sigma_1,\sigma_3\}$; while PN_3 should generate two complete concurrent histories: $H_{31} = \{\sigma_1\}$ and $H_{32} = \{\sigma_2\}$. A question which one might now ask is whether we could define these histories in a more structured and compact way, for example, by using causality-like relations? There are only three causal relationships involving one occurrence of a and one occurrence of b, namely:

c_1 = *a and b are independent*
c_2 = *a precedes b*
c_3 = *a follows b*.

Clearly, c_1 characterises history H_1, c_2 characterises H_{31} and c_3 characterises H_{32}. This means that none of H_2, H_4 and H_5 can be characterised by a suitable causal relationship. To solve the problem we then observe that causality and independency can be characterised in the following way:

If a and b are two events involved in a concurrent history H then
a <u>*precedes*</u> *b* *if in all executions belonging to H, a precedes b.*
a <u>*follows*</u> *b* *if in all executions belonging to H, a follows b.*
If a neither precedes nor follows b, then a and b are <u>*independent*</u>.

By following the above pattern we now can introduce three new invariant relations, called *commutativity*, *synchronisation* and *weak causality*, in the following way:

a <u>*comm*</u> *b* *if in all executions belonging to H, either a precedes b or b precedes a*
a <u>*synch*</u> *b* *if in all executions belonging to H, a is simultaneous with b*
a <u>*wc*</u> *b* *if in all executions belonging to H, a precedes or is simultaneous with b.*

One now may observe that *a* <u>*comm*</u> *b* characterises H_2; *a* <u>*synch*</u> *b* characterises H_4; and *a* <u>*wc*</u> *b* characterises H_5. The new invariant relations can be used to distinguish between the five nets of Figure 2, but it is not at all clear yet whether they would work in the general case. One might also ask several other questions, such as: How can one define commutativity, syn-

chronisation and weak causality in the general case? What is their relationship to the causality relation as well as their mutual relationship? Are there other relations of this kind? These are examples of questions we will try to answer in this paper. As we already mentioned, the distinction between the concurrent histories generated by PN_2 and PN_3 *may or may not be* desirable, depending on the intended interpretation of the nets. Another question which seems to be interesting in this context is whether there is a formal mechanism which, when switched on makes PN_2 and PN_3 semantically different, and when switched off makes them semantically identical. It is then worth observing that under the assumption that for every allowed concurrent history: *the existence of the executions in the opposite orders implies the existence of a simultaneous execution*, PN_2 and PN_3 become equivalent as H_2 is no longer a valid history (we have to decompose it onto H_{31} and H_{32}). We will call such rules *paradigms*, and show how they can be defined and used.

3 THE MODEL

The model we are going to develop is a three-level model: Systems-Invariants-Observations, and we will proceed from the bottom (i.e. the observation level) to the top of the hierarchy. In this paper we will focus on the invariant level. We will provide only the most basic results concerning the observation level (for more details see [JK90,JK90a]), while the system level will be considered in an informal manner at the end of this paper.

3.1 OBSERVATIONS

We define observation as an abstract model of execution. More precisely, by an observation we will mean a special report supplied by an observer who can perceive the evolution of a concurrent system. Such an observer has to fill in a (possibly infinite) matrix with rows and columns being indexed by event occurrences. The observer is supposed to fill in the entire matrix except the diagonal using only three symbols: $\rightarrow$, $\leftarrow$ and $\leftrightarrow$, with $\rightarrow$ denoting precedence, $\leftarrow$ following, and $\leftrightarrow$ simultaneity. (How the observer makes his judgement is beyond our interest.) Together with a natural interpretation of the precedence relation this means that observations can be represented by *partially ordered sets* of event occurrences, where ordering represents *precedence*, and incomparability represents *simultaneity*.

A *partially ordered set* (or *poset*) is a pair $po=(X,R)$, where X is a non-empty set and $R \subseteq X \times X$ is an irreflexive ($\neg aRa$) and transitive ($aRb \wedge bRc \Rightarrow aRc$) relation. We say po is *total* if for all different $a,b \in X$, aRb or bRa. We also denote: $dom(po)=X$, $\rightarrow_{po}=R$, $\leftarrow_{po}=R^{-1}$ and $\leftrightarrow_{po}=\{(a,b) \in X \times X \mid a \neq b \wedge \neg aRb \wedge \neg bRa\}$. Not all partial orders may be interpreted as possible observations. The additional properties we require are that: (1) the observer perceives only a single thread of time, and can only observe a finite number of events in a finite period of time and that (2) an event can last only for a finite period of time. It can be shown that (1) and (2) lead to the following definition of an observation of a concurrent history (see [JK90,JK90a] for details).

Observation is an initially finite interval order of event occurrences.

Note that a poset is initially finite if for every $a \in dom(po)$, the set $\{b \in dom(po) \mid \neg a \rightarrow_{po} b\}$ is finite, and that a poset po is an *interval order* if $(a \rightarrow_{po} b \wedge c \rightarrow_{po} d) \Rightarrow (a \rightarrow_{po} d \vee c \rightarrow_{po} b)$. The definition of interval order is taken from [Fi70], however the origin of this concept can be traced back to Wiener's 1914 paper [Wn14], where he considered interval orders as a way to

analyse temporal events, each event occurring over some finite time span. The main characterisation of interval orders is given below.

THEOREM 1 [Fi70]
A countable poset po is an interval order if and only if there are ϕ, $\rho : dom(po) \to Reals$ such that $\rho(a) > 0$ for all a, and if $a,b \in dom(po)$ then: $a \to_{po} b \Leftrightarrow \phi(a) + \rho(a) < \phi(b)$. □

The above result was strengthened in [JK90a] by showing that we can additionally require that ϕ is injective. The general properties of interval orders and their applications to the measurement theory were discussed in [Fi85], while the application of interval orders to model observations of concurrent histories was discussed in [JK90,JK90a].

A *step sequence* is an initially finite poset po such that $(a \leftrightarrow_{po} b \wedge b \leftrightarrow_{po} c \wedge a \neq c) \Rightarrow a \leftrightarrow_{po} c$, while an *interleaving sequence* is an initially finite total order. Let Obs, Obs_{step}, Obs_{itl} denote respectively the sets of observations, step sequences and interleaving sequences. We have $Obs_{itl} \subseteq Obs_{step} \subseteq Obs$, and throughout the rest of this paper, o (with an index, if necessary) will usually range over Obs.

3.2 INVARIANTS AND HISTORIES

A description of a concurrent system solely in terms of the observations it may generate is unsatisfactory for many reasons. In fact, any argument made in favour of causal partial orders existing in the literature (see, for instance, [BD85]), can also be used to support the introduction of the new invariants. We will first focus on the relationship between different observations of a concurrent history, where a concurrent history is essentially an invariant or a *set of invariants* satisfied by all its observations. It will be shown that the familiar causality relation is just *one* of many possible invariant relations. There are, of course, different ways in which an invariant might be defined for a given set of observations, depending on the specific kind of properties of the system one is interested in. In this paper we restrict ourselves to invariants which seem to be the most basic ones.

A *report set* is a non-empty set Δ of observations such that $dom(o_1) = dom(o_2)$ for all $o_1, o_2 \in \Delta$. We will denote by $dom(\Delta)$ the common domain of the observations in Δ. Note that a report set may be considered as the first approximation of a concurrent history.

Let Δ be a report set with the domain Σ. A *simple (binary) relational invariant* of Δ, is a relation $I \subseteq \Sigma \times \Sigma$ which can be characterised by: $(a,b) \in I :\Leftrightarrow a \neq b \wedge \forall o \in \Delta.\ \Phi(a,b,o)$, where $\Phi(a,b,o)$ is any formula derived from the following grammar:

$\Phi := true \mid false \mid a \to_o b \mid a \leftarrow_o b \mid a \leftrightarrow_o b \mid \neg\Phi \mid \Phi \vee \Phi \mid \Phi \wedge \Phi.$

Some of the basic terms of the above grammar are redundant, e.g., $a \leftarrow_o b$ is equivalent to $\neg(a \to_o b \vee a \leftrightarrow_o b)$. However, this does not cause any problems, while increases readability. Let $SRI(\Delta)$ denote the set of all simple (binary) relational invariants of Δ, and let $\to_\Delta$, $\leftarrow_\Delta$, $\leftrightarrow_\Delta$, $\rightleftharpoons_\Delta$, $\nearrow_\Delta$, $\nwarrow_\Delta$ be binary relations on Σ such that for all $a,b \in \Sigma$,

$a \to_\Delta b :\Leftrightarrow a \neq b \wedge \forall o \in \Delta.\ a \to_o b$
$a \leftarrow_\Delta b :\Leftrightarrow a \neq b \wedge \forall o \in \Delta.\ a \leftarrow_o b$
$a \leftrightarrow_\Delta b :\Leftrightarrow a \neq b \wedge \forall o \in \Delta.\ a \leftrightarrow_o b$
$a \rightleftharpoons_\Delta b :\Leftrightarrow a \neq b \wedge \forall o \in \Delta.\ a \to_o b \vee a \leftarrow_o b$
$a \nearrow_\Delta b :\Leftrightarrow a \neq b \wedge \forall o \in \Delta.\ a \to_o b \vee a \leftrightarrow_o b$
$a \nwarrow_\Delta b :\Leftrightarrow a \neq b \wedge \forall o \in \Delta.\ a \leftarrow_o b \vee a \leftrightarrow_o b.$

The relations $\to_\Delta$, $\leftarrow_\Delta$ are called *causalities*, $\rightleftharpoons_\Delta$ *commutativity*, $\leftrightarrow_\Delta$ *synchronisation*, and $\nearrow_\Delta$, $\nwarrow_\Delta$ *weak causalities*. In the sequel we will use $\to, \leftarrow, \leftrightarrow, \rightleftharpoons, \nearrow, \nwarrow$ to denote mappings, called *in-*

variants, which for every report set return respectively $\rightarrow_\Delta, \leftarrow_\Delta, \leftrightarrow_\Delta, \rightleftharpoons_\Delta, \nearrow_\Delta, \nwarrow_\Delta$. The set of all invariants will be denoted by *SRI*.

PROPOSITION 2
For every report set Δ, $SRI(\Delta)=\{\varnothing, \rightarrow_\Delta, \leftarrow_\Delta, \leftrightarrow_\Delta, \rightleftharpoons_\Delta, \nearrow_\Delta, \nwarrow_\Delta, \Sigma\times\Sigma\text{-}id_\Sigma\}$, and there is Δ such that $SRI(\Delta)$ consists of eight different relations. □

PROPOSITION 3
$\leftarrow_\Delta=(\rightarrow_\Delta)^{-1}$, $\nwarrow_\Delta=(\nearrow_\Delta)^{-1}$, $\rightarrow_\Delta=\nearrow_\Delta\cap\rightleftharpoons_\Delta$ and $\leftrightarrow_\Delta=\nearrow_\Delta\cap\nwarrow_\Delta$. □

Due to the symmetry present in $SRI(\Delta)$ one can in fact consider only four non-trivial invariants, namely $\rightarrow_\Delta$, $\leftrightarrow_\Delta$, $\rightleftharpoons_\Delta$ and $\nearrow_\Delta$. Furthermore, $\rightarrow_\Delta$ and $\leftrightarrow_\Delta$ may be expressed in terms of $\nearrow_\Delta$ and $\rightleftharpoons_\Delta$, so it seems reasonable to try to find a possibly smallest sets of invariants from which all the relations in $SRI(\Delta)$ could be generated.

A *signature* of a non-empty family F of report sets is a set of invariants $S\subseteq SRI$ such that for all $\Delta,\Delta_o\in F$ we have: $(dom(\Delta)=dom(\Delta_o)\wedge\forall I\in S.\ I(\Delta)=I(\Delta_o))\Rightarrow(\forall I\in SRI.\ I(\Delta)=I(\Delta_o))$. A signature is *universal* if F is the family of all report sets. Moreover, a signature S of F is *minimal* if (1) no proper subset of S is a signature of F, and (2) for every $J\in S$ and every $I\in SRI\text{-}S$, if $I(\Delta)\subseteq J(\Delta)$ for all report sets Δ, then $(S\text{-}\{J\})\cup\{I\}$ is not a signature of F. I.e., a signature is minimal if it cannot be 'reduced' by removing any of its invariants (see (1)) or by replacing any invariant by a 'weaker' one (see (2)).

PROPOSITION 4
$\{\nearrow,\rightleftharpoons\}$ and $\{\nwarrow,\rightleftharpoons\}$ are the only minimal universal signatures. □

A history is a report set Δ which is a complete (w.r.t. certain viewpoint) representation of some phenomenon underlying the reports of Δ. This completeness is to be captured by requiring that Δ includes all reports satisfying the relevant properties which can be attributed to the report sets. In our approach, these properties are the domain of Δ, $dom(\Delta)$, and the simple report invariants generated by Δ, $SRI(\Delta)$.

For every $I\in SRI$, let Φ_I denote any formula (see the definition of a simple relational invariant and Proposition 2) such that $(a,b)\in I(\Delta)\Leftrightarrow\forall o\in\Delta.\ \Phi_I(a,b,o)$. Let Δ be a report set and $S\subseteq SRI$. The *S-closure* of Δ, denoted $\Delta\langle S\rangle$, is the set comprising all observations o such that $dom(o)=dom(\Delta)$ and for all $I\in S$, $(a,b)\in I(\Delta)\Rightarrow\Phi_I(a,b,o)$.

PROPOSITION 5
(1) $\Delta\subseteq\Delta\langle S\rangle$.
(2) If S is a universal signature then $\Delta\langle S\rangle=\Delta\langle SRI\rangle$. □

Consider a report set $\Delta=\{o_1,o_2\}$, where o_1 and o_2 are as in Figure 3. Then $a\rightleftharpoons_\Delta b$, $a\rightleftharpoons_\Delta c$ and $b\rightleftharpoons_\Delta c$. Hence $\Delta\langle\rightleftharpoons\rangle=\{o_1,o_2,o_3,o_4,o_5,o_6\}$, where the o_i $(i=3,4,5,6)$ are shown in Figure 3. Thus $\Delta\subsetneq\Delta\langle\rightleftharpoons\rangle$. Moreover, $\Delta\langle\rightleftharpoons\rangle=\Delta\langle SRI\rangle$. We now can introduce the central notion of this paper.

A history is a non-empty report set Δ such that $\Delta=\Delta\langle SRI\rangle$.

I.e., a history is a report set which is fully characterised by the invariants it generates. Thus if Δ is a history, denoted $\Delta\in Hist$, then the following essentially describe the same thing.

Δ
$(\varnothing,\rightarrow_\Delta,\leftarrow_\Delta,\leftrightarrow_\Delta,\rightleftharpoons_\Delta,\nearrow_\Delta,\nwarrow_\Delta,\Sigma\times\Sigma\text{-}id_\Sigma)$
$(\nearrow_\Delta,\rightleftharpoons_\Delta)$
$(\nwarrow_\Delta,\rightleftharpoons_\Delta)$
$(I_1(\Delta),\ldots,I_k(\Delta))$ where $\{I_1,\ldots,I_k\}$ is a signature of any F such that $\Delta\in F$.

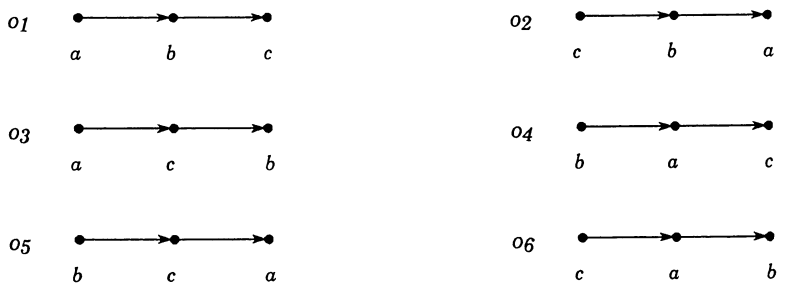


Figure 3

In concurrency theory, the causality relation is sometimes treated as an invariant, and sometimes as the set of all observations (step sequences or interleavings) it generates. We have just shown that this dual treatment can be generalised to other invariants in *SRI*.

3.3 Components and Paradigms

Let Δ be a concurrent history. The set $SRI(\Delta)$ can be treated as any other finite family of sets. In particular, we can find all the components defined by this family, as shown in Figure 4. There are seven non-empty components (non-empty means that there is Δ such that all seven components are non-empty), and we will denote $CSRI(\Delta)=\{\rightarrow_\Delta,\leftarrow_\Delta,\leftrightarrow_\Delta,\leftrightarrows_\Delta,\rightrightarrows_\Delta,\leftrightarrows_\Delta,\|_\Delta\}$.

A formula which says that a given relationship between two event occurrences a and b has been observed in Δ is called a *simple trait*. There are three simple traits: $\psi_\rightarrow \equiv \exists o\in\Delta.\ a\rightarrow_o b$, $\psi_\leftarrow \equiv \exists o\in\Delta.\ a\leftarrow_o b$ and $\psi_\leftrightarrow \equiv \exists o\in\Delta.\ a\leftrightarrow_o b$. One can easily show that the relations in $CSRI(\Delta)$ can be defined as conjunctions of simple traits and their negations.

PROPOSITION 6
For every $a,b\in dom(\Delta)$, we have

$a\rightarrow_\Delta b \Leftrightarrow \psi_\rightarrow \wedge \neg\psi_\leftarrow \wedge \neg\psi_\leftrightarrow$
$a\leftarrow_\Delta b \Leftrightarrow \neg\psi_\rightarrow \wedge \psi_\leftarrow \wedge \neg\psi_\leftrightarrow$
$a\leftrightarrow_\Delta b \Leftrightarrow \neg\psi_\rightarrow \wedge \neg\psi_\leftarrow \wedge \psi_\leftrightarrow$
$a\leftrightarrows_\Delta b \Leftrightarrow \psi_\rightarrow \wedge \psi_\leftarrow \wedge \neg\psi_\leftrightarrow$
$a\rightrightarrows_\Delta b \Leftrightarrow \psi_\rightarrow \wedge \neg\psi_\leftarrow \wedge \psi_\leftrightarrow$

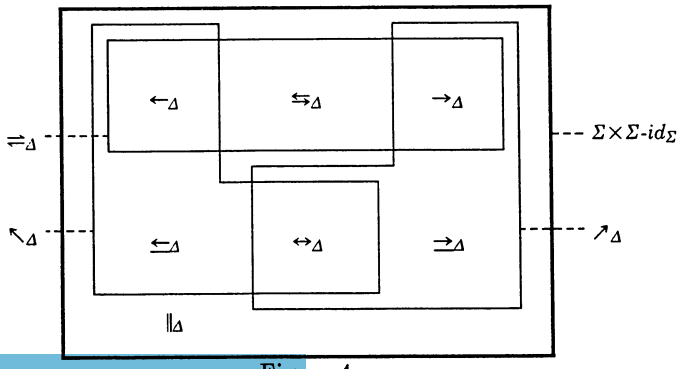


Figure 4

$a \underline{\leftarrow}_\Delta b \Leftrightarrow \neg\psi_\rightarrow \wedge \psi_\leftarrow \wedge \psi_\leftrightarrow$
$a \|_\Delta b \Leftrightarrow \psi_\rightarrow \wedge \psi_\leftarrow \wedge \psi_\leftrightarrow$. □

Since we have $\rightarrow_\Delta = (\leftarrow_\Delta)^{-1}$ and $\underline{\rightarrow}_\Delta = (\underline{\leftarrow}_\Delta)^{-1}$ we need to discuss only five components: $\rightarrow_\Delta$, $\|_\Delta$, $\leftrightarrows_\Delta$, $\leftrightarrow_\Delta$ and $\underline{\rightarrow}_\Delta$. The first component (and also an invariant), $\rightarrow_\Delta$, is *causality*. The second component, $\|_\Delta$, should be interpreted as *concurrency* (two events can be observed simultaneously and in both orders). Both causality and concurrency can be found in the models supporting the notion of true concurrency. The third component, $\leftrightarrows_\Delta$, represents what is usually referred to as *interleaving* (two events can be observed in both orders, but not simultaneously), and is usually dealt with on the level of observations rather than invariants. The fourth component (and also an invariant), $\leftrightarrow_\Delta$, can be interpreted as *synchronisation*. It is currently introduced only in its implicit form, e.g., as a silent action in CCS [Mi80]. The fifth component, $\underline{\rightarrow}_\Delta$, is not to our knowledge a part of any of the existing models. It captures *disabling* of one event by another event and was discussed in [Ja87] from where we took its intuitive meaning. As one now may see, the five components describe quite precisely the semantics of the nets of Figure 2, namely $a \|_{H_1} b$, $a \leftrightarrows_{H_2} b$, $a \rightarrow_{H_{31}} b$, $b \rightarrow_{H_{32}} a$, $a \leftrightarrow_{H_4} b$ and $a \underline{\rightarrow}_{H_5} b$.

The approach to concurrency which is based entirely on the concept of causality relation requires that for every concurrent history the following holds: if two event occurrences can be observed simultaneously, then they can also be observed in both orders, and vice versa. This means that every concurrent history besides being invariant-closed must also satisfy the following: $(\exists o \in \Delta.\ a \leftrightarrow_o b) \Leftrightarrow (\exists o \in \Delta.\ a \rightarrow_o b) \wedge (\exists o \in \Delta.\ a \leftarrow_o b)$. Note that this formula is built from simple traits. In general, any formula built in this way will be called a *paradigm*, and will characterise the internal structure of of concurrent histories.

Formally, the *paradigms*, $\omega \in Par$, are given by the following syntax.

$\omega ::= true \mid false \mid \psi_\rightarrow \mid \psi_\leftarrow \mid \psi_\leftrightarrow \mid \neg\omega \mid \omega \vee \omega \mid \omega \wedge \omega \mid \omega \Rightarrow \omega$

The evaluation of the formulas $\omega \in Par$ follows the standard rules [Mo76]. Note that in this grammar we need all three basic terms $\psi_\rightarrow$, $\psi_\leftarrow$ and $\psi_\leftrightarrow$.

A history $\Delta \in Hist$ *satisfies* a paradigm $\omega \in Par$ if for all $a,b \in dom(\Delta)$, $a \neq b \Rightarrow \omega(a,b,\Delta)$. We denote this by $\Delta \in Par(\omega)$. Two paradigms, ω and ω_o, are *equivalent*, $\omega \sim \omega_o$, if $Par(\omega) = Par(\omega_o)$. Let ω_i $(i = 1,\ldots,5)$ be the following paradigms:

$\omega_1 = \psi_\leftrightarrow \Rightarrow \psi_\rightarrow \vee \psi_\leftarrow$ $\qquad$ $\omega_2 = \psi_\rightarrow \wedge \psi_\leftarrow \Rightarrow \psi_\leftrightarrow$
$\omega_4 = \psi_\rightarrow \Rightarrow \psi_\leftarrow \vee \psi_\leftrightarrow$ $\qquad$ $\omega_3 = \psi_\rightarrow \wedge \psi_\leftrightarrow \Rightarrow \psi_\leftarrow$
$\omega_5 = \psi_\rightarrow \wedge \psi_\leftarrow \wedge \psi_\leftrightarrow \Rightarrow false$

PROPOSITION 7
(1) $\Delta \in Par(\omega_1) \Leftrightarrow \leftrightarrow_\Delta = \emptyset$.
(2) $\Delta \in Par(\omega_2) \Leftrightarrow \leftrightarrows_\Delta = \emptyset$.
(3) $\Delta \in Par(\omega_3) \Leftrightarrow \underline{\rightarrow}_\Delta = \underline{\leftarrow}_\Delta = \emptyset$.
(4) $\Delta \in Par(\omega_4) \Leftrightarrow \rightarrow_\Delta = \leftarrow_\Delta = \emptyset$.
(5) $\Delta \in Par(\omega_5) \Leftrightarrow \|_\Delta = \emptyset$. □

PROPOSITION 8 (equality up to $\sim$)
$Par = \{\omega_{i_1} \wedge \ldots \wedge \omega_{i_k} \mid k \leq 5 \wedge i_j \leq 5\}$. □

From the last proposition it follows that we have $2^5 = 32$ different paradigms. However, the nature of problems considered in Computer Science is such that two of the ω_i's may be safely rejected. The first ω_i that we reject is ω_4 since it rules out causality and hence invalidates the sequential composition construct. For a similar reason we reject ω_5 since it is not compatible

with the standard parallel composition operation. Thus we consider $2^3=8$ paradigms:

$\pi_1=true$ $\quad \pi_3=\omega_2 \quad \pi_5=\omega_1\wedge\omega_2 \quad \pi_7=\omega_2\wedge\omega_3$

$\pi_2=\omega_1 \quad \pi_4=\omega_3 \quad \pi_6=\omega_1\wedge\omega_3 \quad \pi_8=\omega_1\wedge\omega_2\wedge\omega_3$

PROPOSITION 9 (relationship between components and paradigms)

(1) $\Delta\in Par(\pi_1)$.
(2) $\Delta\in Par(\pi_2) \Leftrightarrow \leftrightarrow_\Delta=\emptyset$.
(3) $\Delta\in Par(\pi_3) \Leftrightarrow \leftrightarrows_\Delta=\emptyset$.
(4) $\Delta\in Par(\pi_4) \Leftrightarrow \rightarrow_\Delta=\emptyset$.
(5) $\Delta\in Par(\pi_5) \Leftrightarrow \leftrightarrow_\Delta=\leftrightarrows_\Delta=\emptyset$.
(6) $\Delta\in Par(\pi_6) \Leftrightarrow \leftrightarrow_\Delta=\rightarrow_\Delta=\emptyset$.
(7) $\Delta\in Par(\pi_7) \Leftrightarrow \leftrightarrows_\Delta=\rightarrow_\Delta=\emptyset$.
(8) $\Delta\in Par(\pi_8) \Leftrightarrow \leftrightarrow_\Delta=\leftrightarrows_\Delta=\rightarrow_\Delta=\emptyset$. □

We obtain a hierarchy of eight fundamental paradigms of concurrency shown in Figure 5. In this paper we will only discuss π_1,π_3 and π_8. Paradigm π_1 simply admits all concurrent histories. The most restrictive paradigm, π_8, is the paradigm adopted by the models supporting true concurrency semantics. As we pointed out earlier on, this paradigm has given rise to a number of elegant theories in the field of concurrency, however, it has some limitations such as an inability to model some aspects of systems with priorities. In the next section we will show that π_3 allows us to provide an invariant semantics for inhibitor nets, as well as for priority systems. The following major result characterises minimal signatures of the eight paradigms.

THEOREM 10

(1) $\{\nearrow,\rightleftharpoons\}$ is a minimal signature for $Par(\pi_1)$ and $Par(\pi_2)$.
(2) $\{\rightleftharpoons,\rightarrow\}$ is a minimal signature for $Par(\pi_4)$ and $Par(\pi_6)$.
(3) $\{\rightarrow,\nearrow\}$ is a minimal signature for $Par(\pi_3)$ and $Par(\pi_5)$.
(4) $\{\nearrow\}$ is a minimal signature for $Par(\pi_7)$.
(5) $\{\rightarrow\}$ is a minimal signature for $Par(\pi_8)$. □

Thus when the law $\exists o\in\Delta.\ a\leftrightarrow_o b \Leftrightarrow (\exists o\in\Delta.\ a\rightarrow_o b)\wedge(\exists o\in\Delta.\ b\rightarrow_o a)$ holds, then causality is *the only invariant that is needed,* and this fact is a *theorem* in our approach. Note that in the most general case (i.e. $Par(\pi_1)$) the explicit causality invariant is not needed. We also note that under the paradigm π_3 (and any other paradigm which contains it, i.e. π_5,π_7,π_8) we cannot distinguish between PN_2 and PN_3 of Figure 2.

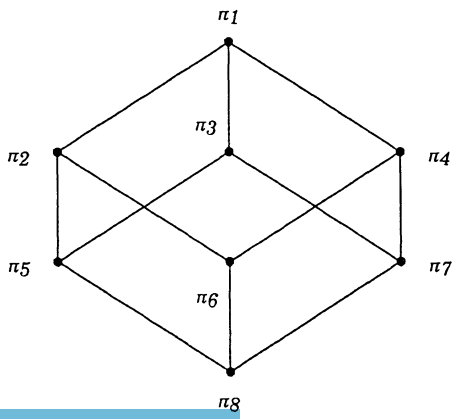


Figure 5

4 APPLICATIONS

4.1 INTERLEAVINGS INSIDE π_8

Paradigm π_8 deserves special attention as it is the only paradigm considered in the present literature. We will show that for histories satisfying paradigm π_8, one only needs the sequential observations. A *base* of a concurrent history Δ is a pair (Δ_o,S), where $\Delta_o \subseteq \Delta$ and $S \subseteq SRI$, such that $\Delta_o\langle S\rangle = \Delta$. In other words, a base provides a complete description of a history in terms of a (possibly smaller) set of observations and a suitable set of simple report invariants.

PROPOSITION 11
If $\Delta \in Par(\pi_8)$ and $\Delta_{itl} = \{o \in \Delta \mid o \in Obs_{itl}\}$ then $(\Delta_{itl},\{\rightarrow\})$ is a base of Δ. □

The above result means that in the case of paradigm π_8 it is possible to adequately represent a concurrent history by taking only its sequential observations. Clearly, this was the basic idea behind many models [Ma86,KP87], and can be traced back to [Sz30]. One should emphasise, however, that Proposition 11 cannot be extended to any other paradigm.

4.2. STEP SEQUENCES INSIDE π_3

In this and the next section we shall assume that all observations are step sequences, and that every history considered belongs to π_3. From Theorem 10 it follows that in this case $\{\nearrow,\rightarrow\}$ is a minimal signature. We shall provide an axiomatisation for this kind of signature and then define an invariant semantics of inhibitor nets. Below Obs_{step} denotes the set of all step sequences, and Σ_o denotes the set of all event occurrences.

A *pre-ordered* set is a pair (X,R) such that X is a non-empty set and $R \subseteq X \times X$ is an irreflexive $(\neg aRa)$ and weakly transitive $(aRb \wedge bRc \Rightarrow a=c \vee aRc)$ relation (see [Fr86]).

Note that for any Δ, the causality $\rightarrow_\Delta$ is always a poset, while the weak causality $\nearrow_\Delta$ is always a pre-ordered set. We will show that if $\Delta \subseteq Obs_{step}$ and π_3 holds, then the pair $\{\rightarrow,\nearrow\}$ can be modelled by a certain relational system which we call a *composet*.

A *combined partial order* (or *composet*) is a relational system $co=(X,P,R)$ such that X is a set and $P,R \subseteq X \times X$ are two relations satisfying the following.
(1) $\neg aRa$
(2) $aRb \wedge bRc \Rightarrow a=c \vee aRc$
(3) $aRb \Rightarrow \neg bPa$
(4) $aPb \Rightarrow aRb$
(5) $aRb \wedge bPc \Rightarrow aPc$
(6) $aPb \wedge bRc \Rightarrow aPc$.

Intuitively, P corresponds to $\rightarrow$, R corresponds to $\nearrow$, and X is a set of step sequence observations. The conditions (1) and (2) say that R is a pre-order; (4) indicates that P is included in R; (1),(4) together with, e.g., (5) imply that P is a poset; (3) is a kind of 'consistency' rule between the two orders; and (5), (6) give a kind of combined transitivity which ties together P and R.

COROLLARY 12
If (X,R,P) is a composet then (X,P) is a partially ordered set and (X,R) is a pre-ordered set. □

PROPOSITION 13
If $\Delta \subseteq Obs_{step}$ then $(dom(\Delta), \rightarrow_\Delta, \nearrow_\Delta)$ is a composet. □

The above proposition is not true if $\Delta\text{-}Obs_{step} \neq \emptyset$ since (5) and (6) may not hold. A relational system $rs=(X,P,R)$ with $P,R \subseteq X \times X$ is called a π_3^{step}*-history descriptor* if $X \subseteq \Sigma_o$, $R = \nearrow_{\Delta(rs)}$ and $P = \rightarrow_{\Delta(rs)}$, where $\Delta(rs) = \{o \in Obs_{step} \mid dom(o) = X \wedge (\forall a,b \in X.\ aRb \Rightarrow a \rightarrow_o b \vee a \leftrightarrow_o b) \wedge (\forall a,b \in X.\ aPb \Rightarrow a \rightarrow_o b)\}$.

THEOREM 14 (axiomatisation of $\{\nearrow, \rightarrow\}$)
Let X be a finite set. A relational system $rs=(X,R,P)$ is a π_3^{step}-history descriptor *if and only if* $X \subseteq \Sigma_o$ and rs is a composet. □

The assumption $\Delta \subseteq Obs_{step}$ is essential. Without it the result does not hold. The above theorem provides an axiomatisation of signatures for histories involving finite step sequences and conforming to the paradigm π_3 (and all paradigms below π_3 in the hierarchy of Figure 5). It says that every finite composet of event occurrences may be interpreted as a representation of a finite concurrent history of the kind described above. In other words, in this case concurrent histories can be unambiguously described by composets (in the same way as the histories in π_8 can be described by causal partial orders). For infinite histories the axiomatisation is less elegant as we have to take into account the fact that step sequences are initially finite posets. We will not discuss this issue in detail, but basically one needs to provide an analysis similar to that for infinite causal partial orders (see [BD85]).

There is certain similarity between our definition of the composet and the axioms for strong and weak precedence relation presented in [La86]. However, the way these two concepts are derived, the motivations, and the reasons for their introduction are quite different. Hence this similarity is either accidental or, as we would suggest, the composet is a natural generalisation of the concept of the partial order, and it may be useful for various, perhaps unrelated, applications.

4.3. COMPOSET SEMANTICS OF INHIBITOR NETS

In this section we outline a method of constructing the set of composets of a concurrent system represented by a 1-safe Petri nets with inhibitor arcs [Pe81]. Note that 1-safeness means that each place may hold at most one token. An inhibitor arc between place p and transition (event) t means that t can only be enabled if p is not marked. In the diagrams inhibitor arcs are identified by small circles. A technique similar to that described below might be used for other kinds of inhibitor nets, as well as for various priority models and nets (see [JL88]), however this would usually require the introduction of some new formal concepts.

The standard approach in which the partial order semantics of ordinary 1-safe Petri nets is derived employs *occurrence nets* [Re85]. An occurrence net can be regarded as a representation of a causality relation on event occurrences (or a single abstract history of the net). It is an unmarked acyclic net whose each place has at most one input and one output transition. Occurrence nets are obtained by unfolding marked nets and resolving the conflicts via the firing rules, as shown in Figure 6(a,b). Each occurrence net induces a poset on event occurrences derived in the following way: First an auxiliary relation $\rightarrow_{aux}$ is derived by transforming each three-node path $event_1 \rightarrow place \rightarrow event_2$ in the graph of the occurrence net into a pair $event_1 \rightarrow_{aux} event_2$. Then a poset is obtained by taking the transitive closure of $\rightarrow_{aux}$. For the occurrence net of Figure 6(b), the relation $\rightarrow_{aux}$ is shown in Figure 6(c), and the resulting poset is shown in Figure 6(d).

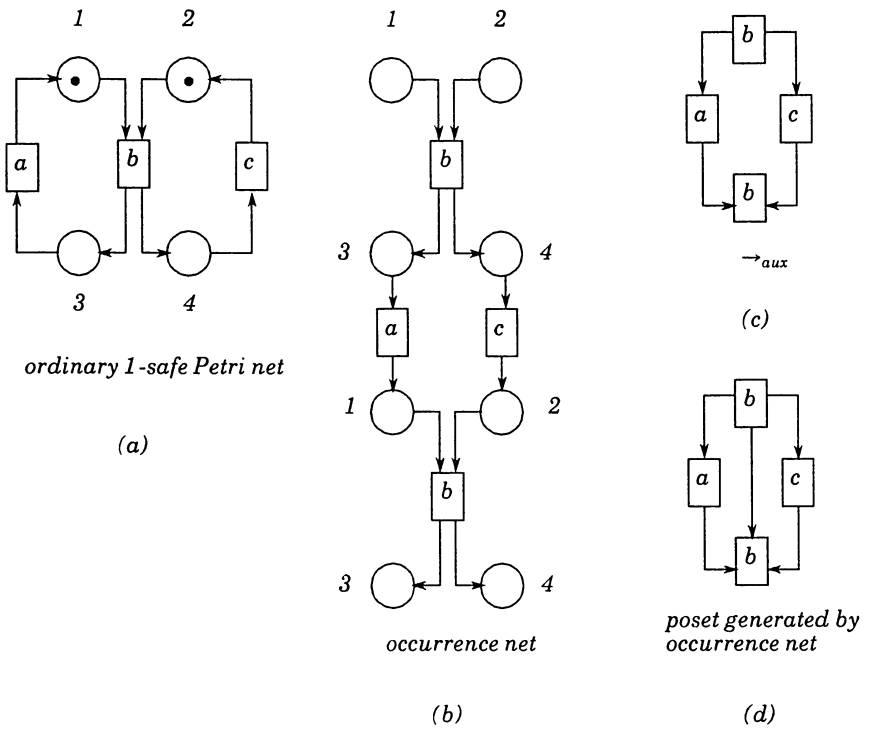


Figure 6

The way in which we construct composets for an inhibitor net will closely follow the above procedure. Let N_I be the inhibitor net shown in Figure 7(a). We first define an *occurrence net* of an inhibitor net by generalising in a straightforward way the standard definition of an occurrence net of an ordinary Petri net. The only new element is the handling of the inhibitor arcs. Since in the occurrence net places represent tokens, it is not possible to join c with place 2 using an inhibitor arc. However, we can join c with the *complement place* [Re85] of 2, i.e. place 5, using an activator arc (with a black dot at one end). Intuitively, this means that c can be executed only when 5 is marked. We also note that there is no restriction on the number of activator arcs which can be adjacent to a single place. A possible occurrence net for the inhibitor net N_I is shown in Figure 7(b). The next step is to transform the structural relationships embedded in the graph of the occurrence net into two auxiliary relations, $\rightarrow_{aux}$ and $\nearrow_{aux}$, from which the composet can be derived. There are three structural relationships which we need to consider, as shown in Figure 8(a). For the occurrence net of Figure 7(b) the two auxiliary relations are shown in Figure 8(b). The final step has to take into account the various transitivities which hold for a composet. More precisely, if $\rightarrow_{aux}$ and $\nearrow_{aux}$ have been defined for an occurrence net ON with Σ being the set of event occurrences, then the composet induced by ON is defined as $co(ON)=(\Sigma,\rightarrow,\nearrow)$, where $(\Sigma,\rightarrow,\nearrow)$ is a minimal (w.r.t. set inclusion for both $\rightarrow$ and $\nearrow$) composet such that $\rightarrow_{aux}\subseteq\rightarrow$ and $\nearrow_{aux}\subseteq\nearrow$. It can be shown that $co(ON)$ is well-defined (i.e. it always exists and is uniquely defined). The algorithm for deriving $co(ON)$ is a straightforward generalisation of an algorithm which yields the transitive

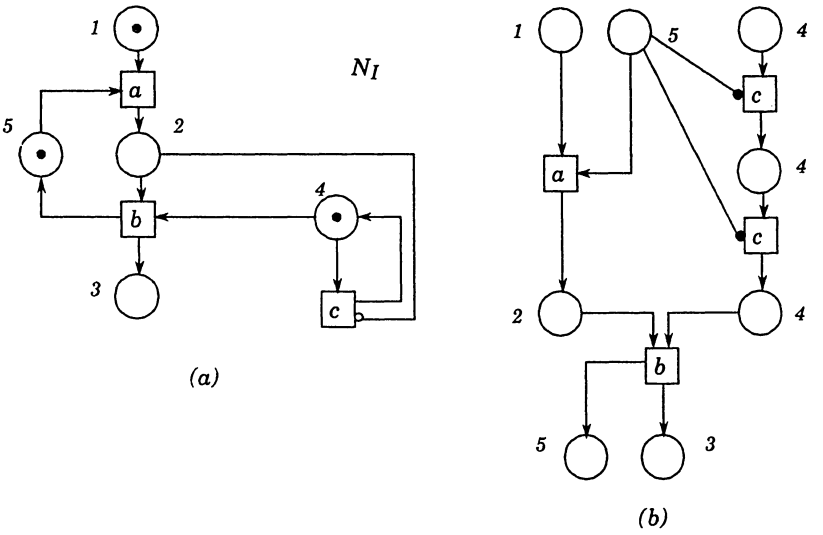


Figure 7

closure of the auxiliary relation in the construction of the poset for an ordinary occurrence net. For the occurrence net of Figure 7(b), the resulting $\rightarrow$ and $\nearrow$ are shown in Figure 8(c).

Final Comments

Our main goal was to show that in order to cope properly with general concurrent behaviours one should not be restricted only to poset based structures. We also tried to show that

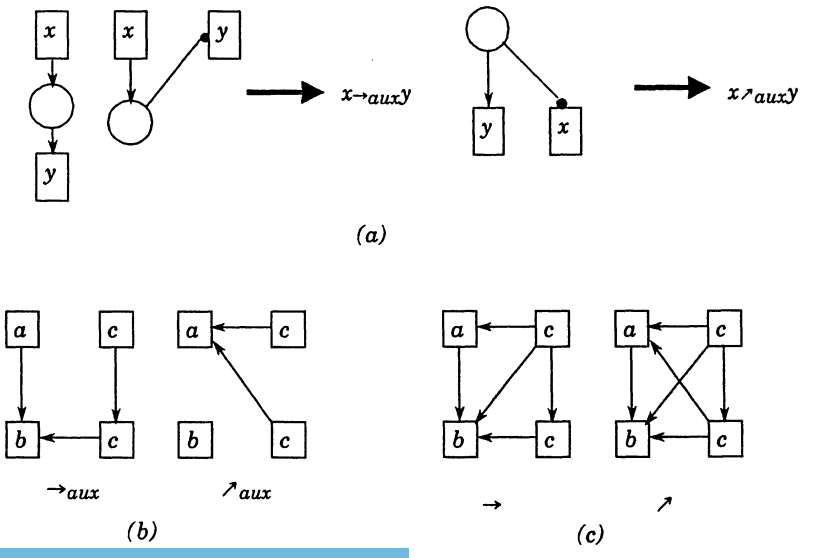


Figure 8

causality is only one of many possible invariants. The other invariants can be derived in a natural way when we use the bottom-top approach starting from the concept of observation as the primary notion. Although in this paper we defined observations as a certain kind of poset, concepts such as invariant, signature, S-closure, history, etc., are not associated with any specific definition of an observation. This paper presents a simplified version of a more general approach. In [JK90a] a general concept of 'report system' is defined, and all the concepts from Section 3 can be rendered in terms of 'reports' - generalising the notion of an observation. Consequently, the results presented here are just special cases of more general results obtained in [JK90a]. The extension of the definition of an observation (e.g. by adding relation representing uncertainty or by using the model similar to that of [AK85]) would not change the general structure of the approach introduced in this paper.

ACKNOWLEDGEMENT

The work of the first author was partly supported by a grant from NSERC No. OGP 0036539, while the work of the second author was supported by ESPRIT Basic Research Action 3148 (project DEMON).

REFERENCES

[AK85] Allen J.F., Kentz H.A., *A Model of Naive Temporal Reasoning*, In: J.R. Mobbs, R.C. Moore (Eds.), Formal Theories of the Commonsense World, Ablex 1985.

[BD85] Best E., Devillers R., *Concurrent Behaviour: Sequences, Processes and Programming Languages*, GMD-Studien Nr. 99, GMD, Bonn, 1985.

[BD87] Best E., Devillers R., *Sequential and Concurrent Behaviour in Petri Net Theory*, Theoretical Computer Science, 55 (1987), pp. 87-136.

[BK91] Best E., Koutny M., *Petri Net Semantics of Priority Systems*, to appear in Theoretical Computer Science.

[Fi70] Fishburn P.C., *Intransitive Indifference with Unequal Indifference Intervals*, J. Math. Psych. 7, 1970, pp. 144-19.

[Fi85] Fishburn P.C., *Interval Orders and Interval Graphs*, J. Wiley, 1985.

[Fr86] Fräise R., *Theory of Relations*, North Holland 1986.

[Ho85] Hoare C.A.R., *Communicating Sequential Processes*, Prentice-Hall, 1985.

[Ja87] Janicki R., *A Formal Semantics for Concurrent Systems with a Priority Relation*, Acta Informatica 24, 1987, pp.33-55.

[JK90] Janicki R., Koutny M., *Observing Concurrent Histories*, in: Real-Time Systems, Theory and Applications, H.M.S. Zedan (Ed.), Elsevier Science Publishers B.V. (North-Holland), 1990, pp 133-142.

[JK90a] Janicki R., Koutny M., *A Bottom-Top Approach to Concurrency Theory Part I: Observations, Invariants and Paradigms*, Technical Report No. 90-04, Dept. of Comp. Sci. and Syst., McMaster University, 1990.

[JL88] Janicki R., Lauer P.E., *On the Semantics of Priority Systems*, 17th Annual International Conference on Parallel Processing, Vol. 2, pp. 150-156, 1988, Pen. State Press.

[KP87] Katz S., Peled D., *Interleaving Set Temporal Logic*, 6th ACM Symposium on Principles of Distributed Computing, Vancouver 1984, pp. 178-190.

[La85] Lamport L., *What It Means for a Concurrent Program to Satisfy a Specification: Why No One Has Specified Priority*, 12th ACM Symposium on Principles of Programming Languages, New Orleans, Louisiana, 1985, pp. 78-83.

[La86] Lamport L., *On Interprocess Communication, Part I: Basic formalism, Part II: Algorithms*, Distributed Computing 1(1986), pp. 77-101.

[LH82] Lengauer C., Hehner E.C.R., *A Methodology for Programming with Concurrency: An Informal Presentation*, Science of Computer Programming 2 (1982), pp. 1-18.

[Ma86] Mazurkiewicz A., *Trace Theory*, Lecture Notes in Computer Science 225, Springer 1986, pp. 297-324.

[Mi80] Milner R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, vol. 92, Springer 1980.

[Mo76] Monk J.D., *Mathematical Logic*, Springer 1976.

[Pe81] Peterson J.L., *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.

[Pr86] Pratt V., *Modelling Concurrency with Partial Orders*, Int. Journal of Parallel Programming 15, 1 (1986), pp. 33-71.

[Re85] Reisig W., *Petri Nets*, Springer 1985.

[Sz30] Szpilrajn-Marczewski E., *Sur l'extension de l'ordre partial*, Fundamenta Mathematicae 16 (1930), pp. 386-389.

[Wn14] Wiener N., *A Contribution to the Theory of Relative Position*, Proc. Camb. Philos. Soc. 17 (1914), pp. 441-449.

[Wi82] Winskel G., *Event Structure Semantics for CCS and Related Language*, Lecture Notes in Computer Science 140, Springer 1982, pp. 561-567.

Acceptance Automata:
A Framework for Specifying and Verifying TCSP Parallel Systems

Luis M. Alonso
Departamento de Lenguajes y Sistemas Informáticos
Universidad del Pais Vasco
E-20080 San Sebastián
Spain.
email: alonso@gorria.if.ehu.es

Ricardo Peña[1]
Departament de Llenguatges i Sistemes Informatics
Universitat Politecnica de Catalunya
E-08028 Barcelona
Spain.
email: ricardo@lsi.upc.es

Abstract

Acceptance Automata, a particular case of labelled transition systems whose semantics is given in terms of the Failures Model, are presented. It is shown how parallel composition and hiding can be defined for them in a way consistent with the TCSP model. A notion of canonical automaton is presented. In the finite case it can be effectively computed and used for proving implementations correct. Finally, the TCSP concept of refinement is characterized in the acceptance automata domain.

1 Introduction

In a previous paper, see [PA 89], a technique for the specification, refinement and proof of correctness of parallel systems, was presented. The Failures Model [BHR 84][Hoa 85] was used as the mathematical model for Communicating Processes. Process behaviours were defined by means of a partial abstract type with certain characteristics. One of the advantages of this technique was that deductive methods developed in the general framework of algebraic specifications, could be used for proving properties concerning the specification of a process.

[1] This work has been partially supported by the ESPRIT-II Basic Research Action COMPASS (Working Group no. 3264).

We call a refinement the decomposition of a process into a set of lower level processes, possibly abstracting the synchronization events. Proving that a refinement exhibits the same behaviour as the original process was considerably more difficult: a few and simple syntactic transformations were applied to the refinement. Such transformations were based upon the algebraic laws satisfied by the mathematical model. A similar verification technique is used in [HoJi 85]. Since then the style of specification has evolved, and we feel that it is powerful and general enough to be useful in most problems. Nevertheless, proving refinements correct by manipulation of algebraic expressions does not seem practical.

In what follows the algebra of Acceptance Automata is introduced. It provides another formal framework for communicating processes, while preserving the semantics defined by the Failures Model. Acceptance automata could be used in the definition of processes. However, their main interest is that they are amenable for mechanical manipulation and that verification techniques similar to bisimulations [Par 81] can be used. As it will be shown, there exists most often, a family of acceptance automata defining the same process. Nevertheless it will always be possible to choose a canonical and unique representative of a given family. When considering finite-state systems, the canonical form might be effectively computed to mechanically prove a refinement correct.

The idea of using transition systems to define processes in the Failures Model goes back to [Jos 88]. There are some differences with the approach taken here. In order to model non-determinism, there it was necessary to use several transitions, with the same original state and labelling event, and with different final states. This amounts to say that there is a family of transition systems defining the same process. The relationships among those transition systems are not clear enough. Moreover, none of them is marked as the canonical form, and it doesn't seem possible doing such a choice. Finally, in [Jos 88] a number of simple rules are provided, by means of which a process can be shown to be a refinement of another. Those rules are sufficient conditions but not general enough.

Another related approach is that of Acceptance Trees as described in [Hen 85][Hen 88]. The differences are mainly two: on the one hand we are interested in the parallel composition and hiding. These operations are not defined for Acceptance trees. On the other hand, the approach is also related to the *readiness model* as described in [OH 86], in the sense that the states of our acceptance automata are labelled by sets of menus. However, our underlying model is the Failures one, the translation from sets of menus to sets of failures being immediate (see definition 2.7, below). Reference [OH 86] contains a short description of the Failures Model for those unfamiliar with it. A non-divergent process in this model is a set of pairs (s, X), where s is a trace of events and X is a set of events that the process may refuse. Our acceptance automata only describe non divergent processes.

Finally, a similar approach has been described in [AGS 90], where the so-called *CSP automata* are defined. They are similar to our acceptance automata, the main difference being the use of a *refusals mapping* in their definition. A notion of canonical form is also given. There are some minor differences: the work is restricted to finite-state systems and parallel composition and hiding are not defined. Besides that, the concept of refinement is not defined and different automata defining the same process are not compared.

2. Acceptance Automata

Definition 2.1 An *Acceptance Automaton* is given by a tuple $(\Lambda, \Sigma, \iota, M, \rightarrow)$ where:

- Λ is a non-empty alphabet of events,
- Σ is the set of states
- $\iota \in \Sigma$, is *the* initial state
- $M \subseteq \Sigma \times \mathcal{P}(\Lambda)$, is a menu relation, that will be denoted: $m \in M(\sigma)$
- $\rightarrow \subseteq \Sigma \times \Lambda \times \Sigma$, is a transition relation, that will be denoted: $\sigma \rightarrow_e \sigma'$

M and $\rightarrow$ shall be defined so that:

1) M is total in its domain, i.e.: $\forall \sigma \in \Sigma, \exists m \in \mathcal{P}(\Lambda).\ m \in M(\sigma)$
2) $\forall \sigma \in \Sigma, m \in M(\sigma), e \in \Lambda.\ e \in m \Rightarrow (\exists \sigma' \in \Sigma.\ \sigma \rightarrow_e \sigma')$
3) $\forall \sigma, \sigma' \in \Sigma, e \in \Lambda.\ \sigma \rightarrow_e \sigma' \Rightarrow (\exists m \in M(\sigma).\ e \in m)$

The role of the menu relation is to represent the internal non-determinism in the system, whilst the states represent the past of the system. We want to remark in this point, that the initial state is unique. Moreover, the transition systems defined in [Jos 88] can be translated into acceptance automata, provided that their initial states are unique. In particular, it is possible to define an acceptance automaton with several transitions, with the same original state and labelling event, and with different final states. The reason for which we allow this situation will be made clear later in the paper. Nevertheless, we will say in advance that acceptance automata used in the definition of processes do not exhibit this characteristic.

Two examples of acceptance automata follow: the first one is a finite state system, while the second one is a system with an infinite number of states:

$\Lambda = \{a, b, c\}$
$\Sigma = \{0, 1\}$
$\iota = 0$
$M = \{(0, \{a\}), (1, \{b\}), (1, \{c\})\}$
$\rightarrow = \{(0, a, 1), (1, b, 0), (1, c, 0)\}$

$\Lambda = \{in, out\}$
$\Sigma = \mathbb{N}$
$\iota = 0$
$M = \{(i, \{in\}) \mid i \in \mathbb{N}\} \cup \{(i, \{in, out\}) \mid i \in \mathbb{N}^+\}$
$\rightarrow = \{(i, in, i+1) \mid i \in \mathbb{N}\} \cup \{(i, out, i - 1) \mid i \in \mathbb{N}^+\}$

The following definitions introduce the concepts of traces and reachable states in this context. After that we will define the failures semantics of an acceptance automaton. In what follows we assume an acceptance automaton $\mathcal{A} = (\Lambda, \Sigma, \iota, M, \rightarrow)$.

Definition 2.2 The *reflexive-transitive closure* $\rightarrow \subseteq \Sigma \times \Lambda^* \times \Sigma$ of the transition relation is defined as the smallest relation satisfying:

- $\forall \sigma \in \Sigma.\ \sigma \rightarrow_{<>} \sigma$
- $\forall \sigma_1, \sigma_2, \sigma \in \Sigma, t \in \Lambda^*, e \in \Lambda.\ \sigma_1 \rightarrow_t \sigma \wedge \sigma \rightarrow_e \sigma_2 \Rightarrow \sigma_1 \rightarrow_{te} \sigma_2$

Definition 2.3 *traces*($\mathcal{A}$) is the set defined by: $\text{traces}(\mathcal{A}) = \{t \in \Lambda^*.\ \exists \sigma \in \Sigma.\ \iota \rightarrow_t \sigma\}$

Definition 2.4 The *reachability set of* $\mathcal{A}$ *after the trace t* is the set defined by:

$\mathcal{A}/t = \{\sigma \in \Sigma.\ \iota \rightarrow_t \sigma\}$

Definition 2.5 The *reachability set of* $\mathcal{A}$ is the set defined by:

$\text{reach}(\mathcal{A}) = \cup_{t \in \text{traces}(\mathcal{A})} \mathcal{A}/t$

Definition 2.6 The *set of next possible events*, for every state $\sigma \in \Sigma$, is the set defined by:

$\text{next}(\sigma) = \{e \in \Lambda.\ \exists \sigma' \in \Sigma.\ \sigma \rightarrow_e \sigma'\}$

Definition 2.7 The *failures semantics* of $\mathcal{A}$ is given by a relation $\mathcal{F}(\mathcal{A}) \subseteq \Lambda^* \times \mathcal{P}(\Lambda)$ defined as the smallest relation satisfying:

1) $\forall t \in \text{traces}(\mathcal{A}), \sigma \in \mathcal{A}/t, m \in M(\sigma).\ (t, \neg m) \in \mathcal{F}(\mathcal{A})$
2) $\forall t \in \text{traces}(\mathcal{A}), F_1, F_2 \in \mathcal{P}(\Lambda).\ (t, F_1) \in \mathcal{F}(\mathcal{A}) \wedge F_2 \subseteq F_1 \Rightarrow (t, F_2) \in \mathcal{F}(\mathcal{A})$

Lemma 2.8 Given an acceptance automaton $\mathcal{A} = (\Lambda, \Sigma, \iota, M, \rightarrow)$, the triple $(\Lambda, \mathcal{F}(\mathcal{A}), \emptyset)$ defines a non-divergent process in the failures model. In what follows, we simply use $\mathcal{F}(\mathcal{A})$ to denote this process.

Proof We only have to show that the following conditions hold [Hoa 85]:

- $(<>, \emptyset) \in \mathcal{F}(\mathcal{A})$
  this is trivial from condition 1 in definition 2.7
- $(st, X) \in \mathcal{F}(\mathcal{A}) \Rightarrow (s, \emptyset) \in \mathcal{F}(\mathcal{A})$
  this comes from the fact that traces($\mathcal{A}$) is prefix closed, hence if $(st, X) \in \mathcal{F}(\mathcal{A})$ then $s \in \text{traces}(\mathcal{A})$ and so $(s, \emptyset) \in \mathcal{F}(\mathcal{A})$
- $(s, Y) \in \mathcal{F}(\mathcal{A}) \wedge X \subseteq Y \Rightarrow (s, X) \in \mathcal{F}(\mathcal{A})$
  this is trivial from condition 2 in definition 2.7,
- $(s, X) \in \mathcal{F}(\mathcal{A}) \wedge x \in \Lambda \Rightarrow (s, X \cup \{x\}) \in \mathcal{F}(\mathcal{A}) \vee (sx, \emptyset) \in \mathcal{F}(\mathcal{A})$
  this comes from the fact that, s being a trace of $\mathcal{A}$, there exists state $\sigma \in \mathcal{A}/s$, and from the fact that, (s, X) being a failure, there exists a menu $m \in M(\sigma)$ such that $X \subseteq \neg m$. Hence, from definition 2.7, if $x \notin m$, then $(s, X \cup \{x\}) \in \mathcal{F}(\mathcal{A})$. Otherwise, if $x \in m$

then from condition 2 in definition 2.1 there exists a transition $\sigma \rightarrow_x \sigma'$ so sx is a trace of $\mathcal{A}$ and $(sx, \emptyset) \in \mathcal{F}(\mathcal{A})$. ◊

This lemma implies that divergent processes can not be modelled by means of acceptance automata. We have taken this decission because we find no practical interest in specifying processes which are allowed to engage in an infinite sequence of internal actions.

The following definition introduces a notion of equivalence between acceptance automata given in semantic terms. Later on, this concept will be characterized in terms closely related to bisimulations defined between transition systems.

Definition 2.9 Given acceptance automata $\mathcal{A}_1$ and $\mathcal{A}_2$, they are said to be *observationally equivalent*, denoted by $\mathcal{A}_1 \approx \mathcal{A}_2$, if $\mathcal{F}(\mathcal{A}_1) = \mathcal{F}(\mathcal{A}_2)$, i.e. if they have the same failures semantics.

See in the following example two equivalent acceptance automata:

$\Lambda = \{tick\}$
$\Sigma = \{"on"\}$
$\iota = "on"$
$M = \{("on", \{tick\})\}$
$\rightarrow = \{("on", tick, "on")\}$

$\Lambda = \{tick\}$
$\Sigma = \mathbb{N}$
$\iota = 0$
$M = \{(i, \{tick\}) \mid i \in \mathbb{N}\}$
$\rightarrow = \{(i, tick, i+1) \mid i \in \mathbb{N}\}$

Up to this moment, the basic notions related with acceptance automata have been established. We proceed now to define the basic operations over acceptance automata. These operations shall correspond to the parallel composition and hiding defined in the Failures Model. None of these operators is defined for acceptance trees defined in [Hen 85], [Hen 88]. Whilst the parallel composition of acceptance trees seems to be rather simple, we think that the definition of the hiding operator will not be so simple if we try to do it strictly in the acceptance tree domain.

Definition 2.10 Given acceptance automata $\mathcal{A}_1 = (\Lambda \cup \Lambda_1, \Sigma_1, \iota_1, M_1, \rightarrow_1)$ and $\mathcal{A}_2 = (\Lambda \cup \Lambda_2, \Sigma_2, \iota_2, M_2, \rightarrow_2)$, with $\Lambda, \Lambda_1, \Lambda_2$ pairwise disjoint, the *parallel composition* of $\mathcal{A}_1$ and $\mathcal{A}_2$, denoted by $\mathcal{A}_1 \parallel \mathcal{A}_2 = (\Lambda_{\parallel}, \Sigma, \iota, M, \rightarrow)$, has Λ as synchronization alphabet and is defined as follows:

1) $\Lambda_{\parallel} = \Lambda \cup \Lambda_1 \cup \Lambda_2$

2) $\Sigma = \Sigma_1 \times \Sigma_2$,

3) $\iota = (\iota_1, \iota_2)$,

4) $M \subseteq \Sigma \times \mathcal{P}(\Lambda)$, is the smallest relation satisfying:

$\forall\, m_1 \in M_1(\sigma_1),\ m_2 \in M_2(\sigma_2)$:

$(m_1 \cap m_2) \cup (\Lambda_1 \cap m_1) \cup (\Lambda_2 \cap m_2) \in M((\sigma_1, \sigma_2))$

5) $\rightarrow \subseteq \Sigma \times \Lambda_{\|} \times \Sigma$, is the smallest relation satisfying:

$\forall\, e \in \Lambda,\ \sigma_1 \rightarrow_{1_e} \sigma_2,\ \mu_1 \rightarrow_{2_e} \mu_2.\ (\sigma_1, \mu_1) \rightarrow_e (\sigma_2, \mu_2)$

$\forall\, e \in \Lambda_1,\ \sigma_1 \rightarrow_{1_e} \sigma_2,\ \mu \in \Sigma_2.\ (\sigma_1, \mu) \rightarrow_e (\sigma_2, \mu)$

$\forall\, e \in \Lambda_2,\ \sigma \in \Sigma_1,\ \mu_1 \rightarrow_{2_e} \mu_2.\ (\sigma, \mu_1) \rightarrow_e (\sigma, \mu_2)$

Fact 2.11 The parallel composition of acceptance automata $\mathcal{A}_1$ and $\mathcal{A}_2$, is another acceptance automaton $\mathcal{A}$.

Fact 2.12 Observe that the following predicate holds for acceptance automata $\mathcal{A}_1$ and $\mathcal{A}_2$, and its parallel composition $\mathcal{A}$:

$\forall\, t_1 \in traces(\mathcal{A}_1),\ \sigma_1 \in \mathcal{A}_1/t_1,\ t_2 \in traces(\mathcal{A}_2),\ \sigma_2 \in \mathcal{A}_2/t_2,\ t \in traces(\mathcal{A})$.

$t_1 = t\uparrow(\Lambda \cup \Lambda_1) \wedge t_2 = t\uparrow(\Lambda \cup \Lambda_2) \Leftrightarrow (\sigma_1, \sigma_2) \in \mathcal{A}/t$

where $t\uparrow\Lambda$ denotes the trace t restricted to the events in Λ. This fact is simple to prove by induction on the lenght of the traces.

Lemma 2.13 Given acceptance automata $\mathcal{A}_1 = (\Lambda \cup \Lambda_1, \Sigma_1, \iota_1, M_1, \rightarrow_1)$ and $\mathcal{A}_2 = (\Lambda \cup \Lambda_2, \Sigma_2, \iota_2, M_2, \rightarrow_2)$ its parallel composition satisfies:

$\mathcal{F}(\mathcal{A}_1 \| \mathcal{A}_2) = \mathcal{F}(\mathcal{A}_1) \| \mathcal{F}(\mathcal{A}_2)$

Proof Let $(t, X) \in \mathcal{F}(\mathcal{A}_1) \| \mathcal{F}(\mathcal{A}_2)$, then from the definition of parallel composition in the failures model, there exist $(t_1, X_1) \in \mathcal{F}(\mathcal{A}_1)$ and $(t_2, X_2) \in \mathcal{F}(\mathcal{A}_2)$ such that:

$t\uparrow\Lambda \cup \Lambda_1 = t_1,\ t\uparrow\Lambda \cup \Lambda_2 = t_2,\ X = X_1 \cup X_2$

Now, from definitions 2.1 and 2.7, there are $\sigma_1 \in \mathcal{A}_1/t_1$ (resp. $\sigma_2 \in \mathcal{A}_2/t_2$) and menu $m_1 \in M_1(\sigma_1)$ (resp. $m_2 \in M_2(\sigma_2)$) such that:

$X_1 \subseteq (\Lambda \cup \Lambda_1) - m_1$ (resp. $X_2 \subseteq (\Lambda \cup \Lambda_2) - m_2$)

Thus, from definition 2.10 and fact 2.15:

$(\sigma_1, \sigma_2) \in (\mathcal{A}_1 \| \mathcal{A}_2)/t$

and

$m = (m_1 \cap m_2) \cup (\Lambda_1 \cap m_1) \cup (\Lambda_2 \cap m_2) \in M((\sigma_1, \sigma_2))$.

It is easy to see that:

$X = X_1 \cup X_2 \subseteq (\Lambda \cup \Lambda_1 \cup \Lambda_2) - m$

so by definition 2.7, $(t, X) \in \mathcal{F}(\mathcal{A}_1 \| \mathcal{A}_2)$. The inverse holds for a similar reasoning. ◊

In what follows, we define the hiding operator on acceptance automata. As it was pointed above, divergent processes can not be modelled by means of acceptance automata. Since the hiding operator might cause divergence, even when applied to non-divergent processes, it is impossible to define a total operator over acceptance automata, while preserving the failures semantics. In order to overcome this difficulty, we first introduce the notion of divergence-free automaton, and then define hiding as a partial operator.

Definition 2.14 [Jos 88] Given an acceptance automaton $\mathcal{A} = (\Lambda, \Sigma, \iota, M, \rightarrow)$, and a set of events $\Lambda_h \subseteq \Lambda$, we say that $\mathcal{A}$ is *divergence free* with respect to Λ_h, if the following holds:

$$\forall\, s \in traces(\mathcal{A}).\ \neg\forall\, n \in \mathbb{N}.\ \exists\, t \in \Lambda_h^*.\ \#t > n \wedge st \in traces(\mathcal{A})$$

Lemma 2.15 The acceptance automaton $\mathcal{A} = (\Lambda \cup \Lambda_h, \Sigma, \iota, M, \rightarrow)$, is divergence free with respect to the set of events Λ_h iff there exists a well founded set $(\Omega, <)$ and a metric:

$$\omega: reach(\mathcal{A}) \rightarrow \Omega$$

satisfying:

$$\forall\, s \in \Lambda_h,\ \sigma \rightarrow_s \sigma'.\ \omega(\sigma') < \omega(\sigma)$$

This is a simple consequence of the definition of well-founded set.

Definition 2.16 Given an acceptance automaton $\mathcal{A} = (\Lambda \cup \Lambda_h, \Sigma, \iota, M, \rightarrow)$, which is divergence free with respect to the set of events Λ_h, $\mathcal{A}$ *after hiding* Λ_h, denoted by $\mathcal{A}\backslash\Lambda_h$, $= (\Lambda, \Sigma, \iota, M_h, \rightarrow_h)$, is defined as follows:

1) $M_h \subseteq \Sigma \times \mathcal{P}(\Lambda)$, is the smallest relation satisfying:

   $\forall\, \sigma \in \Sigma, m \in M(\sigma).\ (\Lambda_h \cap m) = \emptyset \Rightarrow m \in M_h(\sigma)$

   $\forall\, \sigma_1, \sigma_2 \in \Sigma, m_1 \in M(\sigma_1), m_2 \in M_h(\sigma_2), s \in (\Lambda_h \cap m_1).$

   $\sigma_1 \rightarrow_s \sigma_2 \Rightarrow (m_1 - \Lambda_h) \cup m_2, m_2 \in M_h(\sigma_1)$

2) $\rightarrow_h \subseteq \Sigma \times \Lambda \times \Sigma$, is the smallest relation satisfying:

   $\forall\, e \in \Lambda, \sigma_1 \rightarrow_e \sigma_2.\ \sigma_1 \rightarrow_{he} \sigma_2$

   $\forall\, e \in \Lambda, s \in \Lambda_h, \sigma_1 \rightarrow_s \sigma, \sigma \rightarrow_e \sigma_2.\ \sigma_1 \rightarrow_{he} \sigma_2$

Although this definition is recursive, $\mathcal{A}\backslash\Lambda_h$ is well-defined unless $\mathcal{A}$ is not divergence free with respect to the set of events Λ_h. Otherwise, if $\mathcal{A}$ is divergence free then there might exist paths of inestable states, the last element in such a path being a stable state. However, there will never exist closed paths made up of inestable states. Basically, this definition causes the copy of menus from the final state up to the initial state in such a path.

Fact 2.17 If $\mathcal{A}$ is an acceptance automaton, which is divergence free with respect to the set of events Λ_h,then $\mathcal{A}\backslash\Lambda_h$ is an acceptance automaton.

Fact 2.18 If $\mathcal{A} = (\Lambda \cup \Lambda_h, \Sigma, \iota, M, \rightarrow)$ is divergence free with respect to the set of events

Λ_h, then the following predicates hold:

$\forall\, t \in$ traces $(\mathcal{A})$. $t\uparrow\Lambda \in$ traces$(\mathcal{A}\backslash\Lambda_h)$

$\forall\, t \in$ traces $(\mathcal{A})$, $\sigma \in \mathcal{A}/t$, $m \in M(\sigma)$.

$$m \cap \Lambda_h = \emptyset \Rightarrow (\exists\, \sigma_h \in (\mathcal{A}\backslash\Lambda_h)/(t\uparrow\Lambda).\ m \in M_h(\sigma_h))$$

This is simple to prove by induction on the lenght of t.

Lemma 2.19 Given an acceptance automaton $\mathcal{A} = (\Lambda \cup \Lambda_h, \Sigma, \iota, M, \rightarrow)$, which is divergence free with respect to the set of events Λ_h: $\mathcal{F}(\mathcal{A}\backslash\Lambda_h) = \mathcal{F}(\mathcal{A})\backslash\Lambda_h$

Proof Let $(t_h, X_h) \in \mathcal{F}(\mathcal{A})\backslash\Lambda_h$, then, from the definition of the hiding operator in the failures model, there exists $(t, X) \in \mathcal{F}(\mathcal{A})$, such that:

$t\uparrow\Lambda = t_h$, $X = X_h \cup \Lambda_h$

Now, from definitions 2.1 and 2.7, there are $\sigma \in \mathcal{A}/t$ and menu $m \in M(\sigma)$ such that:

$X = X_h \cup \Lambda_h \subseteq \Lambda \cup \Lambda_h - m$

Thus, $m \cap \Lambda_h = \emptyset$. Then from fact 2.18 there exists a state $\sigma_h \in (\mathcal{A}\backslash\Lambda_h)/t_h$, such that $m \in M_h(\sigma_h)$. So by definition 2.7, $(t_h, X_h) \in \mathcal{F}(\mathcal{A}\backslash\Lambda_h)$.

The inverse holds for a similar reasoning.◊

3. Standard Acceptance Automata

In the previous section, the basic notions and operations concerning acceptance automata have been introduced. It is important to remark that acceptance automata, as defined in 2.1, might exhibit some non-desirable features. For instance, they can include non-reachable states. As it will be discussed below, this was necessary in order to define the most interesting operations: parallel composition and hiding. In this section we select a particular class of acceptance automata, the so-called standard acceptance automata, that will serve as the basis both for defining and comparing processes, possibly in a mechanical way. Through this section we assume an acceptance automaton $\mathcal{A} = (\Lambda, \Sigma, \iota, M, \rightarrow)$.

Definition 3.1 $\mathcal{A}$ is said to be *junk free* if:

$\forall\, \sigma \in \Sigma.\ \exists\, t \in$ traces$(\mathcal{A})$. $\sigma \in \mathcal{A}/t$

Fact 3.2 We can always define the corresponding junk free acceptance automaton junk-free$(\mathcal{A}) = (\Lambda, \text{reach}(\mathcal{A}), \iota, M^*, \rightarrow^*)$, with the same failures semantics as $\mathcal{A}$. M^* is defined as the smallest relation satisfying:

$\forall\, \sigma \in \text{reach}(\mathcal{A}), m \in M(\sigma).\ m \in M^*(\sigma)$

Similarly, $\rightarrow^*$ is defined as the smallest relation satisfying

$\forall\, \sigma, \sigma' \in \text{reach}(\mathcal{A}), e \in \Lambda.\ \sigma \rightarrow_e \sigma' \Rightarrow \sigma \rightarrow^*_e \sigma'$

Obviously, since there is no interest in considering non-reachable states, we could try to

restrict the definition of acceptance automata to those tuples with this property. Nevertheless, it must be observed that the parallel composition of junk-free automata does not define, in general, another junk-free automata. The same holds for the hiding operator. Thus, if we want to define those operations over acceptance automata, it must be posible for them to have non-reachable states. This is of no concern because, as it has been pointed above, it is always possible to define the equivalent junk-free automata.

Definition 3.3 $\mathcal{A}$ is said to be *unambiguous* if:

$\forall\, t \in \text{traces}(\mathcal{A}).\ |\ \mathcal{A}/t\ | = 1$

We will use the term *ambiguous* when referring to an acceptance automaton that is not unambiguous.

The following examples define an ambiguous and an unambiguous automaton, both with the same failures semantics:

$\Lambda = \{a, b\}$
$\Sigma = \{0, 1, 2\}$
$\iota = 0$
$M = \{(0, \{a\}), (1, \{b\}), (2, \emptyset)\}$
$\rightarrow = \{(0, a, 1), (0, a, 2), (1, b, 2)\}$

$\Lambda = \{a, b\}$
$\Sigma = \{0, 1, 2\}$
$\iota = 0$
$M = \{(0, \{a\}), (1, \emptyset), (1, \{b\}), (2, \emptyset)\}$
$\rightarrow = \{(0, a, 1), (1, b, 2)$

As it is shown in this example, the process defined by an unambiguous acceptance automaton is not in general deterministic. Moreover, the process defined by an ambiguous acceptance automaton could be deterministic. This motivates our next definition:

Definition 3.4 $\mathcal{A}$ is said to be a *deterministic automaton* if it is unambiguous and:

$\forall\, t \in \text{traces}(\mathcal{A}), \sigma \in \mathcal{A}/t.\ |\ M(\sigma)\ | = 1$

Fact 3.5 The parallel composition of unambiguous automata defines another unambiguous automata. This is trivial from the definition of the parallel composition, and in particular, from the definition of its transition relation.

Observe that the hiding operation applied to an unambiguous acceptance automaton does not define, in general, another unambiguous acceptance automaton. The menu relation was included in the definition of acceptance automata to model the non-deterministic behaviour of concurrent systems. The intuitive idea is that non-determinism should be represented by the existence of several menus in a given state. However, the use of ambiguous acceptance automata allows the modelling of non-deterministic behaviour, even when there is a unique menu associated with every reachable state. According to this, it might seem appropiate to

restrict the definition of acceptance automata to those which are unambiguous. This is not possible if we want to define the hiding operator over acceptance automata. In what follows we show how to deal with this situation defining the equivalent unambiguous acceptance automaton.

Definition 3.6 The *unambiguous form* of $\mathcal{A}$ is the acceptance automaton $\mathcal{A}_{uf} = (\Lambda, \Sigma_{uf}, \iota_{uf}, M_{uf}, \rightarrow_{uf})$ defined as follows:

- $\Sigma_{uf} = traces(\mathcal{A})/\approx$,
  where $\approx \subseteq traces(\mathcal{A}) \times traces(\mathcal{A})$, is defined as the smallest relation satisfying:
  $\forall\, t_1, t_2 \in traces(\mathcal{A}).\ \mathcal{A}/t_1 = \mathcal{A}/t_2 \Rightarrow t_1 \approx t_2$
- $\iota_{uf} = [<>]$
- $M_{uf} \subseteq \Sigma_{uf} \times \mathcal{P}(\Lambda)$, is the smallest relation satisfying:
  $\forall\, t \in traces(\mathcal{A}), \sigma \in \mathcal{A}/t, m \in M(\sigma).\ m \in M_{uf}([t])$
- $\rightarrow_{uf} \subseteq \Sigma_{uf} \times \Lambda \times \Sigma_{uf}$, is the smallest relation satisfying:
  $\forall\, t \in traces(\mathcal{A}), e \in \Lambda.\ te \in traces(\mathcal{A}) \Rightarrow [t] \rightarrow_{uf_e} [te]$

Lemma 3.7 Given an ambiguous acceptance automaton $\mathcal{A}$ and its unambiguous form $\mathcal{A}_{uf}$: $\mathcal{A} \approx \mathcal{A}_{uf}$

For finite acceptance automata, there is an effective procedure to build its unambiguous form.

Definition 3.8 $\mathcal{A}$ is said to be *saturated* if:

$\forall\, \sigma \in \Sigma, m \in \mathcal{P}(\Lambda), m' \in M(\sigma).\ m' \subseteq m \subseteq next(\sigma) \Rightarrow m \in M(\sigma)$

Note that the parallel composition and hiding of saturated automata define another saturated automata. Moreover, given a non-saturated acceptance automaton, the derivation of its saturated form is trivial and it is easy to prove that its failures semantics is the same. This fact is directly reflected in the TCSP laws.

Definition 3.9 Given an acceptance automaton $\mathcal{A}$, we say that it is in *standard form* if it is junk-free, unambiguous and saturated.

This is the most important concept in this section. It must be observed that for finite state acceptance automata we can build mechanically its standard form. Moreover, the result of this transformation process is unique, regardless of the definition of the original automaton. The main interest of this reduction procedure will be made clear in the next section.

4. Minimal Automaton

Given a non divergent TCSP process $\mathbb{P}$, there are many standard automata exhibiting its

behaviour. The purpose of this section is to explore the category SAut ($\mathbb{P}$) of all the standard automata that have $\mathbb{P}$ as semantics. The morphisms will be a restricted version of the classical notion of bisimulation that we call a *simulation*. We show that in this category there exist both initial and final objects. The initial one or *maximal automaton* is the automaton with the greatest number of states corresponding to the TCSP normal form [Nic 85]. The final one or *minimal automaton* is the automaton with the least number of states. For practical purposes this is the most interesting one. For finite acceptance automata there is an efective procedure to compute their final versions.

Definition 4.1 Given a non-divergent process in the Failures Model $\mathbb{P} = (\Lambda, \mathfrak{F}, \mathfrak{D})$, with $\mathfrak{F} \subseteq \Lambda^* \times \mathcal{P}(\Lambda)$ and $\mathfrak{D} = \emptyset$, its *maximal acceptance automaton* $\mathcal{A}_1(\mathbb{P}) = (\Lambda, \Sigma, \iota, M, \rightarrow)$, is defined as follows:

- Λ is the alphabet of $\mathbb{P}$,
- $\Sigma = \{t \in \Lambda^*, (t, \emptyset) \in \mathbb{P}\}$,
- ι is the empty trace $\langle\rangle$
- $M \subseteq \Sigma \times \mathcal{P}(\Lambda)$, is the smallest relation satisfying:
  $\forall (\sigma, F) \in \mathbb{P}. (\forall (\sigma, F') \in \mathbb{P}. F \not\subseteq F') \Rightarrow \neg F \in M(\sigma)$
  $\forall \sigma \in \Sigma, m \in \mathcal{P}(\Lambda), m' \in M(\sigma). m' \subseteq m \subseteq next(\sigma) \Rightarrow m \in M(\sigma)$
- $\rightarrow \subseteq \Sigma \times \Lambda \times \Sigma$, is the smallest relation satisfying:
  $\forall (\sigma, \emptyset), (\sigma e, \emptyset) \in \mathbb{P}. \sigma \rightarrow_e \sigma e$

The first line in the menu relation introduces as menus the complements of the maximal refusal sets after a given trace. The second line is the convex closure of the previous menus [HoJi 85]. It is interesting to note that the menu relation conveys more information than an alternative *refusal relation* as used for instance in [AGS 90]. From the first one it is immediate to obtain the second one (see definition 2.7). To do the opposite we need next(σ), the set of possible events after a trace σ, i.e. we need also to look at transition relation, i.e. at the whole failures set.

It is obvious to see that this definition satisfies the constraints of an acceptance automaton, in particular those of a standard one, and that its failures semantics coincides exactly with the original process $\mathbb{P}$. In fact, this maximal automaton is exactly the TCSP term in normal form that can be constructed from the failures set as in [Nic 85].

Now we proceed to the construction of the category: given the non-divergent TCSP process $\mathbb{P}$, let SAut ($\mathbb{P}$) be the class of all the standard acceptance automata that have $\mathbb{P}$ as semantics.

Definition 4.2 Given $\mathcal{A}_1 = (\Lambda, \Sigma_1, \iota_1, M_1, \rightarrow_1)$, $\mathcal{A}_2 = (\Lambda, \Sigma_2, \iota_2, M_2, \rightarrow_2)$ standard automata, we say that a mapping f: $\Sigma_1 \rightarrow \Sigma_2$ is a *morphism* or a *simulation* from $\mathcal{A}_1$ to $\mathcal{A}_2$ if it satisfies the following conditions:

1) $f(\iota_1) = \iota_2$
2) $\forall \sigma \in \Sigma_1. M_1(\sigma) = M_2(f(\sigma))$

3) $\forall\ \sigma \rightarrow_{1_c} \sigma'.\ f(\sigma) \rightarrow_{2_c} f(\sigma')$

A simulation is a particular case of the notion of bisimulation that can be adapted to acceptance automata in the following way:

Definition 4.3 Given standard acceptance automata $\mathcal{A}_1 = (\Lambda, \Sigma_1, \iota_1, M_1, \rightarrow_1)$ and $\mathcal{A}_2 = (\Lambda, \Sigma_2, \iota_2, M_2, \rightarrow_2)$ a *bisimulation of* $\mathcal{A}_1$ *and* $\mathcal{A}_2$ is a relation $\mathfrak{B} \subseteq \Sigma_1 \times \Sigma_2$, satisfying:

1) $(\iota_1, \iota_2) \in \mathfrak{B}$
2) $\forall\ (\sigma_1, \sigma_2) \in \mathfrak{B}.\ M_1(\sigma_1) = M_2(\sigma_2)$
3) $\forall\ (\sigma_1, \sigma_2) \in \mathfrak{B}, e \in \Lambda, \sigma_1 \rightarrow_{1_e} \sigma, \sigma_2 \rightarrow_{2_e} \sigma'.\ (\sigma, \sigma') \in \mathfrak{B}$

The difference between a general relation and a mapping is that the last one will allow us to go from automata with more states to automata with less states.

Lemma 4.4 Given standard acceptance automata $\mathcal{A}_1$ and $\mathcal{A}_2$, $\mathcal{A}_1 \approx \mathcal{A}_2$ iff there exists a bisimulation between $\mathcal{A}_1$ and $\mathcal{A}_2$.

Proof Let us first see the following fact :
If $\mathfrak{B}$ is a bisimulation of $\mathcal{A}_1$ and $\mathcal{A}_2$ then the following predicate holds:

$\forall\ t \in \text{traces}(\mathcal{A}_1).\ (\mathcal{A}_1/t, \mathcal{A}_2/t) \in \mathfrak{B}$

Where $\mathcal{A}_1/t$, $\mathcal{A}_2/t$ denote the unique state reached after the trace t. This is simple to prove by induction on the lenght of t. From this fact and definition 2.7, it follows that if there exists a bisimulation of $\mathcal{A}_1$ and $\mathcal{A}_2$ then $\mathcal{A}_1 \approx \mathcal{A}_2$. The opposite is also simple to prove. Given that $\mathcal{A}_1 \approx \mathcal{A}_2$, let us define $\mathfrak{B} \subseteq \Sigma_1 \times \Sigma_2$ to be the smallest relation such that:

$\forall\ t \in \text{traces}(\mathcal{A}_1), \sigma_1 \in \mathcal{A}_1/t, \sigma_2 \in \mathcal{A}_2/t \Rightarrow (\sigma_1, \sigma_2) \in \mathfrak{B}$

$\mathfrak{B}$ trivially satisfies conditions 1 and 3 of definition 4.3. Then from $\mathcal{A}_1 \approx \mathcal{A}_2$ and the fact that they are standard, it comes condition 2. So $\mathfrak{B}$ is a bisimulation. $\Diamond$

It its easy to prove that the composition of morphisms gives another morphism and that it is associative with the identity morphism being the identity mapping. So SAut ($\mathbb{P}$) together with all the morphisms defined between them turns out to be a category.

In SAut ($\mathbb{P}$), an isomorphism will be a bijective morphism and two isomorphic automata will be in fact equal up to renaming of states. As usual, there will be a morphism from the initial object (if it exists) to any other object in the category and from any object to the final one (if it exists). The initial and final objects (if they exist) are unique up to isomorphism. We shall see that these objects exist in SAut ($\mathbb{P}$).

Lemma 4.5 Given $\mathbb{P}$, the maximal acceptance automaton $\mathcal{A}_I(\mathbb{P})$ is initial in SAut ($\mathbb{P}$).

Proof: Given any acceptance automaton $\mathcal{A} \in SAut(\mathbb{P})$ the mapping f:traces($\mathcal{A}_I(\mathbb{P})$) $\rightarrow \Sigma$, defined by $f(t) = \mathcal{A}/t$, where $\mathcal{A}/t$ denotes the unique state reached after the trace t, is a morphism and it is unique. This follows from lemma 4. 4. ◊

Now we proceed to the proof of existence and construction of the final automaton $\mathcal{A}_F(\mathbb{P})$ of SAut($\mathbb{P}$). First, we need the concept of *congruence* and of *quotient automaton* by a congruence.

Definition 4.6 Given an standard acceptance automaton $\mathcal{A} = (\Lambda, \Sigma, \iota, M, \rightarrow)$, a *congruence* in $\mathcal{A}$ is an equivalence relation $\equiv \subseteq \Sigma \times \Sigma$ satisfying the following properties:

1) $\forall \sigma_1, \sigma_2 \in \Sigma. \sigma_1 \equiv \sigma_2 \Rightarrow M(\sigma_1) = M(\sigma_2)$
2) $\forall \sigma_1, \sigma_2 \in \Sigma, e \in \Lambda. \sigma_1 \equiv \sigma_2 \wedge \sigma_1 \rightarrow_e \sigma \wedge \sigma_2 \rightarrow_e \mu \Rightarrow \sigma \equiv \mu$

Definition 4.7 Given an standard acceptance automaton $\mathcal{A} = (\Lambda, \Sigma, \iota, M, \rightarrow)$ and a congruence $\equiv$ in $\mathcal{A}$ the *quotient acceptance automaton* of $\mathcal{A}$ by $\equiv$, denoted by $\mathcal{A}/\equiv = (\Lambda, \Sigma_{\equiv}, \iota_{\equiv}, M_{\equiv}, \rightarrow_{\equiv})$ is defined as follows:

1) $\Sigma_{\equiv} = \Sigma/Q$, is the quotient set of Σ by $\equiv$
2) $\iota_{\equiv} = [\iota]$
3) $M_{\equiv}([\sigma]) = M(\sigma)$
4) $[\sigma] \rightarrow_e [\sigma']$ iff $\sigma \rightarrow_e \sigma'$

It is immediate to show that this definition is independent of the representative σ chosen for the class $[\sigma]$. There is a strong connection between congruences and simulations as the following lemmas show:

Lemma 4.8 Given an acceptance automata $\mathcal{A} = (\Lambda, \Sigma, \iota, M, \rightarrow)$ and a congruence $\equiv$ in $\mathcal{A}$, the mapping f: $\mathcal{A} \rightarrow \mathcal{A}/\equiv$ defined by $f(\sigma) = [\sigma]$ is a simulation.

Proof Properties 1, 2, and 3 of definition 4.2 are a direct translation of properties 2, 3, and 4 of definition 4.7. ◊

Lemma 4.9 Given acceptance automata $\mathcal{A}_1 = (\Lambda, \Sigma_1, \iota_1, M_1, \rightarrow_1)$ and $\mathcal{A}_2 = (\Lambda, \Sigma_2, \iota_2, M_2, \rightarrow_2)$, with $\mathcal{A}_1, \mathcal{A}_2 \in SAut(\mathbb{P})$, if there is a simulation f: $\mathcal{A}_1 \rightarrow \mathcal{A}_2$, then the equivalence relation in Σ_1 defined by:

$\sigma_1 \equiv \sigma_2$ iff $f(\sigma_1) = f(\sigma_2)$

is a congruence in $\mathcal{A}_1$ and $\mathcal{A}_1/\equiv$ is isomorphic to $\mathcal{A}_2$.

Proof

1) $\equiv$ is a congruence:

condition 1 of definition 4.6 follows directly from condition 2 of definition 4.2. With respect to condition 2 of definition 4.6, let us assume:

$\sigma_1 \rightarrow_{1_e} \sigma_1'$, $\sigma_2 \rightarrow_{2_e} \sigma_2'$ and $f(\sigma_1) = f(\sigma_2)$

by being f a simulation we have

$f(\sigma_1) \rightarrow_{2_e} f(\sigma_2')$ and $f(\sigma_2) \rightarrow_{2_e} f(\sigma_2')$

then, as $\mathcal{A}_2$ is standard $f(\sigma_1') = f(\sigma_2')$ so $\sigma_1' \equiv \sigma'$

2) First observe that:

$\forall\, \sigma_2 \in \Sigma_2.\ \exists\, t \in \text{traces}(\mathcal{A}_2).\ \{\sigma_2\} = \mathcal{A}_2/t$

and as f is a morphism: $f(\sigma_1) = \sigma_2$, where $\{\sigma_1\} = \mathcal{A}_1/t$. So f is surjective.

Then, the mapping $f^*: \mathcal{A}_1/\equiv \rightarrow \mathcal{A}_2$ defined by:

$f^*([\sigma]) = f(\sigma)$

is a biyective morphism and $\mathcal{A}_1/\equiv$ is isomorphic to $\mathcal{A}_2$. ◊

As a consequence, doing the quotient of an automaton by a congruence, preserves the behaviour.

Definition 4.10 Given an acceptance automata $\mathcal{A} = (\Lambda, \Sigma, \iota, M, \rightarrow)$ and two congruences $\equiv_1, \equiv_2$ in $\mathcal{A}$, the *sum* $\equiv_1 + \equiv_2 \subseteq \Sigma \times \Sigma$ is defined as the transitive closure of $\equiv_1 \cup \equiv_2$.

Fact 4.11 Given an acceptance automata $\mathcal{A} = (\Lambda, \Sigma, \iota, M, \rightarrow)$ and two congruences $\equiv_1, \equiv_2$ in $\mathcal{A}$, the *sum* $\equiv_1 + \equiv_2$ is another congruence on $\mathcal{A}$.

Since the identity mapping on states is a congruence, and since the sum and intersection of two congruences is another congruence, the set $(\text{Cong}(\mathcal{A}), \subseteq)$, with $\text{Cong}(\mathcal{A})$ denoting the family of congruences in $\mathcal{A}$, turns out to be a complete lattice. Let us now denote by $\equiv_F(\mathcal{A})$ the maximum congruence on $\mathcal{A}$ constructed by suming up all the congruences on $\mathcal{A}$.

Definition 4.12 Given a non-divergent process TCSP process $\mathbb{P}$ its *minimal acceptance automaton* $\mathcal{A}_F(\mathbb{P})$ is defined by $\mathcal{A}_I(\mathbb{P})/\equiv_F(\mathcal{A}_I(\mathbb{P}))$. Given an acceptance automaton $\mathcal{A}$, we will use the term *normal form*, denoted by $\mathcal{A}\downarrow$, when referring to the corresponding minimal automaton.

Lemma 4.13 $\mathcal{A}_F(\mathbb{P})$ is final in $\text{SAut}(\mathbb{P})$.

Proof We need to show that for all $\mathcal{A}$ in $\text{SAut}(\mathbb{P})$, there is a unique morphism $f: \mathcal{A} \rightarrow \mathcal{A}_F(\mathbb{P})$. As $\mathcal{A}_I(\mathbb{P})$ is initial, there exists a unique morphism $f_I: \mathcal{A}_I(\mathbb{P}) \rightarrow \mathcal{A}$ and congruence $\equiv_I$ such that $\mathcal{A}_I(\mathbb{P})/\equiv_I$ is isomorphic to $\mathcal{A}$. Also there is a unique morphism $f_F: \mathcal{A}_I(\mathbb{P}) \rightarrow \mathcal{A}_F(\mathbb{P})$ induced by $\equiv_F(\mathcal{A}_I(\mathbb{P}))$. As this is the maximum congruence in $\text{Cong}(\mathcal{A}_I(\mathbb{P}))$, we have that $\equiv_I \subseteq \equiv_F(\mathcal{A}_I(\mathbb{P}))$ and the mapping $f: \mathcal{A} \rightarrow \mathcal{A}_F(\mathbb{P})$, defined by $f([t]_I) = [t]_F$ for any trace t, is well-defined. It is

straightforward to show that f is a morphism, and since $f_F = f^* f_I$, f must also be unique. ◊

Fact 4.14 If $\mathcal{A}(\mathbb{P})$ is finite, then the procedure for computing $\mathcal{A}_F(\mathbb{P})$ is algorithmic. The procedure will be an adapted version of the Moore algorithm [Woo 87].

This fact provides in many situations an effective mean to verify the correctness of parallel systems. The complete method, already advanced in [PA 89], will consist of the following steps:

- specify the system at the outermost level. Construct the acceptance automaton $\mathcal{A}_{sp}$ of the specification. Let us assume that it is finite
- implement the system as a parallel composition of automata $\mathcal{A}_1, .., \mathcal{A}_n$, with synchronization alphabet Λ
- use the definitions in this paper to compute the automaton $\mathcal{A}_{imp} = (\mathcal{A}_1 \| .. \| \mathcal{A}_n)\backslash\Lambda$
- verify that the normal forms of $\mathcal{A}_{sp}$ and $\mathcal{A}_{imp}$ are isomorphic, $\mathcal{A}_{sp}\downarrow = \mathcal{A}_{imp}\downarrow$

The normal form of an automaton with n states can be computed in time $O(n^3)$ by the adapted version of the Moore algorithm. The complete proof methos needs the explicit construction of the parallel composition $\mathcal{A}_1 \| \| \mathcal{A}_n$ with a worst case exponential in the number of states. If the refinements are done in small steps there is hope that this combinatorial explosion can be controlled. In any case the design of a large parallel system will always be done by refining one process at a time.

5. Refinements

Usually the implementation of a system is obtained by first composing in parallel a family of subsystems and then hiding those events corresponding to their internal activity. Asking the implementation to behave exactly like the specification amounts to say that their initial (respectively final) forms are isomorphic. Nevertheless this is a too strong requirement in most situations. For this reason we will adopt the notion of refinement as defined in the Failures Model, and provide an alternative characterization in terms of acceptance automata that will be amenable to mechanical verification.

Definition 5.1 Given acceptance automata $\mathcal{A}_{sp}$ and $\mathcal{A}_{imp}$, $\mathcal{A}_{imp}$ is said to be a *refinement* of $\mathcal{A}_{sp}$, denoted by $\mathcal{A}_{imp} \sqsubseteq \mathcal{A}_{sp}$, if $\mathcal{F}(\mathcal{A}_{imp}) \subseteq \mathcal{F}(\mathcal{A}_{sp})$.

Definition 5.2 Given an acceptance automaton $\mathcal{A} = (\Lambda, \Sigma, \iota, M, \rightarrow)$ it is said to be *deadlock free* if: $\forall \sigma \in \text{reach}(\mathcal{A}).\ \emptyset \notin M(\sigma)$

Lemma 5.3 Let $\mathcal{A}_{sp} = (\Lambda, \Sigma_{sp}, \iota_{sp}, M_{sp}, \rightarrow_{sp})$ and $\mathcal{A}_{imp} = (\Lambda, \Sigma_{imp}, \iota_{imp}, M_{imp}, \rightarrow_{imp})$ be initial acceptance automata:

$\mathcal{A}_{imp} \sqsubseteq \mathcal{A}_{sp}$ iff $\forall t \in \text{traces}(\mathcal{A}_{imp}).\ (t \in \text{traces}(\mathcal{A}_{sp}) \wedge M_{imp}(t) \subseteq M_{sp}(t))$

where $M_{imp}(t), M_{sp}(t) \subseteq \mathcal{P}(\Lambda)$, denote families of menus.

A similar fact might be stated by comparing the respective final forms. In this case, a means for relating states in the specification and implementation must be provided in the form of a relation over states. Most often, this relation will be a mapping even though this is not the general case.

Fact 5.4 Let $\mathcal{A}_{sp} = (\Lambda, \Sigma_{sp}, \iota_{sp}, M_{sp}, \rightarrow_{sp})$ and $\mathcal{A}_{imp} = (\Lambda, \Sigma_{imp}, \iota_{imp}, M_{imp}, \rightarrow_{imp})$ be final acceptance automata. $\mathcal{A}_{imp}$ is a refinement of $\mathcal{A}_{sp}$ if there exists a relation $\Phi \subseteq \Sigma_{imp} \times \Sigma_{sp}$ satisfying:

- $(\iota_{imp}, \iota_{sp}) \in \Phi$,
- $\forall\, (\sigma_{imp}, \sigma_{sp}) \in \Phi.\ M_{imp}(\sigma_{imp}) \subseteq M_{sp}(\sigma_{sp})$
- $\forall\, (\sigma_{imp}, \sigma_{sp}) \in \Phi,\ \sigma_{imp} \rightarrow_{imp_e} \mu_{imp},\ \sigma_{sp} \rightarrow_{sp_e} \mu_{sp}.\ (\mu_{imp}, \mu_{sp}) \in \Phi$

We shall use the term *abstraction relation* when referring to this relation.

Fact 5.5 Given acceptance automata $\mathcal{A}_{sp}$ and $\mathcal{A}_{imp}$, if $\mathcal{A}_{sp}$ is dead-lock free and $\mathcal{A}_{imp}$ is a refinement of $\mathcal{A}_{sp}$ then $\mathcal{A}_{imp}$ is dead-lock free.

This is a general fact concerning refinements and coming from the Failures Model. Besides that, fact 5.4 provides an effective procedure for proving correctness of refinements when dealing with finite state systems. Although this is the case with a great number of interesting problems, finite state systems are far from being the general situation. The following fact attempts to handle this problem.

Fact 5.6 Let $\mathcal{A}_{sp} = (\Lambda, \Sigma_{sp}, \iota_{sp}, M_{sp}, \rightarrow_{sp})$ and $\mathcal{A}_{imp} = (\Lambda \cup \Lambda_h, \Sigma_{imp}, \iota_{imp}, M_{imp}, \rightarrow_{imp})$ be standard acceptance automata. Provided that $\mathcal{A}_{imp}$ is divergence free w.r.t Λ_h, then $\mathcal{A}_{imp} \backslash \Lambda_h$ is a refinement of $\mathcal{A}_{sp}$ iff there is a relation $\Phi \subseteq \Sigma_{imp} \times \Sigma_{sp}$ satisfying:

- $(\iota_{imp}, \iota_{sp}) \in \Phi$,
- $\forall\, (\sigma_{imp}, \sigma_{sp}) \in \Phi,\ m \in M_{imp}(\sigma_{imp}).$
  $m \cap \Lambda_h = \emptyset \Rightarrow m \in M_{sp}(\sigma_{sp})$
  $m \cap \Lambda_h \neq \emptyset \Rightarrow m - \Lambda_h\ \mathrm{next}(\sigma_{sp})$
- $\forall\, (\sigma_{imp}, \sigma_{sp}) \in \Phi,\ \sigma_{imp} \rightarrow_{imp_e} \mu_{imp},\ \sigma_{sp} \rightarrow_{sp_e} \mu_{sp}.\ (\mu_{imp}, \mu_{sp}) \in \Phi$
- $\forall\, (\sigma_{imp}, \sigma_{sp}) \in \Phi,\ s \in \Lambda_h,\ \sigma_{imp} \rightarrow_{imp_s} \mu_{imp}.\ (\mu_{imp}, \sigma_{sp}) \in \Phi$

Observe that if $\mathcal{A}_{sp}$ and $\mathcal{A}_{imp}$ are in initial form, then this relation is in fact a mapping $\mathbb{I}$: $\mathrm{traces}(\mathcal{A}_{imp}) \rightarrow \mathrm{traces}(\mathcal{A}_{sp})$, defined by $\mathbb{I}(t) = t \uparrow \Lambda$.

References

[AGS 90] Autebert, J.M., Gabarro, J., Serna, M.J. "Finite memory CSP and CCS devices." Internal Rport, Departamento de Lenguajes y Sistemas Informáticos, Universidad Politécnica de Cataluña (1990).

[BHR 84] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W. "A Theory of Communicating Sequential Processes." *Journal of the ACM* **31** (1984) 560-599.

[Hen 85] Hennessy, M. "Acceptance Trees." *Journal of the ACM* **32** (1985) 896-928.

[Hen 88] Hennessy, M. *Algebraic Theory of Processes*. MIT Press (1988).

[Hoa 85] Hoare, C.A.R. *Communicating Sequential Processes*. Prentice-Hall (1985).

[HoJi 85] Hoare, C.A.R., Jifeng, H. "Algebraic Specification and Proof of Properties of Communicating Sequential Processes." *Tech. Rep*. PRG-**52** (1985) Oxford University Computing Laboratory.

[Jos 88] Josephs, M.B. "A state-based approach to communicating processes." *Distributed Computing* **3** (1988) 9 - 18.

[Nic 85] de Niccola, R. "Two Complete Axiom Systems for a Theory of Communicating Sequential Processes." *Information and Control* **64** (1985) 136-172.

[OH 86] Olderog, E.-R., Hoare, C.A.R. "Specification-Oriented Semantics for Communicating Processes." *Acta Informatica* **23** (1986) 9-66.

[Par 81] Park, D. "Concurrency and automata on infinite sequences." *Proc. 5th GI Conf. of Theoretical Computer Science*, LNCS 104 (1981) 245-251.

[PA 89] Peña, R., Alonso, L.M. "Specification and Verification of TCSP Systems by Means of Partial Abstract Data Types." In *Proceedings* TAPSOFT'89, LNCS 352 (1989) 328-344.

[Woo 87] Wood, D. *Theory of computation*. John-Wiley (1987).

Models for dynamically placed concurrent processes

J. FANCHON, D. MILLOT (*)
Laboratoire de Recherche en Informatique, CNRS UA 410,
.Bat 490, Université Paris-Sud, 91405 Orsay Cedex. FRANCE
Telephone : (+33) 1.69.41.64.32 ,Telefax : (+33) 1.69.41.65.86 E-mail :fanchon@lri.lri.fr. millot@lri.lri.fr.
(*) Institut National des Télécommunications,
9 rue Charles Fourier,91011 EVRY.

Abstract : We define a formal framework to model executions and dynamic placement of programs on networks of processors. It provides an equational characterization of programs with respect to their expansion.
Key-words: true concurrency, partial orders, operational semantics, bisimulation, dynamic placement, networks .

Introduction

We define formal models for dynamically placed concurrent programs on networks of processors. A programmer is not supposed to know the particular topology of a network on which his program is going to execute, neither what the load of processors at the time of execution will be. We suppose he has a special parallel operator noted p⫫q, with the following meaning : execute p on the current processor and (if possible) execute q elsewhere (see [MV]). When a process p⫫q begins to execute, the network first expands the process q on a target processor, by mean of a particular action called expansion, which involves both source and target processors, and then the parallel execution of p and q is performed.This operator is asymmetric, thus loosing commutativity for the semantics reflecting this particularity. It is necessary to have a new symbol because events can still be causally independent, which is a symmetric property denoted by the usual || symbol.
Our purpose is to model the executions of such programs, and moreover, characterize them by the way they expand on any network.

Formally, we use a now well defined method : a program is a term of an initial algebra, from which we syntactically derive a transition system, which models the executions of the program on an abstract machine. Usually the states are terms, the initial state is the term we want to model, and transitions are triples

term ---action--->term'

meaning that the term on the left can execute the action, and then behaves like term'. This gives the so-called operational semantics. Then different semantic equalities can be defined on terms by identifying those whose transition systems have similar behaviours, by means of bisimulation relations [Mi] [Po] [GV] .

The current semantics based on this method depend mainly on two criteria:

1) what is an action labelling a transition.

In interleaving models, the transitions are labelled by elementary actions executed by any of the processes executing concurrently, and an execution of a term is described by a concatenation (total order) of such elementary actions, leading to the equation allb = ab+ba. Alternatively a transition can be labelled by a multiset of concurrently executable elementary actions, or step. In that case the previous equation does not hold but allb +ab = allb holds. Finally labels can be partially ordered sets labelled by elementary actions, where two events are ordered if and only if they causally depend on each other [Pr] [BC][Wi].This leads to "truly" concurrent models, and we follow this scheme. From now on, we call actions or computations such labelled posets.

2) what is taken into account in the computations when defining a bisimulation.

In strong bisimulations, all the components of the actions are taken into account [BC][GV]. In observational equivalences, one choses a subset of 'observable elementary actions', and two terms are identified through bisimulations taking into account only such observable events. We shall use both methods in the following.The question is : what do we want to observe?

As we model executions of programs with the parallel operator described above, we want to observe the partial ordering of elementary actions of the different concurrent processes, but also to observe the expansion of the program in a network. Due to the variety of networks, we look for syntactic properties of programs which would imply properties of their executions on any network.

For that purpose we introduce the expansion operational semantics of a term, in short e-semantics, which describes its potential expansion on an ideal infinite graph. In this semantics, a state of the transition system is not a simple term, but a tree-like representation of a term modeling the expansions which occured before that state was reached:

- the initial state of an execution is a single node labelled by the term to execute.
- a state (the program at a particular stage of an execution) is a directed tree with nodes labelled by terms, object called expansion tree, in short e-tree, where a new labelled node has been added each time an expansion has been executed (This view can be related to the Petri-net operational semantics for CCS as defined in [DDM] or [Old], where states arise from the decomposition of terms in sequential components. E-trees can be viewed as complete sets of such components together with a topological aspect).

In this framework, an elementary action occurs at a particular node of an e-tree:

-the elementary actions are pairs (node(s), action), called elementary expanded actions, recording the action executed (which can be an expansion), and the node(s) where it occurs (two for an expansion, one for the other actions).

-the computations labelling transitions, called expanded actions or computations, are posets labelled by such pairs.

On such transition systems, we can define bisimulations focusing on different aspects of the expanded computations: hiding the node aspect leads to a commutative semantics of the parallel operator, hiding occurences of expansions and flattening e-trees to obtain simple terms leads to a semantic very close to the model in [BC].
We can also project on occurences of expansions, thus identifying terms which have the same behaviours w.r.t expansion. The obtained bisimulation relation is not a congruence w.r.t. the choice operator. We exhibit the largest congruence included in that relation and we give a set of equations which is correct and complete for this congruence : the quotient algebra is then characterized equationally and this is the second contribution of this work.
Topics like sequencing, (implying distributed termination), communication (both synchronous and asynchronous) and recursion, are partially worked out and will be added in a next paper.

The last part of the present paper gives an example of how this model applies to execution of programs on realistic networks: we define a placed operational semantics to model the executions of a term on a particular graph, where the vertices model processors and the edges give the directions for a program expansion from a vertex. This definition of a network only considers a neighbouring relation on processors. In particular we abstract from quantitative aspects like links speeds or processors performances. When a program p ⫫ q executes on a processor, the network has to choose on which processor the program q is going to expand. We make this choice depend only on the neighbouring relation. This allows a direct relation with the previously defined expansion semantics by means of the placement of the e-trees and their computations. In particular the placement of computations can be related to the tasks graph allocation problem (see [PS]), and a further work is to take into account quantitative aspects of both programs and networks. In particular, depending on such quantitative constraints on programs, one may look for the best suited semantic equality : a bisimilation relation could take some of these constraints into account so that equivalent terms for such a relation will have similar behaviours on more realistic networks.

1 Preliminaries: labelled posets and partial words

In this section, we give some general definitions on labelled partial orders which we use to define the computations of terms in both expanded and placed semantics. Most of these notions are well known [Pr], in particular the isomorphisms of labelled posets, leading to partial words or partially ordered multisets , and the concatenation and parallel operators on labelled partial orders and partial words

Given an alphabet A, we call A -labelled poset, in short A-LP, a triplet s = (E, ≤, l) where

(i) E is a finite set of elements, the events,

(ii) ≤ is a partial order on E,

(iii) l : E —> A is a labelling function.

An isomorphism $s_0 \leftrightarrow s_1$ between two A-LP is a one-to-one mapping of their events sets preserving order and labelling. From now we consider an A-LP as a representative of its isomorphism class, and we call partial words on A, noted PW(A) the set of isomorphism classes of A-LPs (also called pomsets on A in [Pr]). We can view A as a subset of PW(A) , by identifying an element a in A with the A-LP $(\{\varepsilon\}, =, l)$ where $l(\varepsilon) = a$.
We note 1 the class of the empty A-LP : $1 = (\emptyset, \emptyset, \emptyset)$.

We define on A -LP's the operations of sequential and parallel composition.
Given two A -LP's s_0 and s_1, we define their sequential (resp. parallel) composition as the A -LP obtained by juxtaposing s_0 and s_1, and setting the relation $\leq$ (resp. no extra relation) between the events of s_0 and s_1. Formally, assuming $s_i = (E_i, \leq_i, l_i)$ for $i \in \{0, 1\}$, if we call E the disjoint union of E_0 and E_1, i.e. $E = \{0x, x \in E_0\} \cup \{1x, x \in E_1\}$, we define :

$s_0 ; s_1 = (E, \leq, l)$, where $ix \leq jy \iff (i = j \text{ and } x \leq_i y) \text{ or } (i = 0 \text{ and } j = 1)$,
$l(ix) = l_i(x)$

and $s_0 \| s_1 = (E, \leq, l)$, where $ix \leq jy \iff i = j \text{ and } x \leq_i y$
$l(ix) = l_i(x)$

These operations are defined up to isomorphism:
$s_0 \leftrightarrow s_0'$ and $s_1 \leftrightarrow s_1' \Rightarrow (s_0 ; s_1 \leftrightarrow s_0'; s_1')$ and $(s_0 \| s_1 \leftrightarrow s_0' \| s_1')$.
They are then defined on PW(A).
We define the prefixing of an A -LP by an element in A as the sequential composition $a.s = a;s$.
For all p, q , r in PW(A):
$(p ; q) ; r = p ; (q ; r)$,
$(p \| q) \| r = p \| (q \| r)$, $p \| q = q \| p$,
$1;u = u;1 = u$, $1\|u = u\|1 = u$.

We note D(A) the subset of PW(A) finitly generated on $A \cup \{1\}$ by the sequencing and parallel operators.
Let B and C be two alphabets , a morphism $f : D(B) \to D(C)$ is entirely defined by the image of elements in B : $f(1) = 1$, and $f(u \text{ op } v) = f(u) \text{ op } f(v)$, "op" standing for ";" and "||" .

An A-LP $s1 = (E_1, \leq_1, l_1)$ is a sub-A-LP of $s2 = (E_2, \leq_2, l_2)$ if and only if

(i) $E_1 \subset E_2$
(ii) $\leq_1 = \leq_2 / E_1$
(iii) $l_1 = l_2 / E_1$.

An A -LP $s1 = (E_1, \leq_1, l_1)$ is a prefix of an A-LP $s = (E, \leq, l)$, and we write $s1 \leq s$, when

(i) s1 is a sub-A-LP of s
(ii) $\forall x \in s1, y \leq x \Rightarrow y \in s1$

We call residue of s relative to s1, noted $s|s_1$, the sub-A -LP of s whose elements are in $E - E_1$,
i.e. $s|s_1 = (E - E_1, \leq/E - E_1, l/E - E_1)$.
The definitions of prefix and residue extend to partial words in an obvious way.

2 The expansion model

2.1 Syntax for the expansion model

We start from an alphabet A of atomic actions not containing the symbol e, and use it throughout the paper. The terms we consider are a simplified version of CCS terms, defined from the constant Nil by prefixing with an action in A (a.p), choice (p+q) and parallel composition ($p \amalg q$). Then we define e-trees, together with some operations needed in the structured transition relation on e-trees. Finally we define the syntax of expanded computations, introducing a special symbol e not belonging to A, denoting an expansion.

Terms

The set of terms T is the initial algebra on the following one sorted signature :
constants : Nil , unary operators : { a._ , a ∈ A} , binary operators : _+_ and _ $\amalg$ _ .
or equivalently the language generated by the following grammar :

$p := Nil \mid a.p \mid p \amalg p \mid p + p$ for all $a \in A$.

E-trees

Definitions : We call **tree domain** a prefix closed subset of N^*, the set of words on N (where we use N instead of N^+ for the set of strictly positive integers). The empty word ε is called root, each word is a node, and maximal elements are leaves.
We only consider tree domains for which if m.i is a node (where m is a word on N and i is an integer), then m.j is also a node, for every integer $j < i$.
We define the width at depth 1, of a tree domain D as the number w(D) of one letter words contained in D.

Definition :

An **e-tree** is a mapping π from a tree domain denoted dom(π) to the set of terms T.We note eT the set of e-trees. The elements of dom(π) are the nodes of π.
We call root-e-tree an e-tree with only one node. A term p can be identified with the root-e-tree π such that $\pi(\varepsilon) = p$, and we consider T as the subset of root-e-trees in eT : $T \subset eT$.
The width at depth 1 of an e-tree π is the width (at depth 1) of its domain: $w(\pi) = w(dom(\pi))$.

Operations on e-trees

An occurence of a term t within an e-tree π is a node m of π such that $\pi(m) = t$. The e-tree $\pi[m \leftarrow t]$ is derived from an e-tree π by substituting term t to the image of node m :

$dom(\pi[m \leftarrow t]) = dom(\pi)$
$\forall m' \in dom(\pi)$, if $m' \neq m$ then $\pi[m \leftarrow t](m') = \pi(m')$
$\pi[m \leftarrow t](m) = t$

If m is a node of an e-tree π, the sub-e-tree of π at occurence m, denoted $\pi_{/m}$, is the e-tree such that :

$dom(\pi_{/m}) = \{m' \in N^* \mid mm' \in dom(\pi)\}$, and

$\forall\, m' \in dom(\pi_{/m}),\ \pi_{/m}(m') = \pi(mm')$

The e-tree $\pi[/m \leftarrow \pi_1]$ is derived from an e-tree π by substituting the e-tree π_1 to the sub-e-tree $\pi_{/m}$ as follows :

$dom(\pi[/m \leftarrow \pi_1]) = (dom(\pi) - m.dom(\pi_{/m})) \cup m.dom(\pi_1)$

$\pi[/m \leftarrow \pi_1](m') = \pi(m')$ for $m' \in dom(\pi)$ with $m' \neq mm''$

$\pi[/m \leftarrow \pi_1](m'') = \pi_1(m'')$ if $m' = mm''$

It can be noticed that $\pi[/m \leftarrow \pi_{/m}] = \pi$.

Given two e-trees π_1 and π_2 and a term p, the e-tree $\pi_1 \oplus_p \pi_2$ is obtained by sticking back the branches issued from the roots of π_1 and π_2 to a single root with image p. The width (at depth 1) of the e-tree is the sum of the widths of π_1 and π_2.

So the e-tree $\pi_1 \oplus_p \pi_2$ is defined by :

let $w_1 = w(\pi 1)$ and $w_2 = w(\pi 2)$, then

$dom(\pi_1 \oplus_p \pi_2) = dom(\pi 1) \cup \{(w_1 + i)m, i \in N, m \in N^*, im \in dom(\pi 2)\}$

i.e. $w(\pi_1 \oplus_p \pi_2) = w_1 + w_2$

$\forall\, i \in dom(\pi_1 \oplus_p \pi_2) \cap N$,

if $i \leq w_1$, then $(\pi_1 \oplus_p \pi_2)_{/i} = \pi_{1/i}$

if $w_1 < i \leq w$, then $(\pi_1 \oplus_p \pi_2)_{/i} = \pi_{2/i} - w1$

$(\pi_1 \oplus_p \pi_2)(\varepsilon) = p$

Expanded actions

We define the sets of elementary expanded actions and expanded computations for e-trees. The first ones are pairs (node, action) , the seconds partial words of such pairs. The particular symbol e denotes an expansion. Such an expansion occurs when a node m of $dom(\pi)$ is such that $\pi(m) = p \amalg q$; if $i = w(\pi/m)$ (i.e. i is the greatest integer such that $mi \in dom(\pi)$), π executes the expansion (m(i+1),e), and the domain of the resulting e-tree π' is $dom(\pi') = dom(\pi) \cup \{m(i+1)\}$, with $\pi'(m) = p$ and $\pi(m(i+1)) = q$.

Definitions.

The alphabet Æ of elementary expanded actions is the set reunion :

$Æ = \{(m, a) \mid m \in N^*, a \in A\} \cup \{(mi, e) \mid m \in N^*, i \in N\}$

Such an action has a corresponding set of associated nodes defined by :

(i) $nd((m, a)) = \{m\}$

(ii) $nd((mi, e)) = \{m, mi\}$

The set eD of expanded actions is the least subset of D(Æ) containing Æ and the elements defined by the following rules:

if $(\alpha \in Æ$ and $\sigma \in eD)$ then $\alpha.\sigma \in eD$, and $nd(\alpha.\sigma) = nd(\alpha) \cup nd(\sigma)$.

if $(\sigma, \tau \in eD$ and $nd(\sigma) \cap nd(\tau) = \emptyset)$ then $\sigma \| \tau \in eD$, and $nd(\sigma \| \tau) = nd(\sigma) \cup nd(\tau)$.

Finally we define two morphism from eD to eD as the extentions to eD of the following mappings from Æ to Æ:

Prefixing with m , with $m \in N^*$: $m.(m', æ) = (mm', æ)$, for $æ \in A \cup \{e\}$.

Adding k at depth 1 , with $k \in N$: $_k.(im, æ) = ((k + i)m, æ)$, with $i \in N$, $æ \in A \cup \{e\}$,

We have $nd(m.\sigma) = \{mm' \mid m' \in nd(\sigma)\}$, and $nd(_k.\sigma) = \{(k + i)m \mid im \in nd(\sigma)\}$.

2.2. Expansion operational model

We define the expanded transition system ST(t) associated to a term t. It is a quadruple ST(t) = (eT,eD,->,t) , t is the initial e-tree and the transition relation -> is a subset of eT x eD x eT. This relation depends on the structure of e-trees, and is defined by a particular set of inference rules. Compared to the usual rules defining a transition relation on terms, these depend both on the tree structure of e-trees and the values of the nodes.

The transition relation .

We note $\pi_1 \text{—}\sigma\rightarrow \pi_2$ for $(\pi_1, \sigma, \pi_2) \in$ -> , -> is the least subset in eT x eD x eT generated by the following set of rules, where p and q denote terms (i.e. root e-trees):

E00 $\Rightarrow \quad \pi \text{—}1\rightarrow \pi$

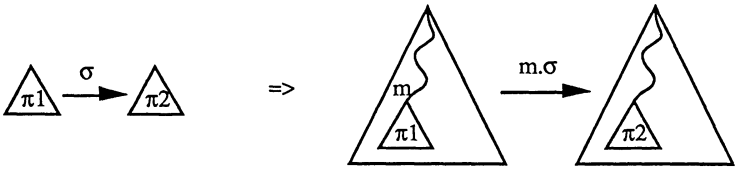
E01 $a \in A \quad \Rightarrow \quad a.p \text{—}(\varepsilon,a)\rightarrow p$

E02 $p \text{—}\sigma\rightarrow \pi \quad \Rightarrow \quad a.p \text{—}(\varepsilon,a).\sigma\rightarrow \pi$

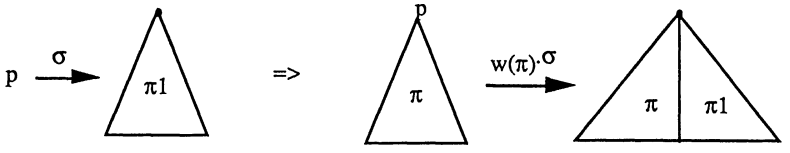
E11/2 $p \text{—}\sigma\rightarrow \pi \quad \Rightarrow \quad p + q \text{—}\sigma\rightarrow \pi$

$q + p \text{—}\sigma\rightarrow \pi$

E21 $\pi_1 \text{—}\sigma\rightarrow \pi_2$ and $\pi_{/m} = \pi_1 \quad \Rightarrow \quad \pi \text{—}m.\sigma\rightarrow \pi[/m \leftarrow \pi_2]$



E22 $\pi(\varepsilon) = p$ and $p \text{—}\sigma\rightarrow \pi_1 \quad \Rightarrow \quad \pi \text{—}_{w(\pi)}.\sigma\rightarrow \pi \oplus [\pi_1(\varepsilon)]\ \pi_1$



E23 $\pi \text{—}\sigma\rightarrow \pi_1, \pi \text{—}\tau\rightarrow \pi_2$ with $nd(\sigma) \cap nd(\tau) = \emptyset$

$\Rightarrow \pi \text{—}\sigma|\tau\rightarrow \pi'$ such that $dom(\pi') = dom(\pi_1) \cup dom(\pi_2)$ with
$\pi'/nd(\sigma) = \pi_1$, $\pi'/nd(\tau) = \pi_2$, $\pi' = \pi$ elsewhere

E31 $\Rightarrow p \amalg q \text{—}(1, e)\rightarrow \pi$ where $dom(\pi) = \{\varepsilon, 1\}$, with $\pi(\varepsilon) = p$ and $\pi(1) = q$

E32 $p \amalg q \text{—}(1, e)\rightarrow \pi$ and $\pi \text{—}\sigma\rightarrow \pi' \Rightarrow p \amalg q \text{—}(1, e).\sigma\rightarrow \pi'$

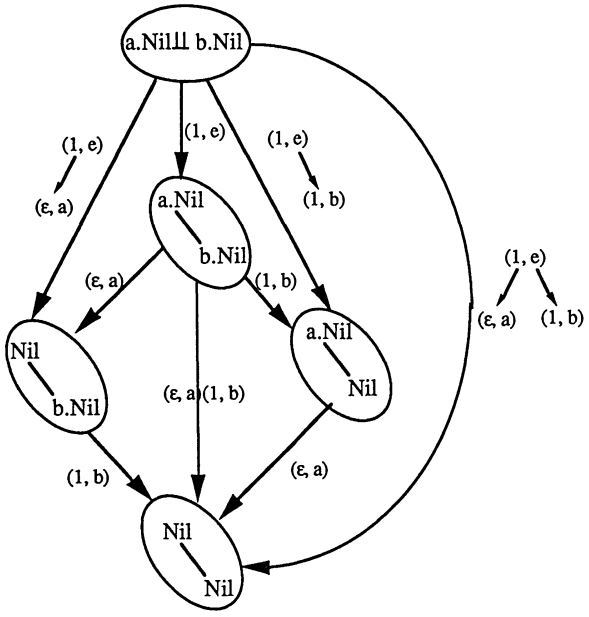


Fig1: The transition system of a.Nil⫫b.Nil .

The expanded actions (which are partial words) labelling the arrows between the (circled) e-trees are represented by directed graphs labelled by elementary expanded actions.

Computations of e-trees.

The computations of an e-tree π are the expanded actions σ such that $\pi \text{—}\sigma\rightarrow \pi$ ' for some e-tree π'. Any prefix of a computation is a computation of the same e-tree, and leads to an e-tree which can execute the residue :

Lemma 2.3:

$\pi \text{—}\sigma\rightarrow \pi$ ', and $\sigma_1 \leq \sigma$, then $\pi \text{—}\sigma_1\rightarrow \pi_1$ and $\pi_1 \text{—}\sigma|\sigma_1 \rightarrow \pi$ ' for some e-tree π_1.

Lemma 2.4

If $\pi \text{—}\sigma_1\rightarrow\pi_1$ and $\pi_1\text{—}\sigma_2 \rightarrow \pi$ ', then it exists σ such that $\pi \text{—}\sigma\rightarrow \pi$ ', $\sigma_1 \leq \sigma$ and $\sigma_2 = \sigma|\sigma_1$.

The following lemma relates the computations of p⫫ q to the ones of p and q:

Lemma2.5:

$p \between q \xrightarrow{\sigma} \pi \Rightarrow \exists\ \pi p, \pi q, \sigma_p, \sigma_q : p \xrightarrow{\sigma_p} \pi p,\ q \xrightarrow{\sigma_q} \pi q,\ \sigma = (1, e).(1.\sigma_p \| 1.\sigma_q)$, and

$\pi = \pi p \oplus [\pi p(\varepsilon)]\ \pi'[/1 <\!\!- \pi q]$, where π' is the e-tree such that $dom(\pi') = \{\varepsilon, 1\}$, $\pi'(\varepsilon) = p$ and $\pi'(1) = q$.

Recursion.

Recursion can be introduced in this semantics, after adding process variables and a recursion operator on terms: p := Nil | x | a.p | p+p | p⫫ p | μx.p

The rule for recursion is :

$p[\mu x.p/x] \xrightarrow{\sigma} \pi \quad \Rightarrow \quad \mu x.p \xrightarrow{\sigma} \pi$

2.3. Bisimulation semantics

For each term we defined an expanded transition system, from which we derive its (expanded) computations. We define here different bisimulations on transition systems, depending on what we observe about the computations. First we define different abstractions on computations, then we associate a particular bisimulation to each of these abstractions. Finally we focus on the so-called expansion bisimulation, giving an equational characterization of the obtained equivalence on terms.

Abstraction from nodes .

We first identify expanded actions which only differ by the node part of their components.

Let u : D(Æ) --> D(A U {e}) the morphism defined by u(m,d) = d for all (m,d) in Æ , we say σ and τ are equivalent on A U {e}, we note $\sigma =_u \tau$, iff $u(\sigma) = u(\tau)$.

Abstraction from expansions and nodes.

We abstract from nodes and observe only events in A , thus eliminating expansions .

Let p : D(Æ) --> D(A) the morphism defined by p(m,a) = a if $a \neq e$, p(m,e) = 1 , we say σ and τ are equivalent on A, we note $\sigma =_a \tau$, iff $p(\sigma) = p(\tau)$.

Projection on expansions.

We identify expanded actions which have same projections on expansion events (m,e).

Let e : D(Æ) --> D(N*.N) the morphism defined by e(m,a) = 1 if $a \neq e$, e(m,e) = m , we say σ and τ are equivalent for expansion, we note $\sigma =_e \tau$, iff $e(\sigma) = e(\tau)$.

Remark : we don't want to hide the nodes where expansions occur, since we would loose an important part of the information, and go back to a commutative parallel operator .

The bisimulations :

Definition : let $\equiv$ be an equivalence relation in eD x eD, we say that a relation R in eT x eT is a bisimulation of two e-trees π_1 and π_2 with respect to $\equiv$, iff :

1) $R(\pi_1 , \pi_2)$

2) $\pi_1 \text{—}\sigma_1\rightarrow \pi_1' \Rightarrow \exists\ \sigma_2 , \pi_2'$ so that $\pi_2 \text{—}\sigma_2\rightarrow\pi_2'$, $R(\pi_1', \pi_2')$ and $\sigma_1 \equiv \sigma_2$.

3)$\pi_2 \text{—}\sigma_2\rightarrow \pi_2' \Rightarrow \exists\ \sigma_1 , \pi_1'$ so that $\pi_1 \text{—}\sigma_1\rightarrow \pi_1'$, $R(\pi_1', \pi_2')$ and $\sigma_1 \equiv \sigma_2$.

π_1 and π_2 are said bisimilar with respect to $\equiv$ iff such a bisimulation exists.

The following is a standard result : the relation of bisimilarity w.r.t an equivalence on eD is an equivalence relation on eT x eT .

For each congruence on D(Æ) defined previously , which restricts in an obvious way to eD , we define a particular bisimulation on eT :

Definitions:

We say π_1 and π_2 are strongly equivalent , noted $\pi_1 \approx \pi_2$, iff they are bisimilar w.r.t. equality of expanded actions.

We say π_1 and π_2 are u-equivalent , noted $\pi_1 \approx u\ \pi_2$, iff they are bisimilar w.r.t $=u$.

We say π_1 and π_2 are a-equivalent , noted $\pi_1 \approx a\ \pi_2$, iff they are bisimilar w.r.t $=a$.

We say π_1 and π_2 are e-equivalent , noted $\pi_1 \approx e\ \pi_2$, iff they are bisimilar w.r. $=e$.

We now restrict these equivalence relations in eTxeT to TxT. As we want to make equational calculii on terms, we look for congruences in T. Only the first two equivalences are congruences for all the operators prefixing, choice and parallel composition on terms.The equivalences $\approx a$ and $\approx e$ are not congruences for the choice operator (In particular we have Nil⇊Nil $\approx a$ Nil, but we have not Nil⇊Nil +p $\approx a$ p for all terms p; the left term can execute an invisible expansion and then stop, when the second may have any other behaviour).

Theorem 2.1

The restrictions of $\approx$, $\approx u$, $\approx a$ and $\approx e$ to TxT are congruences for the prefixing and parallel operators. The restrictions of $\approx$ and $\approx u$ to TxT are congruences for the choice operator .

Lemma 2.6 :

$\approx \subset \approx u \subset \approx a$ and $\approx \subset \approx e$.

Lemma 2.7.

We have $p+q =_x q+p$, $p+(q+r) =_x (p+q)+r$, $p+p =_x p$, $Nil+p =_x p+Nil =_x p$, for any equivalence $=_x$ in $\{\approx,\approx u,\approx a, \approx e\}$, and for all terms p,q,r.

Lets name **C** the following set of equations :

C1: $p+q = q+p$ C2: $p+(q+r) = (p+q)+r$

C3: $p+p = p$ C4 : $Nil+p = p+Nil = p$

As $\approx$ and $\approx u$ are also congruences , the lemma shows that the equations **C** are valid in both the quotient algebras $T/\approx$ and $T/\approx u$.

There is no place here to show the consistency of the equivalence ≈a w.r.t semantics for terms built with the usual || operator, which are roughly characterized by the following transition rules [BC]:

$p \xrightarrow{u} p' \Rightarrow p \| q \xrightarrow{u} p' \| q$

$q \xrightarrow{v} q' \Rightarrow p \| q \xrightarrow{v} p \| q'$

$p \xrightarrow{u} p', q \xrightarrow{v} q' \Rightarrow p \| q \xrightarrow{u \| v} p' \| q'$

In these models , the parallel operator is commutative .

Theorem 2.2 :

For all p and q in T , $p \amalg q \approx u\ q \amalg p$ and $p \amalg q \approx a\ q \amalg p$.

Since ≈u is a congruence , $p \amalg q = q \amalg p$ is valid in T/≈u .

2.4. Expansion semantics

This is the semantics we are most interessed in . The problem arises from the fact that the equivalence w.r.t. expansion ≈e is not a congruence for the choice operator : we note that for any term p and action a, a.p and p are e-equivalent, from the rule $a.p \xrightarrow{(\varepsilon,a)} p$.The bisimulation relation is identity on e-trees together with the pair (a.p, p).

But if $p = r \amalg s$, for any term q not e-equivalent to p , we do not have $a.p + q \approx e\ p + q$, because the left side can execute (ε,a) (by definition (ε,a) =e 1), and rewrites in p , when the second may have to execute an expansion as first action, and so p and p+q would have to be e-equivalent.

Following the tracks of Milner , we define the greatest congruence included in it , and then caracterize it equationally.

Theorem 2.3:

The equivalence relation on TxT noted ≅e , called expansion congruence , and defined by :

$p \cong e\ q <=> (\forall r \in T\ p+r \approx e\ q+r)$,

is a congruence for all prefixing , choice and parallel operators and it is the greatest one included in ≈e .

The proof uses the following lemma:

Lemma 2.7 :

for all a, b in A , p ,q, r in T , we have

1) $p \approx e\ q => a.p \cong e\ b.q$

2) $r \approx e\ q => a.(p + q) + r \cong e\ a.(p +q)$

3) $(p \approx e\ q$ and $r \approx e\ s) <=> p \amalg r \cong e\ q \amalg s$

Let's name **E** the following set of equations :

E0 : a.p = b.p E1 : a.(b.p) = a.p E2 : a.(p + q) +p = a.(p +q)

E3 : $a.p \amalg q = p \amalg q$ E4 : $p \amalg b.q = p \amalg q$

for all a, b in A ,p,q in T .

Let **S** = **C** U **E** , and =s the congruence generated by **S** on T , we have the result that the set of equations **S** is correct and complete for the expansion congruence, i.e. two terms are e-congruent if and only if they are equal in the equational theory of **S** , **=s** . This theorem is the characterization theorem of expansion congruence :

Theorem 2.4:
The expansion congruence is the congruence generated on T by the equations **S:**
for all p, q in T , p =s q <=> p ≅e q .

3.The placed model

The purpose of this section is to show how the expansion semantic defined above can be used when a program executes on a network of processors defined as a graph whose vertices model processors and edges give the directions for a program expansion from a vertex.
We define the placed operational semantics of a term on a network as a particular transition system, whose states, called configurations, describe the network at a particular stage of a computation : to each vertex is associated the multiset of terms currently executing on it. An action occurs on a particular vertex, eventually two vertices for an expansion. All the actions occuring on the same vertex are totally ordered in a placed computation. We show that a configuration corresponds to a so-called placement of an e-tree in the expansion semantics of the executing term. A placement of an e-tree on a network is a graph morphism mapping the nodes of the e-tree into the set of processors. In particular only computation leading to e-trees injectively embeddable in the graph of the network can be executed in true parallelism on that network.

3.1. Placed operational semantics

When modeling the executions of a term on a network, one has to chose an initial configuration, i.e. a vertex on which the term initially executes. All other vertices are iddle.
The elementary actions, called placed actions, contain the information of the processor(s) on which they execute, a single one for actions in A, pairs of processors for expansions. A computation is a partial word of such placed actions.The transition relation defines which (placed) computations can be executed from any particular configuration and the resulting one.

Networks and Configurations

Definitions:

Let G=(V,E) be a non-oriented graph, i.e. E is a symmetric relation included in VxV.

A configuration on G is a mapping

$$h : V \longrightarrow N^T$$

For each vertex i of G h(i) is the multi-set of terms executing on it.

We call C_G the set of configurations on G.

The choice of a target processor for an expansion could be modeled by a function next : C_G x V —> $\mathcal{P}(V)$, or more deterministically, next : C_G x V —> V$\cup$ {∂}, making the choice depend on the configuration (where next(h,i) = ∂ would forbid any expansion from i in the configuration h, meaning the network is saturated). For now, we make the simple assumption that a term expands either on a neighbour of i for the relation E in VxV, or on the processor i itself, i.e. :

$\forall$ i, next(i) = {i} $\cup$ {j | (i, j) $\in$ E}.

In particular an expansion from a vertex i towards any vertex in next(i) is always possible. In the rules P1, P2 and P3 below, defining the transition relation from a configuration to another, the only condition for an expansion from vertex i to vertex j is that j belongs to next(i) and does not depend on the particular configuration in which the expansion occurs.

Placed actions

We represent the elementary actions executed on a network , elementary placed actions, by pairs: the first part represents the processor(s) (two for an expansion), on which it occurs, the second part, the action symbol.

A placed action or computation is a partial word of such elementary placed actions so that the ones placed on a same processor are totally ordered.

Definition: the set Ap(G) of **elementary placed actions** on G is the set reunion

Ap(G) = {(i, a), i $\in$ V, a $\in$ A} $\cup$ {((i, j), e), i $\in$ V, j $\in$ next(i)}.

We omit G in Ap(G) whenever there is no confusion.

Proc is the mapping Ap —> $\mathcal{P}(V)$ which maps a placed action on the corresponding processor(s):

(i) Proc((i, a)) = {i}

(ii) Proc(((i, j), e)) = {i, j}

We then define the set of placed actions as a particular subset of partial words on Ap: we cannot here define placed actions from Ap and the operators ; and || , as we require any two elements whose labels have a common processor to be ordered. We define on Ap-labelled posets a concatenation operator noted "*" , which is a particular form of what Pratt calls local concatenation.

Definitions:

Let $(E,\leq,l)$, $(E',\leq',l')\in$ Ap-LP , then $(E,\leq,l) * (E',\leq',l') = (E\cup E', \leq^*, l\cup l')$, where $\leq^*$ is the least order containing the set $\leq \cup \leq' \cup \{(e,e') \in ExE' , Proc(l(e))\cap Proc(l(e')) \neq \emptyset\}$.

This operator is defined on partial words, i.e. : $s_0 \leftrightarrow s_0'$ and $s_1 \leftrightarrow s_1'$ => $(s_0 * s_1 \leftrightarrow s_0'^* s_1')$

The set D_G of **placed actions on G** is the set of isomorphism classes of Ap labelled posets generated by the following grammar :

$u := a \mid u*u \quad a \in Ap$.

For $u = (E,\leq,l) \in$ Ap-LP , the set of processors of u is $Proc(u) = \cup_{e \in E} Proc(l(e))$.

Lemma3.1:

Let $u = (E,\leq,l)$ in D_G , then for all e, e' in E :

$e \to e'$ => $Proc(l(e))\cap Proc(l(e')) \neq \emptyset$,where $e \to e'$ <=> ($e \leq e'$ and $(e\leq x\leq e'$ => $x=e$ or $x=e')$)

$Proc(l(e))\cap Proc(l(e')) \neq \emptyset$ => $e\leq e'$ or $e'\leq e$.

The transition relation (Placed operational semantics)

In this semantics the transition relation is a subset of $C_G \times D_G \times C_G$, i.e. a configuration h executes a placed action μ and rewrites in another configuration h', noted $h \text{—}\mu\to h'$. We first define a simple transition relation in TxAxT, noted $\to_s$, which does not involve parallelism. Only terms which can execute an action involving a single processor (local action) are concerned. It is generated by the following rules:

S1

$a \in A \quad \Rightarrow \quad a.p \text{—}a\to_s p$

S2

$p \text{—}a\to_s p' \quad \Rightarrow \quad p+q \text{—}a\to_s p'$

$q+ p \text{—}a\to_s p'$

The placed transition relation is the least subset of $C_G \times D_G \times C_G$, generated by the following rules, using the preceding ones :

P0

If a term on a processor i can execute a local action a , the configuration can execute the placed action (i,a) and rewrites accordingly:

if $h(i)(p) > 0$ and $p \text{—}a\to_s p'$,

then $h \text{—}(i, a)\to h'$ with $h'(j) = h(j)$ when $j\neq i$,

$h'(i)(p) = h(i)(p) - 1$ and $h'(i)(p') = h(i)(p') + 1$

P1

If the term $p \perp\!\!\!\perp q$ executes on processor i and j belongs to next(h,i) , then the configuration can execute an expansion of q on j by means of the placed action ((i,j),e) and rewrites accordingly :

if h(i)(p ⫫ q) > 0 and j ∈ next(i)

then h —((i, j), e)→ h' with h'(l) = h(l) when l≠i, j, h'(i)(p Æ q) = h(i)(p Æ q) - 1, and

if i = j

if p = q, then h'(i)(p) = h(i)(p) + 2

else h'(i)(p) = h(i)(p) + 1 and h'(i)(q) = h(i)(q) + 1

if i ≠ j

h'(i)(p) = h(i)(p) + 1, and h'(j)(q) = h(j)(q) + 1

P2

Defines the possible choice of an expansion:

if h(i)(p⫫q + r) > 0 and j ∈ next(i)

then h —((i, j), e)→ h' with h'(l) = h(l) when l ≠i, j, and

h'(i)(p ⫫ q + r) = h(i)(p ⫫ q + r) - 1,

if i = j

if p = q, then h'(i)(p) = h(i)(p) + 2

else h'(i)(p) = h(i)(p) + 1 and h'(i)(q) = h(i)(q) + 1

if i ≠ j

h'(i)(p) = h(i)(p) + 1, and h'(j)(q) = h(j)(q) + 1

P3

like P2, but with r + p⫫q.

The last rule is the only one which builds computations from simpler ones

P4

h —u→ h', h' —v→ h" ⇒ h —u*v→ h".

This means in particular that if Proc(u)∩Proc(v) = ∅, we have h—u||v→ h".

Remark: one can now relate a placed computation to the A-labelled poset obtained by abstracting from the vertices, and what appears is the interleaving of events from concurrent processes executing on the same node. On another hand one can also project on the vertices, and see an execution from the loading point of view.

The prefixes of a placed computation of a configuration are computations too and so are the obtained residues. This last point is due to the particular function next:

Lemma3.2:

h—μ→ h', and $\mu_1 \leq \mu$, then h —μ_1→ h_1 and h_1 —$\mu|\mu_1$ → h ' for some computation h_1.

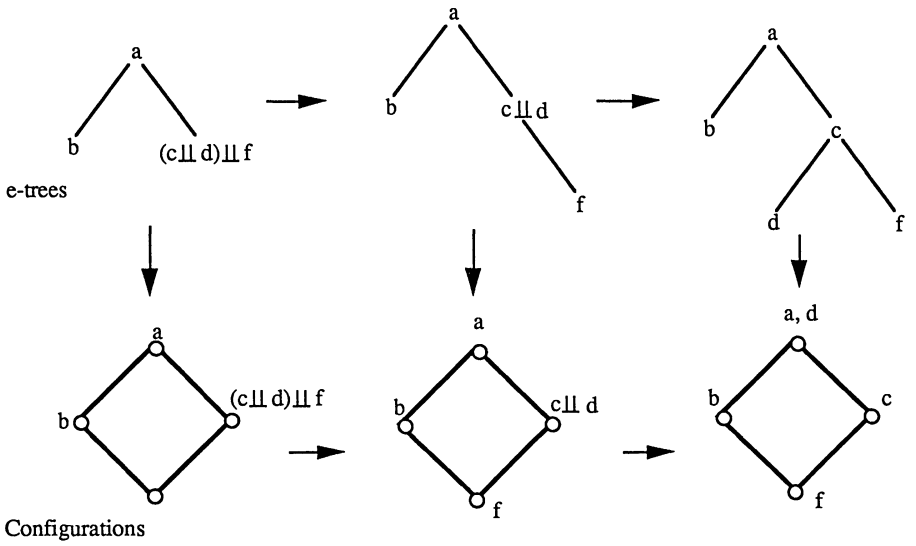


Fig 2.:Placements of e-trees. The vertical arrows represent the placement relation between e-trees, (upwards) and configurations. Horizontal arrows represent the transitions. Computations are not represented.

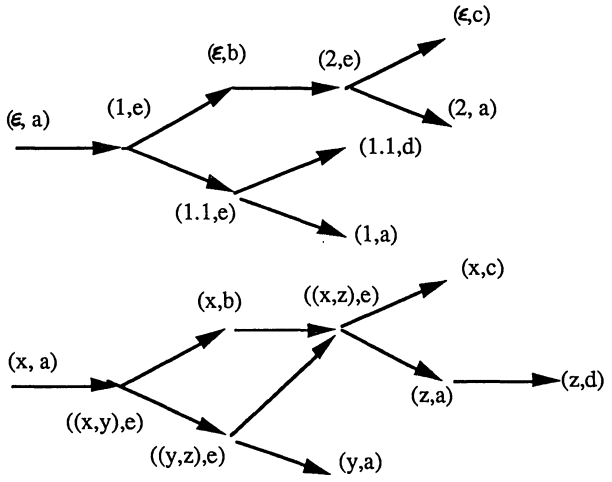


Fig 3. Placement of a computation of the term : a.((b.(c.Nil⫫a.Nil))⫫(a.Nil⫫d.Nil)).
The vertices of the network are x,y and z, the root is placed on x , the node 1 on y, nodes 2 and 1.1 on z.

3.2 Relation with the e-semantics

The behaviours of a term on a network, as defined in the placed semantics, depend on the graph of the network and the initial configuration. We show in the following how they are related to the e-semantics of the considered term, whatever are the network and the initial configuration, mapping e-trees on configurations and expanded actions on placed actions.

Placement of an e-tree

We say that a configuration h is a placement of an e-tree π on G if and only if there is a mapping

$\varphi : dom(\pi) \longrightarrow V$ such that

1) $(\varphi(m), \varphi(mi)) \in E$, or $\varphi(mi) = \varphi(m)$ for all $mi \in dom(\pi)$, $i \in N$.

2) $h(v)(p) = Card(\{m \in \varphi^{-1}(v) \mid \pi(m) = p\})$

Placement of an expanded action

A placed action $\mu = (E, \leq_p, l_p)$ is a placement of an expanded action $\sigma = (E, \leq, l)$ if and only if

1) for any e in E , the second members (elements of $A \cup \{e\}$) of $l_p(e)$ and $l(e)$ are equal :

$pr2(l_p(e)) = pr2(l(e))$

2)for all events e1 and e2 in E :

if $e1 \leq e2$, then $e1 \leq_p e2$

Notation:

We note $Pl \subset eT \times C_G$ and $P \subset eD \times D_G$ the relations:

$Pl(\pi,h) <=>$ (h is a placement of π).

$P(\sigma,\mu) <=>$ (μ is a placement of σ).

The placement relation between e-trees and configurations induces a bisimulation-like relation between the (expansion) transition relation in eT x eD x eT and the (placed) one in $C_G \times D_G \times C_G$ modulo the placement of expanded actions on placed actions.

Theorem 3:

if $Pl(\pi,h)$ then :

1) $\pi \xrightarrow{\sigma} \pi' \Rightarrow \exists\ \mu , h'$ so that $h \xrightarrow{\mu} h'$, $Pl(\pi',h')$ and $P(\sigma,\mu)$.

2) $h \xrightarrow{\mu} h' \Rightarrow \exists\ \sigma , \pi'$ so that $\pi \xrightarrow{\sigma} \pi'$, $Pl(\pi',h')$ and $P(\sigma,\mu)$.

The proof of 2) uses the fact that if a term may expand from a vertex to another, it does not depend on the configuration.The assertion 1) could hold even if some conditions on expansions where added, depending on the configuration.

On can consider an expanded computation as a task graph, and a further work is to look how the investigations and results on placements of task graphs can be applied.

Conclusion :

This first attempt to model dynamic expansion of programs on networks led to :

1) a new formal framework to model executions , both in an abstract way depending on syntax and in a concrete way depending on networks.
2) a first powerful result of equational characterization of programs, which justifies our formalism.
3) their possible application to distributed programs and systems.

[BC] G. Boudol, I. Castellani, *Concurrency and Atomicity*, Theoretical Computer Science 59 (1988), pp. 25-84

[DDM] P.Degano,R.De Nicola ,U.Montanari, *CCS is an augmented contact-free C/E system.*,LNCS 280,1987,pp 144-165.

[GV] R.van Glabeek,F.Vaandrager,*Petri nets models for algebraic theories of concurrency,* LNCS 259 (1987) 224-242.

[Mi] R. Milner, *Lectures on a Calculus for Communicating Systems*, in : Proc. Seminar on Concurrency, LNCS 197, 1985, pp. 197-220

[MV] D. Millot, J. Vautherin, *True Parallelism on an Unknown Topology*, RR. 502, LRI Univ. Paris-Sud, july 89

[Po] L.Pomello,*Some equivalence notions for concurrent systems,* LNCS 222 (1986) 381-400.

[Pr] V. Pratt, *Modeling Concurrency with Partial Orders*, International Journal of Parallel Programming, vol 15 (1), 1986, pp. 36-91

[PS] C.C.Prince, M.A.Salama, *Scheduling of precedence-constrained task on multiprocessors* The Computer Journal Vol 33 (3) 1990, pp 219-229.

[Wi] G.Winskel, *Event structures* ,LNCS 255 (1987) 325-392.

FORMALISATION OF THE BEHAVIOR OF ACTORS BY COLORED PETRI NETS AND SOME APPLICATIONS*

Yamina Sami †
Guy Vidal-Naquet † ‡
† Laboratoire de Recherche en Informatique, URA 410 CNRS,
Université Paris-Sud, 91405 Orsay Cedex, France
e-mail : sami@lri.lri.fr
‡ Ecole Supérieure d'Electricité, Plateau de Moulon
91190 Gif Sur Yvette Cedex, France

ABSTRACT

In this paper we present a formalisation of actors by colored Petri nets. In order to do that, we give a structural description of an actor program which makes it easy to obtain a colored Petri net with the corresponding behavior. We give a sketch of the proof that the derived colored Petri net reflects the behavior of the corresponding actor program. We show how this formalisation allows a translation of dynamic system into static one which has some applications. Finally we discuss some other possibilities for deriving a colored Petri net, and we put this formalisation in perspective with other works.
Keywords : actors, parallel languages, dynamic systems, static systems, colored Petri nets.

1 INTRODUCTION

Several models have been proposed for concurrent computation, for each model different choices were made. They concern the kind of basic computation unit that is used, the expression of concurrency, the type of communication and synchronization, the possibility, if any, of extension and reconfiguration. One can find an overview of these different models in [AnSc83], [BaStTa89]. The CSP model [Hoa78] is based on synchronous communication and static creation of processes during run time. One of the goals for these choices is simplicity, which allows a whole corpus of results on CSP. These choices naturally bring restrictions in term of power of expression. It is argued in [LiHeGi86] that synchronous communication and static creation of processes have limitations that prevent some essential requirements of concurrency to be satisfied. In particular "the principal concurrency requirement is the following : if one activity within the module become blocked, other activities should be able to make progress". These restrictions bring difficulties for the modeling of open systems which present changes and continuous evolution, and where the closed-world assumption cannot be made [Hew85].

* This work was supported by the Esprit basic researsh Action No 3148 : DEMON.

The Actor model [Agh86a] for parallel computation presents many features like encapsulation, delegation and asynchronous communication [Agh86b]. It has two features that programmers find useful in a language expressing parallelism :

- Dynamicity of the number of actors : processes can be created by an explicit instruction, or implicitely, (like in [Jon 86]).
- Dynamicity of the links between actors : links are not established before execution of the processes (at compilation time), therefore addresses of actors may be computed and communicated during execution.

Although, as argued in [BaVi91], actors retain the same computing effectiveness as the PRAM model, one of the prices to pay for these possibilities is the difficulty of analysis and proof for programs written in languages supporting them. Theoretical models have been proposed in order to establish a theoretical basis for the study of such languages. For example in DCCS [HoKoRa89] an operator is added to CCS in order to express dynamic creation of ports, which introduces the possibility of reconfiguration.

In this paper, we show how to represent an actor program by a colored Petri net (called CPN in the following). The execution of an actor program will correspond to the "token game" of the corresponding CPN, which means that the dynamic aspects for the creation of actors and the modification of links , will be taken into account by the evolution of the tokens. In order to prove the correcteness of this formalisation we compare it with the semantics given in [Agh86a]. We remark that in fact this translation represents a translation of a dynamic system with a variable number of processes and a dynamic interconnection between processes, to a static system with fixed number of processes and fixed interconnection between them. A static interconnection represented essentially by transitions of CPN. We present some other ways of modeling, which represent other ways of representing dynamic systems by static ones.

The aim of this work is not to just show an equivalence between the formalism of actors and the formalism of CPN, but to serve as a basis for future work which analyse actors program by using the tools of CPN, and a step which must lead to an efficient simulation and implementation of actors.

2 ACTORS

The first important work done in the area of an Actor model is found in [Hew77], where computation and control structures (recursion and iteration) are expressed as patterns of message passing.

A general goal of the actor model was to increase the degree of parallelism, by splitting information and control as much as possible between actors, and by making the information needed by an actor as available as possible at any time during execution .

It was necessary to establish some laws which had to be respected by actors in order to cooperate. In [HewBak77], requirements that must be respected by two fundamental orders of events, activation and arrival orders are examined. These two orders are related to a notion of global time. These requirements are revised and justified in [Cli81]. The formal work in the area of actors is contained essentially in [Cli81], [Agh84] (and recaptured in [Agh86a]), [JaRo89] . All these works describe an operational behavior that give a snapshot of the states of the different actors, and of the messages. In spite of the efforts made to have as much parallelism as possible, and to destroy sequential relations between actions, it is sometimes necessary to restrict parallelism in a fair and efficient way. This leads to different kinds of actors.

In order to explain our representation, we give here the main aspects of the actor model found in [Agh86a].

Informally, an actor can receive, process and send messages. The situation of an actor is defined by its behavior, and by its acquaintances :

- A behavior is constituted by several scripts, each script corresponds to the treatment of a message.
- The list of acquaintances of an actor specifies the other actors to which it can send messages.

As described in [Agh86a], the computation in actors system is made by asynchronous point to point message passing. This means that :

- When an actor sends a message, it must know the address of the recipient actor.
- The processing of a message can be delayed (but not indefinitely).

In an actor system, messages arrive in a nondeterministic way. An actor processes a message within an environment defined by :

(i) the list of acquaintances,

(ii) the script and

(iii) the contents of the message, called the communication list.

The processing of a message by an actor can lead to:

(i) a change of its behavior and/or its list of acquaintances,

(ii) an emission of a message to an actor whose address is known, this means that this address belongs to its list of acquaintances, or to the communication list,

(iii) the creation of a new actor, with the specification of its behavior and the initialisation of its list of acquaintances.

The actor model as given in [Agh86a] does not allow an assignment command. Every time an actor receives a message, it must create a new entity, a replacement machine. This replacement machine corresponds to a situation of the actor, and has to execute the message within this situation. These different machines are related to the same actor and may correspond or not to different situations. For this reason we have two kinds of actors called serial and unserial:

- For a serial actor, the processing of a message leads to a change of situation, and the next message can be processed only after this change is realized (this means that a replacement machine is created). Note that this kind of actor, can be viewed as playing the same role as monitors of Hoare [Hoa74] and is useful for modeling shared ressources [HeAtLi79],[HeReAgAt84].
- For an unserial actor, several messages can be processed in parallel, the replacement machine being the same for all messages.

In contrast to what was suggested in [JaRo89], where each actor processes only one message at a time, in our model, we suppose, like in [Agh86a], the existence of the two kinds of actors, and the possibility for an actor to execute several messages simultaneously.

In this paper we present some examples in a language inspired by SAL, a declarative actor language based on Algol and described in [Agh86a]. In our language the program is constituted by two parts. The first part consists in behaviors definition. The second consists in the initialisation, where initial actors are created, and initial messages are sent to them. In the following we give the basic commands.

become : corresponds to a change of behavior and/or of the value of acquaintances list, we suppose (like in SAL) that all behaviors are known at compilation time. In addition to what is included in SAL we suppose that a behavior to which an actor pass is known at compilation time.

send : corresponds to an emission of a message to some specified actor.
let : corresponds to a temporary extension of the environment, in which an actor works. This extension is done by a creation of actors whose addresses have to be used in a specified package of commands.
conditional command : is defined as usual.

Now, we give a grammar for the language used in this paper. Terminal symbols are shown with bold characters. For concision's sake, some non terminals are not completely explicited.

<behavior definition> :=**def** <behavior name> (<acquaintance list>) [<communication list>] <command> **end def**
<command> := <conditional command> / <become command> / <send command> / <let command> /{<command>}*
<become command> := **become** <behavior name> (<acquaintance list>)
<send command> := **send** <communication list> **to** <target>
<conditional command> := **if** <logical expression> **then** <command> {**else** <command> } **fi**
<let command> := <let binding> { <command>}
<let binding> := **let** <actor name> = **new**<behavior name> (<acquaintance list>) { **and** <let binding>}
<actor program> := {<behavior definition>}*{<let command>}*

An acquaintance list is constituted by identifiers which represent addresses of actors, a communication list is constituted by heads, each head is followed by a specific list of identifiers and is related to a specific message to which this behavior can react.

Note that in a script attached to a type of message within a behavior, there must be at most one executable command "become". In the absence of this command, the replacement machine created is the same as the original. In contrast to the assumption made in [JaRo89], where commands contained in one script are executed in an arbitrary order, we suppose, like in [Agh86a] that the commands of a script are executed concurrently. An actor modifies his internal state and concurrently modifies its external environment, all the operations concerning the external environment are done concurrently.

In other words concurrency in the actor model comes from the concurrent treatment of messages, as well as the concurrent execution of commands[AgHe85].

2 COLORED PETRI NETS.

The basic ideas for using colored nets are the following : The colors of the tokens are used to differenciate actors, and to represent the information of messages. Subnets will correspond to the different scripts. The firing of a transition will correspond to an action being executed.
In this paper we essentially follow the definition found in [Jen86].

Definition 1 : *A CP-matrix is a 6-tuple* $N=(P, T, C, I_-, I_+, M_0)$, *where*

(1) P is the set of places.
(2) T is the set of transitions.
(3) $P \cap T = \emptyset$ *and* $P \cup T \neq \emptyset$.
(4) C is the color-function defined from $P \cup T$ *into non-empty sets.*
(5) I_- *and* I_+ *are linear functions of multisets, they are called the negative and positive incidence-function and are defined on PxT, such that :* $I_-(p,t), I_+(p,t) \in [C(t)_{MS} \rightarrow C(p)_{MS}]_L$ *for all* $(p,t) \in PxT$,

(6) $\forall p \in P\ \exists t \in T : I_-(p,t) \neq 0 \ \vee\ I_+(p,t) \neq 0$ *and* $\forall\ t \in T\ \exists p \in P : I_-(p,t) \neq 0 \ \vee\ I_+(p,t) \neq 0.$

(7) M_0 *the initial marking is a function defined on P,* $M_0(p) \in C(p)_{MS}\ \forall p \in P.$ ♦

We have the usual notions of a set of transitions being simultaneously firable for a marking, of the firing of a transition leading from one marking to another, and of the set of reachable markings.

A marking of CPN is a function defined on P, such that $M(p) \in C(p)_{MS}$ for all $p \in P$. A step of CPN is a function X defined on T, such that $X(t) \in C(t)_{MS}$ for all $t \in T$. The step X is enabled if and only if :

$$\forall p \in P : \sum_{t \in T} I_-(p,t)(X(t)) \leq M(p))$$

When X is enabled at M_1 it may occur and thus transform M_1 into a directly reachable marking M_2 defined by : $M_2 = (M_1 - I_-*X) + I_+*X$.

Asynchronous communications can be represented in a Petri net, by places used as buffer.

Another important point is the nondeterminism of transition firing : a firable transition needs not to be fired, when several transitions are firable simultaneously, any subset of these transitions can be fired.

This nondeterminism leads to the fact that we must deal with fairness. In considering infinite sequences of computations, we will consider only fair sequences, i.e. a transition cannot be firable an infinitie number of times, without being fired an infinitie number of times. This hypothesis corresponds to the hypothesis in the Actor model, that says that every message is eventually treated.

3 INFORMAL MODELING OF ACTORS BY CPN.

The basic ideas on how to translate an actor program into a CPN are resumed in the three following points :

1) An actor program is defined by two parts :

i) Behavior definitions,

ii) Initialisation.

An actor program evolves by creating actors and messages and by treating messages.

2) The static component of a CPN is determined by arcs, transitions and places, the dynamic component, is determined by tokens. A CPN evolves by moving tokens from one place to another place, by changing their structure, by producing some tokens, and by consuming others.

3) In order to simulate an actor program by a CPN, we will map the static component of the actor program onto the static component of the corresponding CPN, and the dynamic component of the actor program onto the dynamic component of the CPN. This leads to the conclusion that behaviors of an actor program are represented by subnets in the graph of the corresponding CPN, messages and actors are represented by tokens in the CPN. The colored Petri nets are very well adapted because the structure of the token can be complex. We will have several types of tokens, carrying different types of information.

We distinguish two kinds of tokens :

(i) actor-tokens corresponding to actors : their structure will include the address of an actor followed by its acquaintances list,

(ii) message-tokens corresponding to messages : their structure will include the address of the receiver , followed by the selector of the message and the contents of the message.

The behavior of the associated CPN is similar to an actor interpreter, which associates to every created

actor, a new address (color). This address enables to recognize an actor in the system. The evolution of an actor program corresponds to the evolution of actor-tokens and message-tokens.

The creation of a new actor with a specified behavior corresponds to the firing of a transition that puts a new actor-token (with a new address, which means that the number of colors must be unbounded) into a place of this behavior's subnet.

The change of behavior, corresponds to the firing of the transition transferring the actor-token from the subnet of the initial behavior into the subnet of the final behavior.

The emission of a message corresponds to the firing of a transition putting a new message-token into a special "communication place".

As appeared implicitly in [Agh86a], a preconditions to any execution is the presence of an active actor and a message sent to this actor in order to begin computation. So we think [SaVi90b], independently of [Eng90] that the begining of an execution of a script related to some message requires two tokens coming from two places.

The processing of a message corresponds to the firing of a transition that removes a message-token from the communication place and an actor-token from the place corresponding to its behavior. Note that this implies the asynchronous nature of message passing.

In order to construct the CPN modeling an actor program, we need to extract some information from the program. The analysis of the subpart "behavior definitions" of an actor program, enables us to deduce the following informations, for a given behavior :

- The behaviors into which an actor with a specified behavior can change,
- The messages to which an actor having this behavior can respond to,
- The messages that an actor having this behavior can send,
- The list of acquaintances associated to this behavior,
- The commands which may be executed by an actor having this behavior.

Now we explain with the help of examples, the interpretation in terms of CPN of the change of behavior, the creation of actor, and the treatment of messages.

3.1 CHANGE OF BEHAVIOR : The following example gives a counter behavior that can receive three types of messages, an incrementation, a request of value which is transmitted to a customer, or a request to become insensitive.

The change of behavior happens at the execution of a become command. Figure 1.b, gives the CPN associated to the actor program of Figure 1.a. Figure 1.b shows that each behavior corresponds to a subnet, and that the change of behavior is interpreted by a transition transferring an actor token from a place in the subnet associated with the initial behavior, to a place in the subnet associated with the final behavior. The subnet corresponding to the behavior "insensitive" is not shown, due to lack of space.

The enabling of this transition depends on the presence of the message-token and the actor-token to which this message is transmitted. The Figure 1.b shows the behavior of a serial actor, (which contains a become command).

The creation of a replacement machine is make by the become command in the case of messages "incr" and "ins", and in the case of "value" the actor-token is restituted as soon as it is taken. In the case of Figure 2.a, where the actor is unserial, there always exists a replacement machine available to process the next coming message.

```
def counter (n)
        [case operation of
            incr:()
            value : (cust)
            ins : ()
        end case]
    if operation = incr then become counter(n+1) fi
    if operation = value then send (print,n) to cust fi
    if operation = ins then become insensitif (n) fi
end def

def insensitive (n)
        [case operation of
            incr : ()
            value : (cust)
            ins : ()
        end case]
    if operation = incr then fi
    if operation = value then send (print,n) to cust fi
    if operation = ins then fi
end def

let x = new counter (0)
{send incr to x}
```

Figure 1.a

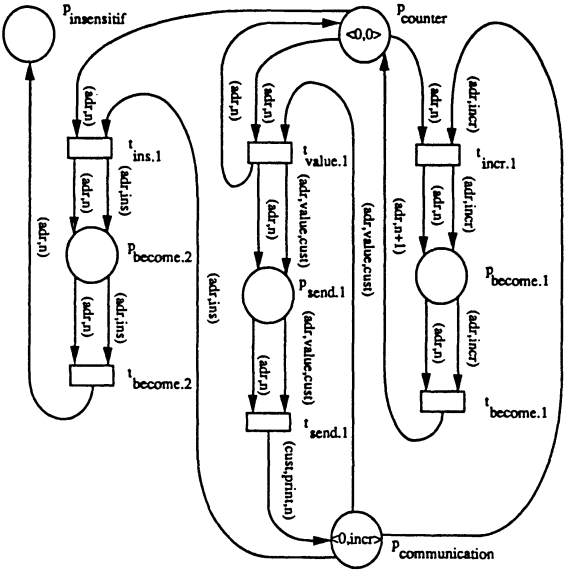


Figure 1.b

3.2 CREATION OF AN ACTOR : The actor program of Figure 2.a behaves in the following way: The behavior f with acquintance list (a) computes either a*u or a*u +b. In the first case it receives u and v, it does the computation and send the result to v, which has address 0 and the behavior printer (which is supposed to be written elsewhere), otherwise, it receives u,b and v, it computes a*u, and then creates an actor (with behavior g and acquaintance list (b)) to which it sends a*u and v, this actor will add b to a*u and sends the result to v.

```
def f(a)
        [case operation of
            op1 : (u,v)
            op2 : (u,b,v)
        end case]
    if operation = op1 then send (print,a*u) to v fi
    if operation = op2 then
        let y = new g(b)
        {send (op3,a*u,v) to y}
    fi
end def

def g(b)
        [case operation of
            op3 : (prod,v)
        end case]
    if operation = op3 then send (print,prod+b) to v fi
end def

let z = new f(2)
    {send (op2,3,4,printer)}
```

Figure 2.a

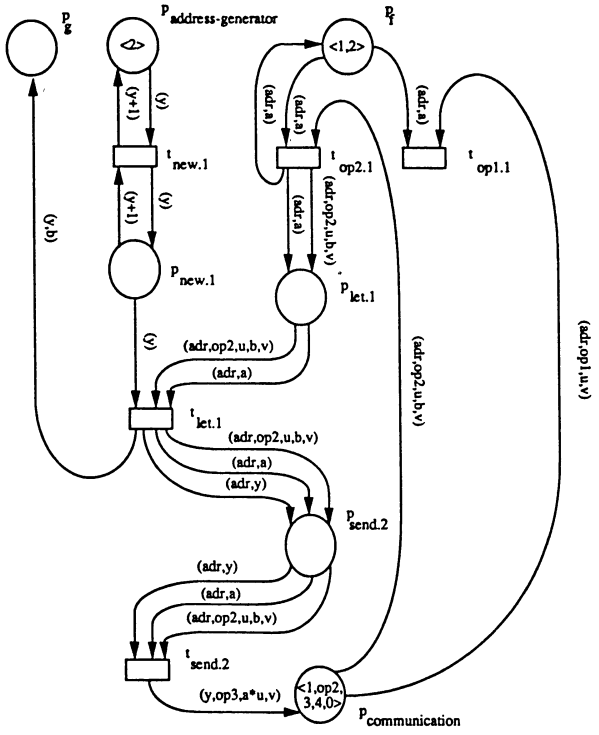


Figure 2.b

The creation of an actor happens at the execution of the let command. Let A be an actor having behavior B. We can deduce the behaviors of actors created by A by analysing B. When constructing the CPN shown in Figure 2.b, we have a transition from the subnet of the behavior of the creator actor to the subnet of the behavior of the created actor. The firing of this transition produces an actor-token in the subnet of the behavior of the created actor, due to lack of space, in Figure 2.b we do not give the subnet corresponding of the behavior "g", and the subnet corresponding to the treatment of message op1.

3.3 TREATMENT OF A MESSAGE : Message-tokens are put into a special "communication place" that acts as a buffer. This is made possible because of arrival order nondeterminism of messages to actors. Communication place play the role of forwarder to other actors.

The transition corresponding to the treatment of a message by a script, will be made firable by the message-token (which is constituted by an address, a message selector, and the remainder of communication list), with the actor-token to which a massage-token is addressed.

Note that the communication place is indispensable, since the emission and the treatment of a message are two distinct actions.

The place of communications, shown in Figures 1.b and 2.b, may have some incoming transitions corresponding to messages transmitted, and some coming transitions corresponding to messages received. Emission of a message is translated by the production of a message-token in the communication place, the firing of the transition "send" in Figure 1.b. The treatment of a message corresponds to the firing of transition $t_{value;1}$, or $t_{ins.1}$, or $t_{incr.1}$ in Figure1.b.

Note that the communication place and the address-generator place simulate the behavior of a mail system : every message sent is received, arrival order is nondeterministic.

Note that a way of constructing the CPN impose that the treatment of a message is not done in one step. We can represent this property by adding auxilliary places and transitions, in such a way that between the first and the last transition of the treatment of a message no other transition is firable.

4 CONSEQUENCES FOR THE SIMULATION OF DYNAMIC SYSTEM BY STATIC ONE

Let us consider a net associated with an actor program, for example the net in Figure 1.b.
We see that in this net messages and actors are treated in the same way, i.e. as tokens created or used by the subnets that represent the different behaviors and scripts.

This allows us to view an actor program as a set of static processes each corresponding to a different behavior , and with, in addition one specialised process acting as a mailbox (the communication place), and one specialised process used to generate new adresses for new actors (the adress generator place). Interactions between these processes are also static, and correspond to the different transitions that represent the different become, let or send commands. A process works after receiving the message which has the status message in an actor program, and a message which has the status actor in an actor program, moreover it works in an environment defined partly by the information contained in these two components.

In this view *both actors and messages of an actor program are treated in the same way, as messages exchanged between static processes.*

More generally, our method of using CPN gives a mean of representing dynamic systems by static ones, and therefore leads to :

- The possibility of using trace theory (which is typically linked to a system with a fixed number of processes),
- A method for implementing actors languages using OCCAM. Each behavior being implemented by one (or several if one wants concurrent execution of actors having the same behavior) process.

5 OTHER WAYS OF MODELING

5.1 BEHAVIORS AS ACTORS : Note that in deriving a CPN, we have add implicitely some behaviors that do not exist in an actor program which we model, i.e.

- The behavior of the communication place, which consists in buffering messages of actor system,
- The behavior of the address-generator which consists in generating a different address at each actor creation.

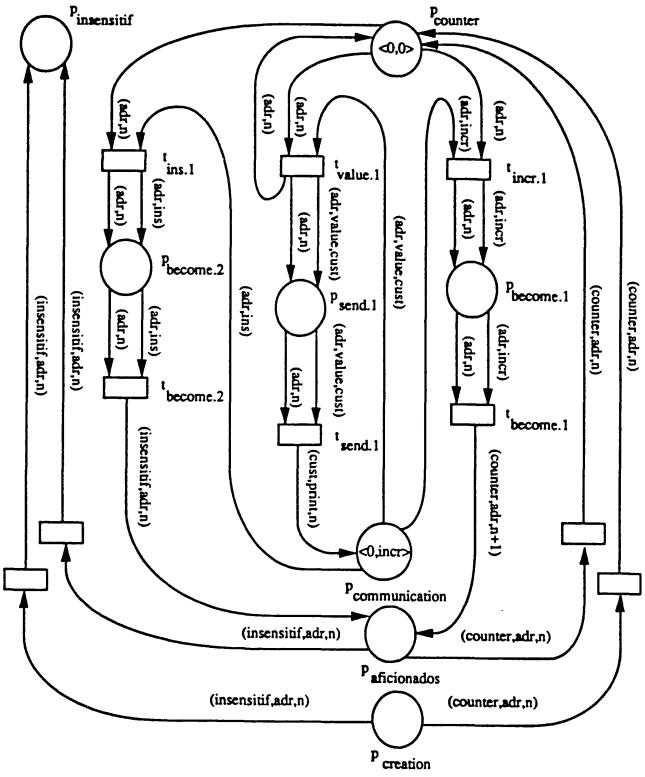


Figure 1.c

Although these behaviors do not exist in an actor program, they exist in an actor interpreter, and are necessary to the functionning of an actor program. So we can add two places one called "aficionados"

[Agh86a] for side effects and another called creation place. The "aficionados" place allows us to model the full version of SAL, as described in chapter3 of [Agh86a], we would not have to foresee a transition between a behavior B1 and B2, where "becomeB2" appears in B1.
We have the same phenomenon for the creation of actors. As an example see Figure 1.c associated to the actor program of Figure1.a.

5.2 SPLITTING COMMUNICATION PLACE : The communication place which appeared in Figure1.b and Figure2.b, can be splitted in order to attribute to each type of message one place. Transitions going to each place represent the emission of messages whose kind is associated with this place, and transitions coming from this place represent receptions of messages with this kind.

5.3 MESSAGES AS ACTORS : The association of one place to one type of message, leads to the approach of the universe of actors model described in [Agh86a], where expressions, commands and communications are themselves actors. So if we maintain also communication place this allows us to have messages which themselves can receive and send messages like what is mentioned in [Eng90].

6 FORMAL MODELING OF ACTORS BY CPN

6.1 ANALYSIS OF AN ACTOR PROGRAM : In order to obtain a net from an actor program, we need to have a structural description of the program, related more to actions than to data. We speak about commands and environment in which these commands are executed.

From a given actor program AP it is possible, in a syntactic way, to obtain the following sets : B, IE, CR, CW, IM, II. In the following we define all these components.

1) B : Set of behaviors in AP

B={b/b is a behavior in AP (b is represented by an identifier)}

For the "Counter" actor program, B={counter,insensitif}

2) IE : is the initial environment, which consists in a set of couples, (behavior, the corresponding acquaintance list). An acquaintance list is a sequence of identifiers, these identifiers represent addresses of actors or values.

IE={(b,al)/b is a behavior in AP∧al is an acquaintance list attached to this behavior}

For the "Counter" actor program, IE={(counter,n),(insensitif,n)}

3) CR : This means the reaction capacity of AP, it associates to each behavior a set of messages, which it may receive, and react to. These messages consist in a sequence of identifiers, where the first element is a selector which identifies a message in AP, the rest consists in a list of identifiers which represent (like in an acquaintance list), a set of addresses and values.

CR={(b,Smr)/b∈ B∧Smr={mr/b can react to mr }}

For the actor program of Figure 1.a, CR={(counter,incr,(value,cust),ins),(insensitif,incr,(value,cust),ins)}

4) CW : means the work capacity, it is the most important part, and needs several intermediate definitions. CW consists in a list of elements, each element gives for each behavior, and for each message the instructions of the corresponding script. In a way it is a copy of the program, but with informations added that enable us to derive a CPN in an automatic way.

CW={(b,mr,Sinst)/mr∈ Smr∧(b,Smr)∈ CR∧Sinst={inst /inst is an instruction in the script for the treatment of mr in b}

Given a behavior b, a message mr to which this behavior can react to, an instruction inst which is among the instructions which an actor with behavior b executes when he receives mr, let us define Ty, View, Env, Contain.

Definition 2 : *Ty(inst) denotes the type of the instruction inst; Ty (inst)∈{**send**,**become**,**let**,**if**}.*

*- If Ty(inst)=**send** then*

*inst=(**send**,View(ms'[1],(b,mr,inst)),a)∧∃b'∈B/(ms'∈Smr'∧(b',Smr')∈CR)∧a∈Env(b,mr,inst)*

(since an instruction is a sequence, inst[1] refers to the first constituent of inst)

(View is defined in Definition 2, Env is defined in Definition 4)

*- if Ty(inst)=**become** then inst=(**become**,b',View(al',(b,mr,inst)))∧(b',al')∈IE*

*- if Ty(inst)=**let** then inst=(**let**,SE,SI)∧SE={inst'/inst'=(x,(new,b',View(al',(b,mr,inst'))))/(b',al')∈IE∧x is an identifier} and SI={inst"/Ty(inst")∈{**send**,**become**,**if**,**let**}}*

*- if Ty(inst)=**if** then inst=(**if**,boolexp,SI1,SI2) where boolexp is a boolean expression∧*

*SI1,SI2={inst /inst∈{**send**,**become**,**if**,**let**}}*

SI1 is a packet of instructions to be executed if boolexp is true in another case the packet SI2 is executed ♦

Definition 3 : *We spoke about the View of an acquaintance list, and we spoke about the View of a selector (not communication list) because a selector may exist in more than one behavior. View is defined according to the three cases where we used it :*

*- if inst=(**send**,View(ms'[1],(b,mr,inst)),a) then View(ms'[1],(b,mr,inst)) is the sequence of identifiers or expressions used in inst to define ms.*

*- if inst=(**become**,b',View(al',(b,mr,inst))) then View(al',(b,mr,inst)) is the sequence of identifiers or expressions used in inst to define al'.*

*- if inst)=(**let**,SE,SI), and inst' ∈ inst[2] with inst'=(x,(new,b',View(al',(b,mr,inst)) then View(al',(b,mr,inst')) is the sequence of identifiers or expressions used in inst to define al'* ♦

For example in the actor program of Figure 2.a we have :

View (op3, (f,(op2,u,b,v),(**send**,(op3,a*u,v),p))) = (op3,a*u,v)

View((b), (f,(op2,u,b,v), (**let**,((p,(new,g,(b)))), ((send,(op3,a*u,v),y))))))=(b)

In the actor program of Figure1.a we have : View((n),(counter,(incr),(**become**,counter,(n+1))))=(n+1)

Definition 4 : *Env associates to each instruction in an actor program an environment in which this instruction operates:*

*- if Ty(inst)∈{**send**,**become**,**if**}and ∄inst'∈Sinst/(b,mr,Sinst)∈CW∧Ty(inst')=**let**∧inst ∈Contain(inst') then Env(b,mr,inst)=al∪mr/(b,al)∈IE.*

*- if Ty(inst)=**let** and ∄inst'∈Sinst /(b,mr,Sinst)∈CW∧Ty(inst')=**let** ∧ inst∈Contain(inst') then Env(b,mr,inst)=al∪mr∪{x/x=inst"[1]∧inst"∈inst[2]}/(b,al)∈IE.*

*- if Ty(inst)∈{**send**,**become**,**if**}∧∃inst'∈Sinst/(b,mr,Sinst)∈CW∧(Ty(inst')=**let** ∧ inst ∈ Contain(inst'))∧∄inst"/Ty(inst")=**let** ∧inst"∈Contain(inst')∧inst∈Contain(inst")) then Env(b,mr,inst)=Env(b,mr,inst').*

*- if Ty(inst)∈{**let**}∧∃inst'∈Sinst/(b,mr,Sinst)∈CW∧(Ty(inst')=**let**∧inst∈Contain(inst'))∧∄inst"/Ty(inst")=**let**∧inst"∈Contain(inst')∧inst∈Contain(inst"))then*

Env(b,mr,inst)=Env(b,mr,inst')∪{x/x=inst1[1]∧inst1∈inst[2]} ♦

Definition5 : *Contain is defined recursively.*

*- if Ty(inst) ∈{**become**,**send**} then Contain(inst) =inst,*

*- if Ty(inst) = **let**, and inst = (let,SE,SI) then Contain(inst) = Contain(SI),*

- *if Ty(inst) = if, and inst = (if ,SI1,SI2) then Contain(inst) = Contain(SI1)∪Contain(SI2)* ♦

For the "Counter" actor program,

CW={(counter,incr,(become,counter,n+1)),(counter,(value,cust),(send,n,cust)),(counter,ins,(become,n,insensistif)),(insensitif,incr, ø),(insensisitif,(value,cust),(send,n,cust))(insensitif,ins, ø)}

5) IM (initial mapping) : This function maps, some actors to behaviors.

IM = {(b,add,Value(al)) / (b,al) ∈ IE ∧ add ∈ IN}

(We relate each actor to an address. This address consists in a natural number. Therefore zero is related to the first actor created, and every creation increment add by one. Value(al) consists in instanciation of al)

For the "Counter" actor program, IM = { (counter,0,0)}

6) II (initial invoking) : This function maps the actors initially created to messages

II = { (add,Value(cl)) / (b,add,Value(al)) ∈ IM ∧ cl ∈ Smr / (b,Smr) ∈ CR}

For the "Counter" actor, II = { (0,incr) }

6.2 ALGORITHM OF DERIVATION : The derived CPN from AP is defined by (P,T,C,I_-,I_+,M_0), in the following we explain the way to obtain all these components,

1) $P=P_B \cup P_{CW} \cup P_{CEE} \cup P_{address\text{-}generator} \cup P_{communication}$

P_B is a set of places where each place corresponds to a behavior. There exists a bijective correspondance f between B and P_B. $P_B=\{f(b)=p_b / b \in B\}$.

P_{CW} is a set of places where each place corresponds to an instruction related to a behavior and a message to which this behavior can react to. There exists a bijective correspondance g between CW and P_{CW}. $P_{CW}=\{g(inst)=p_{Ty(inst).i} / inst \in Contain(Sinst) \wedge (b,mr,Sinst) \in CW \wedge i \in NI\}$.

In order to create an actor during the execution, we have to generate an address associated to it.

P_{CEE} is a set of places related to a set of creations which leads to the extension of environment in a let command. There exists a correspondance u between CW and P_{CEE}.

$P_{CEE}=\{u(inst)=p_{new.i} / inst \in SE \wedge (\textbf{let},SE,SI) \in Contain(Sinst) \wedge (b,mr,Sinst) \in CW \wedge i \in NI\}$

$P_{address\text{-}generator}$ is a set constituted by a place of generating addresses.

Note that if $P_{CEE} = \emptyset$ then $P_{address\text{-}generator} = \emptyset$

$P_{communication}$ is a set constituted by a place of communication.

2) Transitions represent a set of messages to which AP can react to, a set of instructions related to each message, a set of instructions contained in a let command, a set of instructions contained in the two branchs of the conditional command and a set of instructions related to all creations appeared in a let command. $T=T_{CR} \cup T_{CW} \cup T_{CEE} \cup T_{IC}$

T_{CR} is a set of transitions in which each transition is related to a message to which a behavior can react to. There exists a bijective correspondance f' between CR and T_{CR}.

$T_{CR}=\{f'(mr)=t_{mr[1].i} / mr \in Smr \wedge (b,Smr) \in CR \wedge i \in NI\}$.

T_{CW} is a set of transitions in which each transition is related to an instruction contained in a packet of instructions related to a specified message to which a behavior can react to. There exists a correspondance g' between CW and T_{CW}.

$T_{CW}=\{g'(inst)=t_{Ty(inst).i} / inst \in Contain(Sinst) \wedge (b,mr,Sinst) \in CW \wedge Ty(inst) \neq \textbf{if} \wedge i \in NI\}$.

T_{CEE} is a set of transitions in which each transition is related to a creation appeared in a let command. There exists a correspondance u' between CW and T_{CEE}.

T_{CEE}={u'(inst)=$t_{new.i}$/inst∈ SE∧(**let**,SE,SI)∈ Contain(Sinst)∧(b,mr,Sinst)∈ CW∧i∈ NI}

T_{IC} is a set of couples of transitions in which each couple is related to the two branchs of conditional command. There exists a correspondance v' between CW and T_{IC}.

T_{IC} = {v'(inst)=($t_{then.i}$,$t_{else.i}$)/inst∈ Contain(Sinst)∧(b,mr,Sinst)∈ CW∧Ty(inst)=**if**∧i∈ NI}

3) The colors of places and transitions are defined as follows : by an abuse of notation, instead of a color of a place p or transition t as defined in Definition 1, we give the structure of tokens that may be contained in t or p. When more than one structure exists, we give the sum of the different structures. "adr" represents an address of an actor which is an integer, the first element of a communication list (cl) is a string and as supposed in [Agh86] other components of al, cl or environment of a given instruction represent addresses of actors, strings, integer or whatever. From this we can easily deduce the colors for a place or a transition.We note Env'(b,mr,inst)=Env(b,mr,inst)-{al,cl} and if Env'(b,mr,inst)=Ø then (adr,Env'(b,mr,inst)) is reduced to nil.

If p∈ P_B, C(p)=(adr,al)/(b,al)∈ IE∧p=f(b)

If p=$P_{communication}$, C(p)=(adr,cl)/cl∈ Smr∧(b,Smr)∈ CR

If p∈ $P_{address\text{-}generator}$ ∪P_{CEE}, C(p)=(adr)

If p∈ P_{CW}, ∃inst∈ Sinst ∧(b,mr,Sinst)∈ CW∧(b,al)∈ IE/

p=g(inst)∧C(p)=(adr,al)+(adr,mr)+(adr,Env'(b,mr,inst))

If t ∈ T_{CR}, ∃mr∈ Smr ∧(b,Smr)∈ CR∧(b,al)∈ IE/t=f'(inst)∧C(t)=(adr,al)+(adr,mr)

If t∈ T_{CW}, ∃inst∈ Sinst∧(b,mr,Sinst)∈ CW∧(b,al)∈ IE/

t=g'(inst)∧C(t)={(adr,al)+(adr,mr)+(adr,Env'(b,mr,inst))

If t ∈ T_{CEE}, C(t)=(adr)

If t ∈ T_{IC},∃ inst∈ Sinst∧inst=(If,boolexp,SI1,SI2)∧(b,mr,Sinst)∈ CW∧(b,al)∈ IE/(t',t")=v'(inst)∧

C(t')=(adr,al)+(adr,mr)+(adr,Env'(b,mr,inst)/boolexp=true}∧

C(t")=(adr,al)+(adr,mr)+(adr,Env'(b,mr,inst)/boolexp=false

4) The negative incidence matrice is defined as follows : by an abuse of notation, in this section and in 5) we do not give functions as elements of this matrice but expressions that appeared in the CPN graph. If several arcs from p to t exist, we give the sum of expressions appeared on these arcs. In [Jen86] it is shown how to obtain the corresponding functions. Note that in our derivation we choose to distribute the environment of an active actor in three tokens (actor, message and sometimes the extension) these structures begin with the address of an actor.The manner in which information is saved in tokens may be refined.

The processing of a message corresponds to the firing of a transition that removes a message-token from the communication place and an actor-token from the place corresponding to its behavior.

∀(b,Smr)∈ CR∧∀mr∈ Smr,∃p=f(b),t=f'(mr),(b,al)∈ IE/I_(p,t)=(adr,al)∧I_($p_{communication}$,t)=(adr,mr)

Note that with CPN we can easily choose an environment that we need to broadcast, and easily extend and reduce this environment, essentially when this environment is supported by tokens. So we broadcast acquaintance list and communication list in all commands that appeared in behaviors.

∀(b,mr,Sinst)∈ CW∧(∀inst∈ Contain(Sinst)/Ty(inst)≠**let**),∃p=g(inst),t=g'(inst),(b,al)∈ IE/

I_(p,t)=(adr,al)+(adr,mr)+(adr,Env'(b,mr,inst))

In the case of a let command we associate to the instruction the environment existing before extension.

∀(b,mr,Sinst)∈ CW∧(∀inst∈ Contain(Sinst)/inst=(**let**,SE,SI)),∃p=g(inst),t=g'(inst),(b,al)∈ IE/

I_(p,t)=(adr,al)+(adr,mr)+(adr,Env'(b,mr,inst)-{x/x=inst'[1]∧inst'∈ SE})

To every creation, that appeared in a let command we have a transition that takes the last address created, in the address generator and increments it.

$\forall(b,mr,Sinst)\in CW\wedge(\forall inst\in SE/(\textbf{let},SE,SI)\in Contain(Sinst)),\exists p\in P_{CEE}/p=u(inst),\exists t\in T_{CW}/$
$t=g'((\textbf{let},SE,SI)),\exists t'\in T_{CEE}/t'=u'(inst)/I_{-}(p,t)=inst[1]\wedge I_{-}(P_{address\text{-}generator},t')=inst[1]\wedge$
$I_{-}(p,t')=inst[1]+1$

In a conditional command the environment of this instruction is transmitted to the two branchs.

$\forall(b,mr,Sinst)\in CW\wedge(\forall inst\in Contain(Sinst)/Ty(inst)=\textbf{if}),\exists p\in P_{CW}/p=v(inst),\exists(t,t')\in T_{IC}/(t,t')=v'(inst),/$
$I_{-}(p,t)=I_{-}(p,t')=((adr,al)+(adr,mr)+(adr,Env'(b,mr,inst))$

5) The positive incidence matrice is defined as follows : if a script attached to some message and some behavior does not present a become command, we have to foresee a default behavior.

$\forall(b,mr,Sinst)\in CW\wedge(\nexists inst\in Contain(Sinst)/Ty(inst)=\textbf{become}),\exists p=f(b),t=g'(mr),(b,al)\in IE/$
$I_{+}(t,p)=(adr,al)$

As we said before, the change of a behavior is made by displacing an actor-token from an initial behavior to a final behavior.

$\forall(b,mr,Sinst)\in CW\wedge(\exists inst\in Contain(Sinst)/inst=(become,b',View(al',(b,mr,inst)))),\exists p'=f(b'),t'=g'(inst)/$
$I_{+}(t',p')=(adr,View(al',(b,mr,inst)))$

The acquaintance list and communication list have to be transmitted to the commands contained in a top of each script. A let command has to transmit its environment to all commands it contain, the conditional command has to transmit its environment to all commands contained in the two branchs.

$\forall(b,mr,Sinst)\in CW\wedge(\forall inst\in Contain(Sinst)/(\nexists inst'\in Contain(Sinst)\wedge inst\in Contain(inst'))),\exists p=g(inst),$
$t=f'(mr),\ (b,al)\in IE/I_{+}(t,p)=(adr,al)\cup(adr,mr)$

$(\forall inst\in Contain(Sinst)/(b,mr,Sinst)\in CW\wedge(b,al)\in IE\wedge inst=(\textbf{if},boolexp,SI1,SI2)))\wedge$
$(\forall inst'\in SI1\cup SI2),\ \exists p\in P_{CW}/p=g(inst'),\exists(t_1,t_2)\in T_{CW}/(t_1,t_2)=v'(inst)/$
$I_{+}(t_1,p)=I_{+}(t_2,p)=(adr,al)+(adr,mr)+(adr,Env'(b,mr,inst))$

$(\forall inst\in Contain(Sinst)/(b,mr,Sinst)\in CW\wedge(b,al)\in IE\wedge inst=(\textbf{let},SE,SI))\wedge(\forall inst'\in SI),$
$(\exists p\in P_{CW}/p=g(inst'))\wedge(\exists t\in T_{CW}/t=g'(inst))\ /I_{+}(t,p)=(adr,al)+(adr,mr)+(adr,Env'(b,mr,inst))$

We interpret a send command by putting a message into the communication place.

$\forall(b,mr,Sinst)\in CW\wedge\forall inst=(\textbf{send},View(ms'[1],(b,mr,inst)),a)\in Contain(Sinst),\exists t=g'(inst)/$
$I_{+}(t,p_{communication})=(a,View(ms'[1],(b,mr,inst)))$

As we said before the creation of an actor with some behavior is done by putting the actor token into the subnet related to this behavior

$(\forall inst\in Contain(Sinst)/(b,mr,Sinst)\in CW\wedge inst=(\textbf{let},SE,SI))\wedge(\forall inst'\in SE/inst'=(x,(New,b',View(al',$
$(b,mr,inst)))),\exists p'=f(b'),t=g'(inst)/I_{+}(t,p')=(adr,View(al',(b,mr,inst)))\wedge$
$\exists p1\in P_{CEE}/p1=u(inst'),\exists t1\in T_{CEE}/t1=u'(inst')/I_{+}(t1,p1)=inst'[1]\wedge I_{+}(t1,P_{address\text{-}generator})=inst'[1]+1$

6) $\forall(b,add,Value(al))\in IM,\ \exists p\in P_B/p=f(b)\wedge(add,Value(al))\in M_0(p)$

$M_0(P_{address\ generator})=/IM/$ and $M_0(P_{communication\ place})=II$

6.3 DETAILLED SKETCH OF PROOF : As shown in [Agh84], [Agh86a], an actor system may be described in terms of its possible configurations, a configuration is defined in terms of local states function and a set of unprocessed tasks. A possible transition allows change from one configuration to another. In order to guarantee the property of fairness, the notion of a subsequent transition is introduced. All these

notions are found in [Agh86a].

As envisaged in [SaVi91a] in order to prove that the derived colored Petri net cpn=(P,T,C,I_-,I_+,M_0) reflects the behavior of the corresponding Actor program AP=(B,IE,CR,CW,IM,II), we establish a correspondance between G_{AP} : the graph of the evolution of configurations of AP and G_{cpn} : the markings graph of cpn. In the first graph, vertices represent configurations of AP and edges represent possible tasks. In the second graph, vertices represent markings where no tasks is under execution, and edges represent the sequence of transitions of cpn corresponding to the full treatment of one message. A difficulty arises from the fact that the full treatment of a message is not represented in cpn by an atomic action (although it could be with additional places and transitions as indicated in 3.1) . This is a difference with the semantic given in [Agh86a] where the treatment of a message is done in one step. Also, since colored Petri nets allow multisets, we will identify a set where two equivalent tasks have two different tags, with a multiset where the task appears twice and is represented only by a target and a message.

Lemma 1 : *To every configuration c=(L,T) of G_{AP} corresponds a single marking M_{cpn} of G_{cpn}* ♦

Proof : given a configuration in AP c=(L,T) where L:M⟶B, now we compute the corresponding M_{cpn} of cpn. In this proof and in the following, we use also Value which represents an instanciation, we do not specify this instanciation because the semantic given in [Agh86a] did not take in account the contain of local memory of an actor (its acquaintance list). The marking is constructed by the three following steps.

1) for actors :$\forall m \in M, (\forall b \in B / b=L(m)), \exists p_b \in P_B / p_b=f(b), \exists (b,al) \in IE / <m,Value(al)> \in M_{cpn}(p_b)$.

2) for tasks :$\forall T \in T / T=(m,k)$ then $<m, k> \in M_{cpn}(P_{communication})$

3) for addresses :|M| = n then $M_{cpn}(P_{address\text{-}generator}) = <n>$ ♦

Lemma 2 : *To every marking M_{cpn} of G_{cpn} corresponds a single configuration c=(L,T)* ♦

Proof : given a M_{cpn} in cpn, we compute the corresponding c=(L,T) of AP, where L : M⟶ *B* with the following steps.

1) for actors :$\forall p_b \in P_B$, $\forall <adr,Value(al)> \in M_{cpn}(p_b)$ then adr∈ M and L (adr)= b.

2) for tasks :$\forall <adr,Value(cl)> \in M_{cpn}(P_{communication}), \exists T \in T / T=(m,Value(cl))$ ♦

Lemma 3 : *Given a task T and two configurations c_1 and c_2 of G_{AP}, where $c_1 \_\_T\rightarrow c_2$. In G_{cpn}, there always exist a marking M_{cpn1} corresponding to a configuration c_1, a marking M_{cpn2} corresponding to a configuration c_2 and an enabled sequence of transitions T_{cpn} such that $M_{cpn1} \_\_T_{cpn}\rightarrow M_{cpn2}$* ♦

Proof : The construction of M_{cpn1}, M_{cpn2} from c_1, c_2 are explained in the proof of Lemma 5.1, we compute T_{cpn} which corresponds to *T* . If $T \in$ tasks(c_1) and $T=(m,k)$, $\exists$ b ∈ B / L(m) = b and (b,k,Sinst) ∈ CW and $T_{cpn} = T_{cr} \times T_{cw} \times T_{cee} \times T_{ic}$ / T_{cr} = { f'(k)} and

T_{cw} = {g'(inst) / inst ∈ Contain(Sinst) and Ty(inst)≠**if**} and

T_{cee} = {u'(inst) / inst ∈ SE and (**let**,SE,SI) ∈ Contain(Sinst) }and

T_{ic} = {v'(inst) / inst ∈ Contain(Sinst) and Ty(inst)=**if**} (where f', g', u' and v' are defined in 4.b) ♦

Lemma 4 : *Given a sequence of transitions and two markings of G_{cpn}, M_{cpn1} and M_{cpn2} where $M_{cpn1} \_\_T_{cpn}\rightarrow M_{cpn2}$. There always exists in AP a configuration c_1 corresponding to a marking M_{cpn1}, a configuration c_2 corresponding to a marking M_{cpn2} and a task T such that $c_1 \_\_T\rightarrow c_2$* ♦

Proof : The construction of c_1, c_2 from M_{cpn1}, M_{cpn2} are explained in the proof of Lemma 5.1, now we compute *T* which corresponds to T_{cpn}. If $<m,Value(cl)> \in M_{cpn1}(P_{communication})$, and $<m,Value(cl)> \notin M_{cpn2}(P_{communication})$, then there exist $T=(m,Value(cl))/T \in$ tasks(c_1) and

$c_1 \xrightarrow{T} c_2$ ♦

The lemmas 5.1, 5.2, 5.3, 5.4 imply the following Theorem.

Theorem 1: *There always exist an isomorphism between* G_{AP} *and* G_{cpn} ♦

This Theorem and the assumption like added in [Agh86a],[Agh84] (subsequent transition), that cpn operates in such a way that all transitions corresponding to the execution of tasks are subsequently enabled, imply that to every actor program exists a colored Petri net (obtained by the algorithm of derivation in 4.2 and where the execution of a message is made atomic) which reflects its behavior.

7 DISCUSSION

In [EnLeRo90] and [Eng90] actors are formalized by a model called POTs (a parallel object-based transition system), which is essentially a place transition net with an additional structure imposed on its places. An object in a given state is represented by a place and an event is represented by a transition (therefore one obtains an infintie net), the structure of the objects is represented by an additional structure of the places, formalized through the functions 'obj', 'mod' and 'acq', where

- 'obj' maps each place to the object it represents,
- 'mod' maps each place to the mode of the represented object (unborn, alive or dead),
- 'acq' maps each place to a set of acquaintances in the state of the object represented by the place.

In our model the role of obj, mod and acq are played by a token, so if an actor exists, a corresponding actor-token exists and its address and acquaintance list are contained in its structure. Moreover we have a finite net which allows us to give a static view to an actor program in which appears the exact design of an actor program, contrarily to actor event diagram [Clin81], to actor grammars [JaRo89] and to the modeling by POTs where just the evolution is expressed, without showing the design of an actor program. Moreover we express here dynamicity in Petri Nets without increasing the number of transitions and places, and without changing the flow relation.

Note that creation of an actor may be viewed as a call of procedure, the acquaintance list of an actor may be viewed as the local memory of a process, the become command as the change of this local memory. So following the conjecture suggested in [Agh84]; "every model of concurrent computation can be derived as a special case of an actor model", and the dicussion concerning supporting multiparadigm programming by actors in [Agh89] we can apply the derivation of CPN to other languages.

When we capture all the information contained in the actor program and draw the graph for the corresponding CPN, we remark that we exhibit all the possibilities of interaction in the sense of communication by "send" and in another sense by "become" and "new". We hope in a further step of this work to find a relation between the design of an actor program and the corresponding evolution, in order to find a good design which allows an evolution on some topologies with good properties like the Hypercube [SaSc88] or the De Bruijn and Kautz networks [BePe87] . This way we should be able to optimize the implementation of this type of languages.

Another direction of our work is to apply the tools existing for CPN, and obtain method to make some diagnosis on Actor programs.

ACKNOWLEDGMENT The authors would like to thank C.Johnen and a referee for helpful comments on this work.

REFERENCES

[Agh84] Gul A.Agha; Semantics considerations in the Actor Paradigm of Concurrent Computation, Seminar on Concurrency, LNCS 197, July 1984

[AgHe85] Gul A.Agha; Carl Hewitt; Concurrent Programming Using Actors : Exploiting Large-scale Parallelism, A.I.Memo No. 865, October 1985.

[Agh86a] Gul A.Agha; Actors : a model of concurrent computation in distributed systems, The MIT press, 1986.

[Agh86b] Gul A.Agha; An overview of actor languages, SIGPLAN Notices Vol 21, 1986.

[Agh89] Gul A.Agha; Supporting Multiparadigm Programming on Actor Architectures, Parle'89 Parallel Architectures and Languages Europe , Volume II, LNCS 366, E.Odjik, M.Rem, J-C.Syre, eds, 1989.

[AnSc83] Gregory R.Andrews, Fred B.Schneider; Concepts and Notations for Concurrent Programming, ACM Computing Surveys, Vol 15, No1, 1983.

[BaStTa89] Henri E. Bal, Jennifer G. Steiner, Andrew S. Tanenbaum; Programming Languages for Distributed Computing Systems, ACM Computing Surveys, Vol.21, No.3, September 1989.

[BaVi91] F.Baude, G.Vidal-Naquet; Actors as a parallel programming model, STACS91, LNCS 480, C.Choffrut, M.Jantzen, eds, 1991.

[BePe87] J-C.Bermond and and C. Peyrat; The De Bruijn and Kautz networks : a competitor for the hypercube ?; Hypercube and Distributed Computers, F.André and J.P. Verjus (Eds), Elseiver Science Publishers B.V. (North-Holland), 1989.

[Clin81] W.D.Clinger; Foundations of actors semantics, AI-TR-633, MIT artificial intelligence Laboratory, May 1981.

[EnLeRo90] J.Engelfriet, G.Leih, G.Rozenberg; Parallel object-based systems and petri nets, Technical Reports 90-04 and 90-05, Leiden University, 1990.

[Eng90] J.Engelfriet; Net-based description of parallel object-based systems, or POTs and POPs,Workshop on Foundations of Object-Oriented Languages, May28 - June1, 1990.

[Hew77] Carl Hewitt; Viewing control structures as patterns of passing messages, in Artificial Intelligence, An MIT Perspective, Brown & Winston, eds, 1977.

[HewBak77] C.Hewitt, H.Baker; Actors and continuous functionals, Formal Description of progr.Concepts ,1977.

[HeAtLi79] C.E Hewitt, G.Attardi, and H.Lieberman; Specifying and proving properties of guardians for distributed systems. In praceedings on semantics of concurrent computation, INRIA, Evian, France, 1979.

[HeReAgAt84] Carl Hewitt, Tom Reinhardt, Gul A.Agha, Giuesppe Attardi; linguistic support of receptionists for shared ressources, Memo MIT, 1984.

[Hew85] C.E.Hewitt; The challenge of open systems. Byte 10(4) : 223-242, 1985.

[Hoa78] C.A.R. Hoare; Communicating Sequential Processes; Communications from the ACM, Vol21, No8, 666-677, 1978.

[Hoa74] C.A.R. Hoare; Monitors : An Operating System Structuring Concept, Communications from the ACM, 17 (10), 549-557, 1974.

[HoKoRa89] R.P. Hopkins, M. Koutny, B.Randell; Some Results on Dynamically Structures Communicating Systems; Research Memorandum, University of Newcastle upon Tyne, 1989.

[JaRo89] D.Janssens, G.Rozenberg; Actor grammars, Math. Syst. Theory 22, 75-107, 1989.

[Jen86] Kurt Jensen, Colored petri nets, Advances in petri nets, Part1, LNCS 254, W.Brauer, W.Reisig and G.Rozenberg, eds, 1986.

[Jon86] Peter De Jong, Compilation into Actors, In sigplan notices, Vol 21, October 1986.

[LiHeGI86] Barbara Liskov, Maurice Herlihy, Lucy Gilbert; Limitations of Synchronous Communication With Static Process Structure in Languages for Distributed Computing, Proceedings of the 13th ACM Symposium on Principles of Programing Primitives, St. Petersburg, Florida, January 1986.

[SaSc88] Y.Saad, M.Shultz, Topological properties of Hypercubes, IEE Trans. on computers, vol 37, N° 7, July 1988.

[SaVi90a] Y.Sami, G.Vidal-Naquet, Formalisation of the behavior of actors by colored Petri nets and some applications, technical report 605, LRI, CNRS URA 410, 1990.

[SaVi90b] Y.Sami, G.Vidal-Naquet, talks given at DEMON working group meeting at Newcastle Mai 90 and DEMON AGM1, Paris, June 90.

Program Refinement in Fair Transition Systems

Ambuj K. Singh*
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, California 93106

Abstract

The idea of program refinements is discussed in the context of fair transition systems. Two kinds of refinements – property preserving and fixed-point preserving, are defined. Conditions are developed under which known program transformations (e.g., refinement of atomicity, abstract data type implementation) are property preserving and fixed-point preserving refinements. The usefulness of the developed theorems is illustrated through a number of examples.

1 Introduction

Concurrent program verification has been a prolific area over the last two decades. Beginning with Floyd's and Hoare's seminal papers on sequential program verification [8, 10], numerous logics [5, 7, 17, 19, 21] and methodologies [5, 6, 9, 12, 13, 22] have been proposed for concurrent programs. In spite of the elegance and the expressive power of the proposed theories, concurrent program verification remains a very difficult and complex task. There are two possible explanations for this complexity. First, most of the concurrent programs are unstructured and written in an ad-hoc way. This makes the separation of concerns next to impossible and consequently, the proofs are entangled. Second, most of the concurrent programs contain a lot of irrelevant information that has nothing to do with the underlying algorithm's correctness. As a result, it is far easier to prove the algorithms that the programs are based on than the programs themselves.

A possible solution to the aforementioned drawbacks of concurrent program verification is to obtain programs by a formal derivation process based on refinements. Such a formal derivation achieves the separation of concerns by postponing efficiency and architectural decisions until late in the design process. The initial stages of the design are concerned mainly with the development of the algorithm. Only after the main ideas have been developed and all algorithmic questions have been settled, the efficiency and architectural considerations enter into the picture. At that time, the program is mapped onto a given architecture through another series of refinements. This approach of formally deriving a program by an architecture-independent sequence of steps and then an architecture-dependent sequence of steps makes the resulting programs modular and easily transportable across architectures. It also allows the programmer to focus on the problem at hand without getting lost in the peculiarities of the given architecture.

*Work supported in part by NSF Grant CCR-9008628.

Formal program derivation can be roughly classified into two types: property refinement and program refinement. In property refinement, an individual property p that is a part of the specification is refined into a stronger property q that is less *abstract* or closer to a machine architecture. Thus, given a high-level specification S, each requirement in S is refined individually or collectively to another set of properties T that implies the specification S. In addition to being closer to the machine level, the properties in refinement T usually permit a division of the program into submodules each with its local set of properties. Program derivation through property refinement proceeds until it is clear as to how a program may be drawn up in order to meet the requirements [5]. Once this initial program has been drawn up, program derivation proceeds by program refinement. During property refinement, all properties of the program, not just those required in the initial specification, are preserved. Program refinement usually forms the architecture-dependent sequence of steps in the development of a program. As opposed to property refinement, which is a correctness preserving mapping in the semantic (or logical) domain, program refinement is a correctness preserving mapping in the syntactic (or programming language) domain. Both program and property refinements are useful in the development of programs – while property refinements are usually domain-specific and therefore, more useful in the initial stages of development, program refinements are more general-purpose (i.e., less domain-specific) and therefore, more useful in the later stages of program development. For example, property refinement may be used in developing a high-level solution strategy whereas program refinement may be used for addressing implementation issues such as process synchronization, abstract data type implementation, and scheduling of processes [25].

In this paper we address program refinements in the context of fair transition systems [18]. A fair transition system is an abstract computational model characterized by a set of variables, a set of states, a set of transitions, initial conditions, and fairness requirements. Fair transition systems encompass most existing systems and have been widely used for modeling reactive systems. For our purposes in this paper, we choose the specific framework of Unity [5] as the representative fair transition system. The main reason behind our choice is the simplicity of Unity's logic and the ease with which refinements can be expressed and proved in it. Even though we concentrate on one specific formal framework, most of our results translate equally well across any other framework based on fair transition systems.

Previous work on program derivation has focussed mainly on property refinements [5, 6, 9]. The existing work on program refinements has focussed more on proving the correctness or completeness of refinements [1, 13, 17] rather than on developing general purpose refinements for development of programs. Atomicity of statements is perhaps the only area for which some general purpose refinements have been proposed. Back develops a method for refining atomicity for sequential programs in [2] and extends it to terminating parallel programs in [3, 4]. Recently, Lamport and Schneider have extended Lipton's work [16] and developed conditions for preservation of safety properties in program refinements [14]. However, refinements of atomicity that preserve both safety and progress properties in the context of reactive programs have not been explored elsewhere.

Program refinements can be broadly divided into two classes: those that preserve all properties (safety and progress), and others that preserve the *fixed-point* (i.e., if the original program terminates, then so does the refined program, and the final state of refined

program implies the final state of the original program). The first kind of refinement is useful for interactive programs whereas the second kind is useful for terminating programs. Program refinements have been used successfully for developing a solution to the pin alignment problem for DNA strings [25]. In that paper, a simple architecture-independent solution to the problem is first developed. Later, this solution is mapped to a shared-memory architecture through a series of program refinements. These refinements are based on some of the theorems reported here and address issues such as synchronization of shared data, implementation of abstract data types, and scheduling of processes. Because the refinements are correctness preserving, the correctness of the final refined program is based solely on the correctness of the original simple program.

In this paper, we address both property preserving and fixed-point preserving refinements. We state and prove a number of properties and theorems about them. These refinements deal with issues such as implementation of abstract objects, strengthening of guards, implementation of shared variables by asynchronous channels, and refinement of atomicity. We show the applicability of these refinements through a number of small examples. Though the presented refinements are individually quite simple, together they can be quite a powerful tool in the development of programs [25].

Recently, Sanders [23] has proposed a mixed specification language (similar to Lamport's transition axiom method [15]) for stepwise development of parallel programs. In this method, a program is specified by a set of assignment statements and a set of explicit program properties. Any fair execution of the assignment statements that satisfies the explicit program properties is an acceptable behavior of the program. She defines the idea of a refinement in the proposed language and presents theorems about the preservation of implicit progress properties. The results presented here differ from her work in that we focus on developing conditions under which useful program refinements such as refinement of atomicity and strengthening of guards preserve the desired properties of a program.

The rest of the paper is organized as follows. Section 2 discusses Unity and presents a brief introduction to its logic. Section 3 defines program refinements in Unity and presents some of their basic properties. Section 4 contains a number of theorems about program refinements and examples illustrating their application. Finally, Section 5 includes concluding remarks.

2 A Brief Introduction to Unity

We discuss the syntax of Unity in Section 2.1 and the logic of Unity along with program compositions in Section 2.2.

2.1 The Unity Syntax

A Unity program consists of four sections — a *declare* section that declares the variables used in the program, an *always* section that consists of a set of proper equations, an *initially* section that describes the initial values of the variables, and an *assign* section that consists of a non-empty set of assignment statements. An assignment statement consists of one or more assignment components separated by ‖. An assignment component is either an enumerated assignment or a quantified assignment. An enumerated assignment has a variable list on the left, a corresponding expression list in the middle,

and a boolean expression on the right called the *guard* (which by default is *true*):

$< \textit{variable-list} > := < \textit{expression-list} > \ \textit{if} \ < \textit{guard} >$.

A quantified assignment specifies a quantification and an assignment that is to be instantiated with the given quantification; a quantification names a set of bound variables and a boolean expression (the range) satisfied by the instances of the bound variables. An assignment component is executed by first evaluating all expressions and then assigning the values of the evaluated expressions to the appropriate variables, if the associated boolean expression is *true*; otherwise, the variables are left unchanged.

The set of assignment statements in the *assign* section is written down either by enumerating every statement singly and using $[]$ as the set constructor, or by using a quantification of the form $\langle [] \ var : range : statement \rangle$. Symbol $[]$ is called the Union operator.

A program execution starts from any state satisfying the initial conditions and goes on forever; in each step of execution some assignment statement is selected nondeterministically and executed. Nondeterministic selection is constrained by the following fairness rule: every statement is selected infinitely often [5].

Example 1: The following program sorts integer array $A[0..N]$, $N \geq 0$, in ascending order by swapping adjacent elements if they are out of order. Its *assign* section consists of N statements, one for every pair of adjacent positions.

Program *sort*
  assign
    $\langle [] \ i : 0 \leq i < N :: A[i], A[i+1] := A[i+1], A[i] \ \textit{if} \ A[i] > A[i+1] \rangle$
end □

Notation: The fixed-point of a program, usually represented by FP, describes the state of the program upon termination; it is obtained by replacing the assignment symbol $:=$ by the equality symbol $=$ in every statement of the program and taking the conjunction over all such predicates. For example, the fixed point of program *sort* is

$(\forall i :: A[i] > A[i+1] \Rightarrow A[i] = A[i+1])$,

which simplifies to $(\forall i :: A[i] \leq A[i+1])$. □

2.2 The Unity Logic

Program properties are expressed using four relations on predicates — *unless, invariant, ensures,* and *leads-to*. The first two are used for stating safety properties whereas the last two are used for stating progress properties.

Unless: For any two predicates p and q, the property p *unless* q holds in a program iff for all statements s in the program the following Hoare triple [10] holds

$\{p \wedge \neg q\} \ s \ \{p \vee q\}$.

Informally, if p is *true* at some point in the computation, then either q never holds and p holds forever from this point on, or q holds eventually and p continues to hold until q holds.

Invariant: For any predicate p, the property *invariant* p holds in a program iff p holds initially and the program never falsifies p, i.e.,

initially p $\wedge$ p *unless false*.

An invariant can be substituted for *true* and vice-versa in the context of a program (this includes both the proof and the text of the program). This is referred to as the Substitution axiom. Sometimes, for *invariant* p, we simply write p.

Ensures: For any two predicates, p and q, the property p *ensures* q holds in a program iff p *unless* q holds in the program and there exists a statement s in the program such that

$$\{p \wedge \neg q\}\ s\ \{q\}.$$

Thus, if p is *true* at some point in the computation then q holds eventually and p continues to hold until q holds. Statement s that establishes q is called the helpful statement.

Leads-to: The relation *leads-to* is denoted as $\mapsto$, and is defined to be the strongest relation satisfying the following three rules.

- p *ensures* $q \ \Rightarrow\ p \mapsto q$,
- $(p \mapsto q \ \wedge\ q \mapsto r) \ \Rightarrow\ p \mapsto r$, and
- For any set W,
  $(\forall m : m \in W : p.m \mapsto q) \ \Rightarrow\ ((\exists m : m \in W : p.m) \mapsto q)$.

The first two rules imply that $\mapsto$ includes the transitive closure of *ensures* and the third rule allows us to induct over sets. Given that $p \mapsto q$ in a program, we can assert that once p becomes *true*, eventually q becomes *true*. However, unlike p *ensures* q, we cannot assert that p will remain *true* as long as q is *false*.

2.2.1 Program Composition by Union

Let F and G be programs with compatible *declare* sections (i.e., the declaration of the variables are non-conflicting), compatible *always* sections (i.e., the two sets of equations are consistent), and compatible *initially* sections (i.e., the initial values of the variables are non-conflicting). Then, their composition is a new program denoted $F \parallel G$; every section of this program is obtained by a union of the corresponding sections of F and G. The following theorem, called the Union theorem, follows from the definitions of *unless* and *ensures*.

$$p \textit{ unless } q \text{ in } F \parallel G \ \equiv\ p \textit{ unless } q \text{ in } F \ \wedge\ p \textit{ unless } q \text{ in } G, \text{ and}$$

$$p \textit{ ensures } q \text{ in } F \parallel G \ \equiv\ (p \textit{ ensures } q \text{ in } F \ \wedge\ p \textit{ unless } q \text{ in } G) \vee (p \textit{ unless } q \text{ in } F \ \wedge\ p \textit{ ensures } q \text{ in } G)$$

2.2.2 Program Composition by Superposition

Superposition is another mechanism to structure programs. Suppose we are given a program F and a statement r that does not assign to any of the variables of F. Then, the statement r can be superposed on program F in two ways – either it can be combined with a statement s of F to yield an augmented statement $s \| r$, or it can be added by itself to F, thus resulting in the composite program $F \parallel r$. In either case, all the *unless*, *ensures*, and *leads-to* properties of the original program are preserved. Moreover, the fixed-point of the transformed program implies the fixed-point of the original program. This result is referred to as the Superposition theorem.

3 Program Refinement in Unity

Program refinements in Unity can be divided into two classes: those that preserve all *unless* (safety) and *leads-to* (progress) properties [5], and others that preserve the fixed-point (i.e., if the original program terminates, then so does the refined program and moreover, the fixed-point of the refined program implies the fixed-point of the original program). The first kind of refinement is useful for interactive programs whereas the second kind is useful for terminating programs. Formally, the two refinements are defined as follows.

Let F be a program, let G be a refinement of F, and let $FP\text{-}F$ and $FP\text{-}G$ be the fixed-points of the two programs. We say that G is a *property preserving* refinement of F iff for all predicates p, q the following two assertions hold.

- (p *unless* q holds in F) $\Rightarrow$ (p *unless* q holds in G), and
- ($p \mapsto q$ holds in F) $\Rightarrow$ ($p \mapsto q$ holds in G).

Note that the preservation of *ensures* properties is not required in this definition. This is because *leads-to* (and not *ensures*) is used as the basic relation for the specifying progress properties, and preservation of *leads-to* is all that is required in most situations. Similarly, we say that G is a *fixed-point preserving* refinement of F iff the following two assertions hold.

- ($true \mapsto FP\text{-}F$ in F) $\Rightarrow$ ($true \mapsto FP\text{-}G$ in G), and
- $FP\text{-}G \Rightarrow FP\text{-}F$ in G.

It is not too difficult to construct fixed-point preserving refinements that are not property preserving. The existence of property preserving refinements that are not fixed-point preserving is more interesting. The relationship between the two kinds of refinements is considered next.

Theorem 1: A property preserving refinement that does not introduce any new variables (i.e., does not modify the state space) is also fixed-point preserving.

Proof: Consider a program F and its property preserving refinement G. Let $FP\text{-}F$ and $FP\text{-}G$ be the fixed-points of F and G. Let p be a predicate that defines a unique value for each program variable, i.e., p represents a point in the state space of the programs. Consequently, $\neg p$ denotes the state space minus the state denoted by p. Now, observe the following.

$p \Rightarrow FP\text{-}F$
$\Rightarrow$ {property of fixed-points [5]}
p *unless false* in F
$\Rightarrow$ {preservation of *unless*}
p *unless false* in G
$\Rightarrow$ {property of fixed-points}
$p \Rightarrow FP\text{-}G$
$\Rightarrow$ {property of fixed-points}
$\neg(p \mapsto \neg p$ in $G)$

$\Rightarrow$ {preservation of *leads-to*}
$\neg(p \mapsto \neg p$ in $F)$
$\Rightarrow$ {property of *unless* [5]}
p *unless false* in F
$\Rightarrow$ {property of fixed-points}
$p \Rightarrow FP\text{-}F$

Thus, $FP\text{-}F \equiv FP\text{-}G$. Therefore, $FP\text{-}G \Rightarrow FP\text{-}F$, which is the second condition for fixed-point preserving refinements. The first condition for fixed-point preserving refinements follows from the fact that *leads-to* properties are preserved and $FP\text{-}F \equiv FP\text{-}G$. □

Example 2: For an example of a property preserving refinement that is not fixed-point preserving, consider any terminating program F and add to it a statement $t :: x := x+1$, where x is a fresh variable. It follows from the Superposition theorem that program $F \,[\!]\, t$ is a property preserving refinement of F. However, since $F \,[\!]\, t$ does not terminate, program $F \,[\!]\, t$ is not a fixed-point preserving refinement of F. □

The following theorem also relates the two kinds of refinements.

Theorem 2: If G is a property preserving refinement of F and if both F and G terminate, then G is also a fixed point preserving refinement of F.

Proof: Because program G terminates, our proof obligation is to show that $FP\text{-}G \Rightarrow FP\text{-}F$. Observe the following.

$true \mapsto FP\text{-}F$ in F	, assumption
$true \mapsto FP\text{-}F$ in G	, G is a property preserving refinement
$\neg FP\text{-}F \mapsto FP\text{-}F$ in G	, strengthening [5]
$FP\text{-}G \wedge \neg FP\text{-}F$ *unless false* in G	, property of fixed-points
$FP\text{-}G \wedge \neg FP\text{-}F \mapsto$ *false* in G	, PSP theorem [5] on above two
$\neg(FP\text{-}G \wedge \neg FP\text{-}F)$ in G	, impossibility theorem [5]
$FP\text{-}G \Rightarrow FP\text{-}F$ in G	, predicate calculus □

Theorem 3: The relations "*is a property preserving refinement of*" and "*is a fixed-point preserving refinement of*" are preorders (i.e., reflexive and transitive).

Proof: Follows from the reflexivity and transitivity of implication. □

4 Some Useful Program Refinements

In this section we discuss some program refinements that are useful in the formal derivation of programs. Though these refinements are stated in the UNITY logic, equivalent formulations in other formal frameworks such as temporal logic are possible. In Section 4.1 we consider the subject of data refinements. In Section 4.2 we consider the strengthening of guards. Finally, in Section 4.3 we consider the question of atomicity refinement. The proof of one of the theorems appears in the appendix; the rest of the proofs appear in [26].

4.1 Data Refinement

Data refinement is a very effective tool in program derivation as it provides a programmer the freedom to express his algorithm using a convenient abstract data type. Later, the chosen abstract data type is implemented by an available data type on the target machine while preserving the correctness of the original algorithm. In this section we develop conditions under which data refinement can be carried out for reactive programs.

Let $F \,[\!]\, s$ be a program using a variable x of abstract data type X. Assume that statement $s :: x := f(x)$ *if* $p(x)$, which performs operation f on the abstract object x provided guard p holds, is the only statement modifying x. We wish to examine conditions under which variable x can be implemented by a fresh variable y of the concrete data type Y. For this purpose, let $t :: y := g(y)$ *if* $q(y)$ be a statement that performs operation g on concrete object provided guard q holds. In order for program $F \,[\!]\, t$ to *simulate* program $F \,[\!]\, s$, there should exist a function h (called the abstraction function) from Y to X such that $h(y)$ simulates x at all times. Before we state the theorem we discuss some preliminaries.

Notation: For any expression e, define e' to be the expression obtained by a syntactic substitution of term x by the term $h(y)$ and define F' to be the program obtained by syntactically substituting x by $h(y)$ everywhere. Let FP_0, FP_1 denote the fixed-points of $F \,[\!]\, s$ and $F \,[\!]\, t$ respectively. Define conditions $A0, A1, A2$, and $A3$ as follows.

- h(initial value of y) = initial value of x. ……$(A0)$
- $p(h(y)) \;\Rightarrow\; h(g(y)) = f(h(y))$, for all y. ……$(A1)$
- $p(h(y)) \equiv q(y)$, for all y. ……$(A2)$
- $p(h(y)) \;\Rightarrow\; [h(g(y)) = h(y) \;\Rightarrow\; g(y) = y]$, for all y. ……$(A3)$

□

Theorem 4: Program $F' \,[\!]\, t$ is a property preserving refinement of $F \,[\!]\, s$ under the replacement of x by $h(y)$ (i.e., p *unless* q in $F \,[\!]\, s \;\Rightarrow\; p'$ *unless* q' in $F \,[\!]\, t$, and $p \mapsto q$ in $F \,[\!]\, s \;\Rightarrow\; p' \mapsto q'$ in $F \,[\!]\, t$) if conditions $A0, A1$, and $A2$ hold. Furthermore, $F' \,[\!]\, t$ is a fixed-point preserving refinement of $F \,[\!]\, s$ under the replacement of x by $h(y)$ (i.e., $FP_1 \Rightarrow FP_0'$, and $true \mapsto FP_0$ in $F \,[\!]\, s \;\Rightarrow\; true \mapsto FP_1$ in $F \,[\!]\, t$) provided all four conditions $A0 - A3$ hold. □

The above theorem is motivated by Hoare's correctness conditions for implementation of abstract data types for sequential programs [11]. In its current form, it was used to implement a set of elements by a hash table in [25]. Sanders develops similar conditions for refinements in [23].

Note: If the program to be refined includes multiple statements s_i that modify the abstract variable, then it can be refined by including a new statement t_i corresponding to each abstract statement s_i. The correctness condition for the refinement now includes $A0$, and a set of conditions $A1 - A3$ corresponding to each pair of statements (s_i, t_i). □

Given a property preserving refinement of an abstract object x by a concrete object y through an abstraction function h, it is possible to add a statement $t :: y := g(y)$ *if* $q(y)$,

where $h(g(y)) = h(y)$ for all y. Statement s refines a *skip* statement $s :: x := x$ *if* $q(y)$ of the abstract program. Condition $A1$ is satisfied because f is the identity function and $h(g(y)) = h(y)$. Condition $A2$ is satisfied vacuously. Such statements that preserve $h(y)$ by merely restructuring the concrete object are called *restructuring* statements. As an example, consider implementing a bank account x by a checking account y and a savings account z through the abstraction function $h(y, z) = y + z$, i.e., the sum of accounts y and z is meant to simulate abstract account x. In this case, if conditions $A0 - A2$ (which link program variables x, y, and z and are needed for the proof of correctness of the refinement) hold, then a restructuring statement $s :: y, z := y - 1, z + 1$ *if* $y \geq 1$ that transfers one dollar from y to z can be added without affecting the correctness of the refinement. Restructuring statements cannot be added freely in the case of fixed-point preserving refinements because these statements may never reach a fixed point. For example, if we add another restructuring statement $t :: y, z := y+1, z-1$ *if* $z \geq 1$, then the refinement may not terminate even if the original program terminates.

Example 3: Consider the following program in which the variable x is a natural number.

Program *simple*
  initially $x = 0$
  assign
    $x := x + 1$ *if* $x > 5$
  [] $z := x^2$
end

We wish to replace variable x by a fresh variable y of the type *queue*. The abstraction function h that we choose here maps a queue to the number of elements in the queue, i.e, $h(y) = size(y)$.

The incrementing of x is replaced by the appending (the symbol ';' denotes concatenation) of some arbitrary element e to the queue. Condition $A0$ is satisfied by setting y to null initially. Condition $A1$ is satisfied because $size(y; e) = size(y) + 1$. Condition $A2$ is satisfied vacuously. Condition $A3$ is satisfied because $size(y; e) \neq size(y)$. Thus, we obtain the following refined program that preserves all *unless* and *leads-to* properties as well as the fixed-point.

Program *simple*
  initially $y = null$
  assign
    $y := y; e$ *if* $size(y) > 5$
  [] $z := (size(y))^2$
end □

Example 4: In this example taken from [5], we consider replacing a shared variable by unbounded FIFO channels. Consider a shared variable x that is shared between two processes F and G. Process F accesses x only be statement s and process G accesses x only by statement t; these statements are defined as follows:

$s :: x := x \oplus d$ *if* p, and
$t :: vs, x := f(vs, x), g(x)$ *if* $b(x) \wedge q$.

Variable vs represents local variables of G. It is assumed that predicates p, q do not mention x. Let $type(x) = X$, $type(d) = X$ and $type(\oplus) = X \times X \rightarrow X$. It is apparent from examining the statements that process F only modifies x while process G tests, reads, and modifies x.

We wish to replace the shared variable x by two variables: one, a channel from process F to G called c and the other, a local copy of x at G called y. Thus, $type(y) = X$ and $type(c) = X^*$. We wish to transform the assignment statement s to a statement s' in which variable d is appended to the channel variable c. Similarly, we wish to transform the assignment statement t to a statement t' in which process G accesses variables y and c instead of variable x.

s' :: $c := c; d$ *if* p, and
t' :: $vs, y := f(vs, y \oplus c), g(y)$ *if* $b(y \oplus c) \wedge q$.

The question then arises: under what conditions does this transformation preserve all *unless* and *leads-to* properties? The answer lies in the conditions $A0, A1$, and $A2$ of Theorem 4 presented earlier. Here x represents the abstract object and the pair (y, c) represents the concrete object. We choose a representation function h as follows:

$h(y, c) = y \oplus c$,

where the function $\oplus$ is extended to handle a string input as follows:

$$y \oplus c = \begin{cases} y & \text{if } c = null \\ (y \oplus head(c)) \oplus tail(c) & \text{otherwise} \end{cases}$$

Condition $A0$ of Theorem 4 is satisfied by choosing initial values for y and c such that the initial value of x = initial value of $y \oplus c$. Condition $A2$ of the theorem is satisfied as $b(y \oplus c) = b(h(y, c))$. Henceforth, we concentrate on the satisfaction of condition $A1$. In order to satisfy it, we have to show that

- $h(y, c; d) = h(y, c) \oplus d$, and
- $b(y \oplus c) \Rightarrow h(g(y), c) = g(h(y, c))$.

The proof of satisfaction of the first condition is as follows:

$h(y, c; d)$
= {definition of h}
$y \oplus (c; d)$
= {definition of $\oplus$, induction}
$(y \oplus c) \oplus d$
= {definition of h}
$h(y, c) \oplus d$

In order to prove that the second condition is satisfied, we assume property $B0$ defined as follows:

$b(x \oplus c) \Rightarrow [g(x \oplus c) = g(x) \oplus c]$, for all x, c. ……$(B0)$

Based on this property, the proof of second condition is as follows.

second condition
= {definition}
$b(y \oplus c) \;\Rightarrow\; [h(g(y), c) = g(h(y, c))]$
= {definition of h}
$b(y \oplus c) \;\Rightarrow\; [g(y) \oplus c = g(y \oplus c)]$
= {assumption $B0$}
true

This proves that the transformation of statements s, t to statements s', t' is legal if condition $B0$ is satisfied. Note that property $B0$ follows from the simpler property $B0'$ defined below. (The proof is by induction on the length of sequence c.)

$g(x \oplus d) = g(x) \oplus d$ ……$(B0')$.

Next, we add a restructuring statement u to the transformed program:

u :: $y, c := y \oplus head(c), tail(c)$ *if* $c \neq null$.

This transformation fulfills the conditions of restructuring statements because

$c \neq null \;\Rightarrow\; h(y, c) = h(y \oplus head(c), tail(c))$.

At this point let us recapitulate what we have done so far. We set out with the task of replacing shared variable x by an asynchronous channel from F to G. Statement s' in process F represents the transmission of a data item to channel c and so fits in with the message passing paradigm. Statement u (which is a part of process G) represents the reception of a data item (along with an update of local variable y) and therefore, also fits in with the message passing paradigm. However statement t' (which is a part of process G) is not yet in the right form as it mentions the channel variable c. Thus, this statement will require further transformation. We will return to this example in the next subsection after we have stated and proved another theorem about program transformations. □

4.2 Strengthening of Guards

Strengthening the guard of a statement obviously preserves all the safety properties of a program as any state that is reachable in the refined program is also reachable in the original program. In this section we develop conditions under which this program transformation preserves all the desired program properties including progress properties.

Theorem 5: Let F be a program and let s :: A *if* p be a statement. Let statement t :: A *if* $p \wedge q$ be obtained by strengthening the guard of statement s. Then, program $F \,[\!]\, t$ is a property and a fixed-point preserving refinement of the program $F \,[\!]\, s$ if the following two conditions hold in F.

- $p \mapsto q$
- Let x be the set of variables of $F \,[\!]\, s$. Then, there exists a non-increasing function g of x that is bounded from below such that

  $g(x) = k \;\wedge\; q$ *unless* $\neg p \;\vee\; g(x) < k$, for all k. □

Proof: Appears in Appendix.

Corollary 1: Let statement s be A *if* p, statement t be A *if* $p \wedge q$, and F be any program. Then, program $F \,[\!]\, t$ is a property and fixed-point preserving refinement of $F \,[\!]\, s$ if the following two conditions hold in F:

- $p \mapsto q$, and
- q *unless* $\neg p$.

Proof: Define g to be a constant function. Thus, g is non-increasing and bounded from below. Consequently, both the conditions of Theorem 5 are satisfied. □

Corollary 2: Let statement s be A *if* p, statement t be A *if* q, and F be any program. Then, program $F \parallel t$ is a property and fixed-point preserving refinement of $F \parallel s$ if the following three conditions hold:

- $q \Rightarrow p$ in $F \parallel t$,
- $p \mapsto q$ in F, and
- q *unless* $\neg p$ in F.

Proof: Let statement u be A *if* $p \wedge q$. Then, it follows from Corollary 1 that $F \parallel u$ is a property and fixed-point preserving refinement of $F \parallel s$. Because $q \Rightarrow p$ is an invariant of $F \parallel t$ and u has a stronger guard than t, it follows that $q \Rightarrow p$ is also an invariant of $F \parallel u$. Therefore, $p \wedge q \equiv q$ in $F \parallel u$. Consequently, the guard of statement u can be changed from $p \wedge q$ to q by the substitution axiom, thus yielding program $F \parallel t$. The desired theorem follows. □

Theorem 5 was first reported in [28] and used in [25] to synchronize processes that accessed a common resources. Processes accessed the resource when a certain predicate was set to *true* by an underlying mutual exclusion algorithm. This predicate was added to the guard of each statement that accessed the resource. The correctness of the refinement followed from the starvation-freedom property of the mutual exclusion algorithm. Corollary 2 first appeared in [5].

Example 4: (Continued from previous subsection) In the last section we discussed the implementation of variable x shared between processes F and G by a channel c from F to G and a local variable y at G. We transformed the pair of statements s (in F) and t (in G) to three statements s' (in F), t' (in G), and u (in G). The transformation was proved to be correct provided condition $B0$ (or the stronger condition $B0'$) held. The transformed statements s' and u were in the right form whereas statement t' needed further refinements. Here, we use Theorem 5 to transform statement t' into statement v defined as follows:

$v :: vs, y := f(vs, y), g(y)$ *if* $b(y) \wedge q$

This statement is in the right form as it mentions only variables local to process G.

In order to carry out the transformation from t' to v, we assume the following two conditions:

$b(x) \Rightarrow b(x \oplus d)$, for all x, d, and ……$(B1)$

$b(x) \Rightarrow f(vs, x \oplus d) = f(vs, x)$, for all x, d, vs. ……$(B2)$

The proof of correctness is in two steps: first, statement t' is transformed to statement $t'' :: vs, y := f(vs, y \oplus c), g(y)$ *if* $b(y) \wedge q$ and later, statement t'' is transformed to statement v.

It follows from condition $B1$ and induction on the length of c that $b(y) \Rightarrow b(y \oplus c)$. Therefore, by Corollary 2, the guard of statement t' can be strengthened to $b(y) \wedge q$ provided the following two conditions hold in the remainder of the program (i.e., the program without statement t'):

1. $y \oplus c = m \mapsto y = m$, and
2. $b(y)$ *unless false*.

For a proof of Condition 1, observe that on account of statement u, $y \oplus head(c) = k$ *ensures* $y = k$. The required condition follows from the repeated application of this progress property. For a proof of Condition 2, observe that statement u is the only statement in the remainder of the program that modifies variable y. Furthermore, from condition $B1$, $b(y) \Rightarrow b(y \oplus head(c))$. Consequently, $b(y)$ *unless false* holds over the remainder of the program. This completes the proof of correctness of the transformation of statement t' to statement t''.

Next, we consider the refinement of statement t'' to statement v. It follows from repeated application of conditions $B1$ and $B2$ that

$$b(y) \Rightarrow f(vs, y \oplus c) = f(vs, y).$$

Consequently, by the substitution axiom, the expression $f(vs, y \oplus c)$ may be substituted by the expression $f(vs, y)$ in statement t''. As a result, we obtain statement v, our goal statement.

Summing up the sequence of refinements, the statements s and t can be replaced by the statements s', u, and v provided conditions $B0, B1$, and $B2$ hold. Similar conditions were defined by Chandy and Misra in [5] as the *Asynchrony Condition*. Our stepwise derivation of the condition provides a useful illustration of the program refinement theorems discussed here, in addition to providing some insight into their theorem. Recently, Sanders has also presented a development of the Asynchrony Condition in the framework of *mixed specifications* [23].

If, in addition, we want the above program refinement to be fixed-point preserving, then it can be shown, on the basis of condition $A3$ of Theorem 4, that the following two conditions suffice:

$p \Rightarrow [x \oplus d \neq x]$, for all x, d, and ……$(B3)$

$b(x \oplus c) \wedge q \Rightarrow [g(x) \oplus c = x \oplus c \Rightarrow g(x) = x]$, for all x, c. ……$(B4)$

□

The following example taken from [5] illustrates an application of the refinements developed here.

Example 5: Let statements s and t be defined as follows:

s :: $x := x - d$ *if* p, and
t :: $vs, x := vs + 1, x + e$ *if* $x < 0 \wedge q$, where $d \geq 0$ and $e \geq 0$.

Conditions $B0'$ (which implies $B0$), $B1$, and $B2$ for this example translate to the following three properties respectively.

$(x - d) + e = (x + e) - d$,
$x < 0 \Rightarrow x - d < 0$, and
$x < 0 \Rightarrow vs + 1 = vs + 1$.

Because all these conditions hold, statements s, t can be transformed to the following set of statements that use a channel c and a local variable y while preserving all *unless* and $\mapsto$ properties.

$$
\begin{array}{l}
\quad c := c; d \ \textit{if} \ p \\
[\!] \ y, c := y - \textit{head}(c), \textit{tail}(c) \ \textit{if} \ c \neq \textit{null} \\
[\!] \ vs, y := vs + 1, y + e \ \textit{if} \ y < 0 \ \wedge \ q
\end{array}
$$

It follows from conditions $B3$ and $B4$ defined earlier that this refinement also preserves the fixed-point provided $p \Rightarrow d > 0$ is an invariant of the program. □

4.3 Refining Atomicity

It is often easier to prove or derive a program with a coarse grain of atomicity than a program that uses a fine grain of atomicity. Consequently, one method for program development is to derive a program with a coarse grain of atomicity and later refine it into another program that uses a finer grain of atomicity [14]. In this section we develop conditions under which such atomicity refinements preserve all the desired properties. We focus our attention to the refinement of a complex guard in an assignment statement. We consider two syntactic forms – one in which the guard to be evaluated is a disjunction of predicates and another in which the guard to be evaluated is a conjunction of predicates. These are examined in sections 4.3.1 and 4.3.2 respectively.

4.3.1 Transforming Existential Quantification in Guards

Theorem 6: Let F be any program and $s \ :: \ A \ \textit{if} \ (\exists i :: p.i)$ and $t \ :: \ \langle \ [\!] \ i :: t.i \rangle$ where $t.i :: A \ \textit{if} \ p.i$, be any group of statements. Then, the program $F \ [\!] \ t$ is a property and a fixed-point preserving refinement of the program $F \ [\!] \ s$ if program F meets the following safety property:

$p.i \ \textit{unless} \ \neg(\exists i :: p.i)$, for each i. □

A simplified version of the above theorem was used in [25] to map a program onto a set of processes. There, an assignment statement represented a unit of work. The dummy in the existential quantification of the guard of a statement ranged over all the processes and the term of the existential quantification represented the mapping of a unit of work to a process. The code of a process was then obtained by an application of the theorem.

4.3.2 Transforming Universal Quantification in Guards

Theorem 7: Let F be any program and $s \ :: \ p := \textit{true} \ \textit{if} \ \neg p \ \wedge \ (\forall i :: q.i)$ be any statement. Consider a refinement of statement s in which the predicates $q.i$ are computed asynchronously with the help of fresh variables $y.i$ as follows: variables $y.i$ are initialized to false and statement s is replaced by the following set of statements G:

$$
\begin{array}{rlll}
p & := \ \textit{true} & \textit{if} & \neg p \ \wedge \ (\forall i :: y.i) \\
\quad [\!] \langle [\!] i :: y.i & := \ \textit{false} & \textit{if} & \neg p \ \wedge \ (\forall i :: y.i) \ \rangle \\
[\!] \ \langle \ [\!] \ i :: y.i & := \ \textit{true} & \textit{if} & \neg p \ \wedge \ q.i \rangle
\end{array}
$$

Then program $F \parallel G$ is a property and a fixed-point preserving refinement of program $F \parallel s$ if program F satisfies $\neg p \wedge q.i$ *unless false* , for each i. □

The above theorem originates from the two-phase handshake used in network protocols and can be understood as follows. The statement that sets program variable p to *true* represents a *controller* process and the statements that assign predicates $q.i$ represent *subordinate* processes. After setting $q.i$ to *true*, a subordinate process waits for an acknowledgement from the controller (in the form of variable p being set to *true*) before resetting it to *false*. The controller polls each subordinate (through local variable $y.i$) and sets p to *true* when it finds all the $y.is$ to be *true*. The correctness of the refinement is based on the fact that the subordinate processes do not reset predicates $q.i$ while they are being polled by the controller. It is possible to introduce more asynchrony into the above refinement by setting variables $y.is$ to *false* asynchronously. However, it has to be ensured that the next phase of polling $q.is$ does not begin until all the $y.is$ from the previous phase have been reset.

The refinement of a synchronous computation of guards by an asynchronous computation is similar to the idea of delay insensitivity discussed in the context of electronic circuits by Seitz in [24] and Martin in [20], and in the context of programs by Chandy and Misra in [5]. A program is said to be delay insensitive if for all the assignment statements, the right hand side of the assignment does not change as long as the left hand side of the assignment does not equal the right hand side. Thus, delay insensitivity allows an electronic circuit to be designed without using a common clock. For example, consider the statement $s :: p := p \vee (q \wedge r)$ that represents a combinatorial logic-gate. (Note that this statement is equivalent to the statement $p :=$ *true if* $\neg p \vee (q \wedge r)$.) For this statement to be delay insensitive, the program that s is a part of should satisfy the following property:

$\neg p \wedge q \wedge r$ *unless* p. ……(P)

In other words, a state in which p is *false* and q and r are *true* persists until p is set to *true*. On the other hand, the conditions for refinement of atomicity derived earlier ensure that the expression on the right hand side of a statement can be computed piecemeal. Delay insensitivity in general does not ensure this piecemeal evaluation; it ensures asynchrony among different statements as opposed to atomicity refinement which ensures asynchrony in the execution of a statement. For example, consider the statement s defined earlier and let us try using property P as the condition for atomicity refinement. In that case, it is possible that predicates q and r keep oscillating between *true* and *false* such that they are never *true* at the same time. In that case, condition P is vacuously satisfied. However, if q and r are evaluated asynchronously then it is possible that both will be found to be *true* and the subsequent setting of p to *true* will be incorrect. This shows that condition P is not acceptable as the correctness requirement. As evident from Theorem 7, the correct conditions for refinement of atomicity in this case are:

$\neg p \wedge q$ *unless false* and $\neg p \wedge r$ *unless false*.

5 Concluding Remarks

In this paper we discussed the idea of property preserving and fixed-point preserving refinements and presented some theorems under which some common program transformations are correct. We hope that these theorems will be a first step towards building

an adequate set of tools for program transformations. Once a sufficient number of theorems have been developed, it may be possible to implement recurring themes in parallel program derivation such as abstract data type implementation, process synchronization, and process scheduling, by program refinements alone.

Acknowledgements

The ideas and theorems discussed here have been motivated by discussions with Jay Misra.

References

[1] Abadi, M., and L. Lamport, The Existence of Refinement Mappings, Proceedings of the 3rd IEEE Symposium on Logic in Computer Science, 1988, pp. 165 – 175.

[2] Back, R. J. R., Correctness Preserving Program Refinements: Proof Theory and Applications, Mathematical Center Tracts, 131, Center for Mathematics and Computer Science (CWI), Amsterdam, 1980.

[3] Back, R. J. R., A Method for Refining Atomicity in Parallel Algorithms, Parallel Architectures and Languages Europe 1989, Eindhoven, June 1989, pp. 199 – 216.

[4] Back, R. J. R., and K. Sere, Stepwise Refinement of Parallel Algorithms, Science of Computer Programming, 13, 1989-90, pp. 133 – 180.

[5] Chandy, K. M., and J. Misra, Parallel Program Design: A Foundation, Reading, Massachusetts: Addison-Wesley, 1988.

[6] Dijkstra, E. W., A Discipline of Programming, Englewood Cliffs, New Jersey: Prentice-Hall, 1976.

[7] Emerson, E. A., and J. Y. Halpern, "Sometimes" and "Not Never" Revisited: On Branching Time versus Linear Time Temporal Logic, Journal of the ACM, 33(1), January 1986, pp. 151 – 178.

[8] Floyd, R. W., Assigning Meaning to Programs, Proceedings of American Mathematical Society Symposia in Applied Mathematics, 19, 1967, pp. 19-32.

[9] Gries, D., The Science of Programming, New York: Springer-Verlag, 1981.

[10] Hoare, C. A. R., An Axiomatic Basis for Computer Programming, Communications of the ACM, 12, 1969, pp. 576 – 580.

[11] Hoare, C. A. R., Proofs of Correctness of Data Representations, Acta Informatica, 1(4), 1972, pp. 271 – 281.

[12] Jones, C. B., Systematic Software Development Using VDM, Englewood Cliffs, N.J.: Prentice Hall, 1986.

[13] Lam, S. S., and A. U. Shankar, Protocol Verification via Projections, IEEE Transactions on Software Engineering, 10(4), July 1984, pp. 325 – 342.

[14] Lamport, L., and F. Schneider, Pretending Atomicity, Technical Report TR-89-1005, Department of computer Science, Cornell University, May 1989.

[15] Lamport, L., A Simple Approach to Specifying Concurrent Systems, Communications of the ACM, 32:1, Jan. 1989, pp. 32 – 47.

[16] Lipton, R. J., Reduction: A Method For Proving Properties of Parallel Programs, Communications of the ACM, 18(12), Dec. 1975, pp. 717 – 721.

[17] Lynch, N., and M. R. Tuttle, Hierarchical Correctness Proofs for Distributed Algorithms, Proc. Sixth Annual ACM Symposium on the Principles of Distributed Computing, 1987, pp. 137 – 151.

[18] Manna, Z., and A. Pnueli, How to Cook a Temporal Proof System for Your Pet language, Proceedings of the 9th ACM Symposium on Principles of Programming Languages, 1983, pp. 141 – 154.

[19] Manna, Z., and A. Pnueli, Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs, Science of Computer Programming, 4, 1984, pp. 257-289.

[20] Martin, A. J., Compiling Communicating Processes into Delay-Insensitive VLSI Circuits, Journal of Distributed Computing, 1(3), 1986.

[21] Milner, R., Calculi for Synchrony and Asynchrony, Theoretical Computer Science, 25,1983, pp. 267-310.

[22] Owicki, S., and D. Gries, An Axiomatic Proof Technique for Parallel Programs I, Acta Informatica, 6, 1976, pp. 319-340.

[23] Sanders, B., Stepwise Refinement of Mixed Specifications of Concurrent Programs, Proceedings of IFIP Conference on Programming Concepts and Methods, Israel, April 1990, eds. M. Broy and C. B. Jones, Elsevier Science Publishers B. V. 1990.

[24] Seitz, C., System Timing, in Introduction to VLSI Systems, eds. C. Mead and L. Conway, Reading, Massachusetts: Addison-Wesley, 1980.

[25] Singh, A. K., and R. Overbeek, Derivation of Efficient Parallel Programs: An Example from Genetic Sequence Analysis, International Journal of Parallel Programming, 18(6), Dec. 1989, pp. 447 – 484.

[26] Singh, A. K., Program Refinement in Fair Transition Systems, Technical Report 91-2, University of California at Santa Barbara, March 1991.

[27] Singh, A. K., Leads-to and Program Union, Notes on Unity: 06-89, The University of Texas at Austin, Texas, May 1989.

[28] Singh, A. K., On Strengthening the Guard, Notes on Unity: 07-89, The University of Texas at Austin, Texas, June 1989.

Appendix

Proof of Theorem 5: The proof is in four parts: first, we show that $F \parallel t$ preserves all the safety properties of $F \parallel s$; second, we show that $F \parallel t$ preserves all the progress properties of $F \parallel s$; next, we show that the fixed-point of $F \parallel t$ implies the fixed-point of $F \parallel s$; finally, we show that if $F \parallel t$ terminates then $F \parallel s$ also terminates.

Part 1: Let b *unless* c be a property of $F \parallel s$. Because the Hoare triple $\{b \wedge \neg c\}\ s\ \{b \vee c\}$ implies $\{b \wedge \neg c\}\ t\ \{b \vee c\}$, b *unless* c is also a property of $F \parallel t$. Thus, $F \parallel t$ preserves all *unless* properties of $F \parallel s$. (End of Part 1)

Part 2: Let $b \mapsto c$ be a property of $F \parallel s$. We prove that $b \mapsto c$ in $F \parallel t$ by induction on the proof of $b \mapsto c$ in $F \parallel s$.

Base case: b *ensures* c in $F \parallel s$.

From the definition of *ensures*, b *unless* c in $F \parallel s$ and there exists a statement u in $F \parallel s$ such that $\{b \wedge \neg c\}\ u\ \{c\}$. If $u \in F$ then the proof follows as the refinement preserves all the safety properties. Otherwise, $u = s$, i.e., $\{b \wedge \neg c\}\ s\ \{c\}$. Therefore,

$b \wedge \neg p \Rightarrow c$, and ..... $(C0)$
$\{b \wedge \neg c\}\ A\ \{c\}$. ..... $(C1)$

We prove the following two properties of program $F \parallel t$,

$b \wedge p \wedge g(x) = k \mapsto (b \wedge p \wedge q \wedge g(x) = k) \vee c \vee g(x) < k$, and ..... $(C2)$
$b \wedge p \wedge q \wedge g(x) = k \mapsto c \vee g(x) < k$, ... $(C3)$

and then observe the following in $F \parallel t$:

$b \wedge \neg p \Rightarrow c$	, property $C0$
$b \wedge \neg p \mapsto c$	, property of $\mapsto$ [5]
$g(x) = k$ *unless* $g(x) < k$	, g is non-increasing
$b \wedge \neg p \wedge g(x) = k \mapsto c \vee g(x) < k$	, PSP theorem
$b \wedge p \wedge g(x) = k \mapsto c \vee g(x) < k$	, transitivity on $C2, C3$ [5]
$b \wedge g(x) = k \mapsto c \vee g(x) < k$	, disjunction on above two [5]
$b \mapsto c$	, disjunction over k and g is bounded from below [5] □

Proof of $C2$:

$p \mapsto q$ in F	, assumption
$\neg(p \wedge q) \Rightarrow t.FP$	, t is A *if* $p \wedge q$
$p \mapsto q \vee (p \wedge q)$ in $F \parallel t$	, composition of $\mapsto$ [27]
$p \mapsto q$ in $F \parallel t$	, predicate calculus ..... $(C4)$
b *unless* c in $F \parallel s$	, assumption
b *unless* c in $F \parallel t$	, preservation of safety properties
$b \wedge p \mapsto (b \wedge q) \vee c$ in $F \parallel t$	, PSP theorem with $C4$
$b \wedge p \mapsto (b \wedge \neg c \wedge q) \vee c$ in $F \parallel t$	, predicate calculus
$b \wedge p \mapsto (b \wedge \neg c \wedge p \wedge q) \vee c$ in $F \parallel t$	, $b \wedge \neg c \Rightarrow p$ by $C0$
$b \wedge p \mapsto (b \wedge p \wedge q) \vee c$ in $F \parallel t$	, predicate calculus

$g(x) = k$ *unless* $g(x) < k$ in $F \,[\!]\, t$, g is non-increasing
$b \wedge p \wedge g(x) = k \mapsto (b \wedge p \wedge q \wedge g(x) = k) \vee c \vee g(x) < k$ in $F \,[\!]\, t$
, PSP theorem

Proof of $C3$:

$\{b \wedge \neg c\}\ A\ \{c\}$, property $C1$
$\{b \wedge p \wedge q \wedge \neg c\}\ t\ \{c\}$, t is A *if* $p \wedge q$
$b \wedge p \wedge q$ *ensures* c in t , definition of *ensures*
$g(x) = k$ *unless* $g(x) < k$ in t , g is non-increasing
$b \wedge p \wedge q \wedge g(x) = k$ *ensures* $c \vee g(x) < k$ in t , conjunction [5] ·····($C5$)
p *unless* $\neg p$ in F , property of *unless* [5]
$q \wedge g(x) = k$ *unless* $\neg p \vee g(x) < k$ in F , assumption
$p \wedge q \wedge g(x) = k$ *unless* $\neg p \vee g(x) < k$ in F , conjunction
b *unless* c in F , assumption
$b \wedge p \wedge q \wedge g(x) = k$ *unless* $(b \wedge \neg p) \vee c \vee g(x) < k$ in F
, conjunction
$b \wedge p \wedge q \wedge g(x) = k$ *unless* $c \vee g(x) < k$ in F , property $C0$
$b \wedge p \wedge q \wedge g(x) = k$ *ensures* $c \vee g(x) < k$ in $F \,[\!]\, t$, union theorem with $C5$

This concludes the base case.

Induction step:

- $b \mapsto r$ in $F \,[\!]\, s$, r *ensures* c in $F \,[\!]\, s$.

  $b \mapsto r$ in $F \,[\!]\, t$, induction hypothesis
  $r \mapsto c$ in $F \,[\!]\, t$, proof similar to base case
  $b \mapsto c$ in $F \,[\!]\, t$, transitivity

- $b \equiv (\exists\, i :: b.i)$ and $b.i \mapsto c$ in $F \,[\!]\, s$, for all i

  $b.i \mapsto c$ in $F \,[\!]\, t$, induction hypothesis
  $(\exists\, i :: b.i) \mapsto c$ in $F \,[\!]\, t$, disjunction over i
  $b \mapsto c$ in $F \,[\!]\, t$, $b \equiv (\exists\, i :: b.i)$

(End of Part 2)

Part 3: Let FP be the fixed-point of program F and R be the fixed-point of assignment A. Then, the fixed-point of programs $F \,[\!]\, s$ and $F \,[\!]\, t$ are $FP \wedge (p \Rightarrow R)$ and $FP \wedge (p \wedge q \Rightarrow R)$, respectively.

$p \wedge \neg q \Rightarrow (p \wedge q \Rightarrow R)$, predicate calculus
$p \wedge \neg q \Rightarrow$ fixed-point of t , observing t
$FP \equiv$ fixed-point of F , assumption
$FP \wedge p \wedge \neg q \Rightarrow$ fixed-point of $F \,[\!]\, t$, property of fixed-points
$FP \wedge p \wedge \neg q$ *unless false* in $F \,[\!]\, t$, property of fixed-points
$p \mapsto q$ in $F \,[\!]\, t$, property $C4$
$FP \wedge p \wedge \neg q \mapsto$ *false* in $F \,[\!]\, t$, PSP theorem
$\neg(FP \wedge p \wedge \neg q)$ in $F \,[\!]\, t$, impossibility theorem ·····($C6$)

Now observe the following in the context of program $F \parallel t$.

fixed-point of $F \parallel t$
$=$ {definition of fixed-point}
$FP \wedge (p \wedge q \Rightarrow R)$
$=$ {predicate calculus}
$FP \wedge (\neg p \vee \neg q \vee R)$
$=$ {predicate calculus}
$FP \wedge (\neg p \vee (p \wedge \neg q) \vee R)$
$=$ {distributing $\wedge$ over $\vee$}
$(FP \wedge (\neg p \vee R)) \ \vee \ (FP \wedge p \wedge \neg q)$
$=$ {simplifying using $C6$}
$FP \wedge (p \Rightarrow R)$
$=$ {definition of fixed-point}
fixed-point of $F \parallel s$ ……$(C7)$

(End of Part 3)

Part 4: Let FP and FP' indicate the fixed-points of programs $F \parallel s$ and $F \parallel t$, respectively. Observe the following.

$true \mapsto FP$ in F , assumption
$true \mapsto FP$ in G , preservation of $\mapsto$
$true \mapsto FP'$ in G , $FP \equiv FP'$ from $C7$

(End of Proof)

Communication Abstraction and Refinement

J.T. Yantchev

Department of Electronics and Computer Science,
University of Southampton, Southampton SO9 5NH, UK
and
Department of Electronic and Electrical Eng.,
University of Surrey, Guildford, UK.

Abstract

Concurrent systems are collections of data, processes, and communication channels. Top-down, hierarchical design of concurrent systems needs powerful abstraction facilities provided by the implementation language. While most languages provide some structuring mechanisms for data and process abstraction, none seems to provide any equivalent mechanisms for communication structuring. Communication channels are to communicate data and, therefore, all data structuring mechanisms provided by a programming language must be available to structure channels as well. In order to preserve behaviour through successive levels of design refinement, these means of communication structuring must preserve the abstraction of atomic transfers of values of arbitrary types.

key words: interprocess communication and synchronisation, multiparty interaction, abstraction and refinement.

Introduction

Most concurrent programming languages [5, 6, 8, 1] support the abstraction of concurrent systems as collections of data, processes, and communication channels. However, while they provide some structuring mechanisms for data and process abstraction, none seems to provide any equivalent mechanisms for communication structuring. Interprocess communication is almost universally viewed as a synchronised atomic exchange of values between two concurrently active processes. This affects the whole design process and intervenes with the freedom and ease in the refinement of the process structure. The design transformation steps may be non-trivial in some cases and, therefore, difficult to arrive

at and verify. In addition, the implementation may be less efficient, both in storage and speed, because of unnecessary data copying and context creation for process spawning.

The data structuring mechanisms supported by the contemporary programming languages provide a uniform view on data and data types. Paraphrasing Einstein's principle that 'Physics is simple only when viewed locally', we can say that they allow the component data within a large data structure to be subject to the same general principles of organisation. Structured data types may consist of components of arbitrary types, including themselves, and values of such types are treated as wholes and may be passed as parameters, returned as results of functions, and assigned to variables.

The same principle applies to processes [5, 8]. No distinction of kind need be made between systems with and without substructure and, indeed, a system which at one level of abstraction may be considered to consist of a process and the *environment* in which it evolves, may be considered as a single system at a higher level of abstraction. Thus the term process is used very broadly to mean any system whose behaviour consists of atomic communications; a process which for one purpose is taken to be atomic may, for other purposes, be decomposed into a collection of processes acting concurrently and independently.

It should be possible to abstract over communications, just as it is possible to abstract over processes and data. The communication of a data structure has a very natural internal structure, which matches the structure of the data, and can be conceived as the result of the simultaneous participation of more than two processes. For example, a number of concurrent processes may cooperate to provide the components of the data structure at the output end of a communication channel and, similarly, several concurrent processes may cooperate at its input end to consume the data structure. Overall synchronisation ensures that the data structure communicated on the channel is in a consistent state and that the atomic character of communication remains unchanged; atomicity, however, need only be temporal and not spatial.

In other words, communication channels are to communicate data and, therefore, all data structuring mechanisms provided by a programming language must be available to structure channels as well. Einstein's principle applied here means that no distinction should be made between communications with structure and communications without structure. A channel that communicates values of some structured type V may be viewed and, indeed, used both as a single channel of that type and as a structure of channels which communicate the components of V. Furthermore, no distinction should be made between the execution of a structured action by one process and by a system of processes. Just as a process may be either atomic or composed of concurrently acting subprocesses, an output (or input) action may be executed either by a single atomic process or by a collection of concurrent processes, where each process outputs (resp. inputs) one of the components of V.

This provides a means of *communication abstraction* and *refinement*, which is close to our intuitions and complements the means for data and process abstraction and refinement. It facilitates a top-down hierarchical design methodology and, furthermore, allows the design of more efficient programs. It is the purpose of this paper to introduce the

concept and consider some of its implications on design, implementation, and efficiency. The language CSP [5], which captures very well the notions of concurrency and process interaction, will be used as a notational tool. CSP has a well defined semantical model [2, 5] and has served as an underlying model for many concurrent programming languages, eg. [6]. The new formulation of communication will be modelled in basic CSP as a class of interactions which may involve an arbitrary number of processes.

1 Communicating Sequential Processes

C.A.R. Hoare's CSP [4], first published in 1978, is one of the most widely known programming languages for concurrency. In this language input and output, i.e. communication between two concurrent processes, was taken as primitive programmer-specified action with synchronisation as the mechanism for communication. The language, based on Dijkstra's guarded commands, allowed input requests to be used in guards, so that communication could determine process behaviour in a very elegant way. Non-determinism is inherent in the operation of concurrent systems, and a CSP process may exhibit non-deterministic behaviour, for example, when executing a guarded command in which more than one guard is satisfied.

In this early version of CSP communication channels are not named. Instead, the component processes of a parallel construction have unique names and communication channels are established by *direct process naming.* A channel is, therefore, between two processes only and it is logically impossible to violate this restriction. Within process A,

$$\textbf{proc } P = A||B \quad \text{where,} \quad \textbf{proc } A = \begin{array}{l} \vdots \\ B!v; \\ \vdots \end{array} \qquad \textbf{proc } B = \begin{array}{l} \vdots \\ A?x; \\ \vdots \end{array}$$

the command $B!v$ outputs the value v to the process named B. The value is input by a command $A?x$ occurring within the process B.

CSP was later developed [5] into a more basic mathematical language, refered to as TCSP (for *T*heoretical CSP), which abstracts from the details of interprocess communication by representing behaviour using the primitive notion of *event.* The occurrence of an event is regarded as an instantaneous or an atomic action without duration. Events may require the simultaneous participation of several processes in which case they are also called multiway synchronisations.

A process in TCSP is considered to be an agent which interacts with its environment in some universal *alphabet* Σ of events. The alphabet of a particular process P will be denoted as αP, with the obvious requirement that $\alpha P \subseteq \Sigma$. In the version of TCSP to be used here processes will be constructed using the following set of operators, where P and Q denote processes.

1. $STOP$, the *deadlocked process* which never engages in any action.

2. $SKIP$, the process which terminates immediately.

3. *prefix*, $a \rightarrow P$, $(a \in \Sigma)$. The unary operator $a \rightarrow$ has the effect of prefixing the event a to the behaviour of P.

4. *general choice*, $P[]Q$, behaves either as P or as Q, the control being exercised by their environment on the first action. If the first action is possible for both P and Q, the choice between them is non-deterministic.

5. *sequential composition*, $P;Q$, is a process which first behaves like P, but when P terminates successfully, $(P;Q)$ continues to behave like Q.

6. *concurrent composition*, $P||Q$, has the effect of concurrently combining the two processes which may evolve independently or interact through synchronising actions that are in both their alphabets $\alpha P \cap \alpha Q$. $(P||Q)$ terminates successfully when both components do so.

7. *hiding*, $P\backslash C$, $(C \subseteq \Sigma$ and C is finite), is a process which behaves like P, except that each event in C is concealed. Actions in C become internal to $P\backslash C$ and occur without being observed or controlled by the environment.

While it is certainly possible not to take the assignment of conventional sequential languages as primitive, but rather model it together with program variables as processes, we will choose not to do so. For we are just as interested in the internal state transformation of a process as in its external behaviour. The process

$$(x := e;\ P)$$

will then behave as P, except that the initial value of x will be defined to be the initial value of the expression e. Single assignments generalise easily to multiple assignments in an obvious way

$$x_1, \ldots, x_n := e_1, \ldots, e_n$$

In addition, conditional iteration will be denoted by

$$b * P$$

where b is a boolean expression, with $(\mathrm{true} * P)$ often abbreviated as $(*P)$.

1.1 Communication of values

Communication in TCSP follows closely the concept of communication of early CSP as a synchronised exchange of values by two concurrently active processes. However, rather than directly naming the participating processes, it uses the notion of explicit

communication channels on which the communications take place [5]. The communication of a value of type V is an event that is described by a pair

$c.v$, with $v \in V$

where c is the name of the channel and v is the value of the message.

A process that first outputs a value v on channel c and then behaves like P is defined as

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

the only event in which it is initially prepared to engage is the communication event $c.v$.

A process which is initially prepared to input any value $x \in V$, and then behave like $P(x)$, is defined as

$$(c?x \rightarrow P(x)) = []_{v \in V}(c.v \rightarrow P(v))$$

The abstraction of binary communication is maintained by the syntactically enforced convention that channels are used for communication in one direction only and between only two processes.

Channels will be declared and internalised using *channel declarations*. The scope of a declaration is between the enclosing pair of brackets and is defined as

$$(c : \textbf{chan of } T; P) = P \backslash \{c.v \mid v \in T\} \qquad \text{where } T \text{ is finite}$$

1.2 Process creation and program structure

The || and ; operators impose some structure on a concurrent program that allows easy identification of program segments that can be executed concurrently. For example, the execution of

$$P_1 \,||\, \cdots \,||\, P_n$$

causes the concurrent execution of $P_1, \ldots, P_n$. Its execution terminates only when execution of all $P_1, \ldots, P_n$ have terminated.

Since no sharing of program variables is allowed, this explicit single-entry, single-exit control structure allows the state transformation implemented by a program segment to be understood by itself. The understanding of a complete program is built up by understanding its constituent parts.

Earlier proposals, like **fork** and **join**, allowed the specification of arbitrary control structures, but failed to separate process definition from process synchronisation. Because **fork** and **join** can appear in conditionals and loops, the resulting program structure can be very difficult to understand and reason about. Program transformation is perhaps even more difficult to attempt.

On the other hand, the ability to specify arbitrary program control structures al-

lows the design of very efficient programs which exploit all the parallelism available in a computation, where using only || and ; would impose over-constrained sequentiality. This occurs, for example, when the external synchronisation graph is mapped onto the data flow graph of the internal state transformation. Although both graphs may have single-entry, single-exit structures, the resulting graph need not.

1.3 Data types

In addition to the conventional atomic types like **integer**, **char**, **real**, etc., we will assume the existence of the array, record, and union types. An *array* is a data type which is a rectangular collection of *items*, all of the same type.

m : **array**$[n, n]$ **of** T;

declares m to be an $n \times n$ matrix of items of type T. The items of m are indexed in the usual way, eg. $m[i, j]$.

A *record* is a data type which has a number of named fields.

r : $\{a$: **integer**, b : **real**$\}$;

r is a record of two fields a and b of type **integer** and **real**, respectively. The components of r are referenced using the the dot notation, as in $r.a$, for example. Values of record types can be constructed as in $\{26, 2.5\}$ or $\{a : 26, b : 2.5\}$.

A *union* is a data type which has a number of variant fields.

u : $\{e$: **integer** $+\!+$ f : **real**$\}$;

u is a union of two types. The components of a union type are denoted using the dot notation, as before. Values of union types may be constructed as in $\{e : 27\}$.

In addition, types may be given names. For example,

$t = \{a$: **integer**, b : **real**$\}$;

gives the name t to the record type $\{a$: **integer**, b : **real**$\}$; t may be then used to refer to that type. Variable and type declarations obey the same scope rules as channel declarations.

2 The problem with binary communication

One of the aims of the previous section was to emphasise that communication of values in CSP has been conceived from the outset as a binary interaction primitive. It remained essentially unchanged despite later developments that provided the notational tools to describe multiway interactions which may affect process behaviour in complex and intricate ways.

The expressive power and elegance of TCSP, however, are to a large extent due to the ability to consider an event a as being executed by a single process P at one level of process abstraction and by several concurrently active processes $P_1, \ldots, P_n$ at a different level. Although events in TCSP have no inherent structure and each P_i when taken in isolation will still be executing the same event a, this may, nevertheless, be considered as providing some form of event refinement. It is orthogonal to process structuring and facilitates the design process of refining one atomic process into concurrent subprocesses. For example, the equivalence

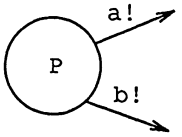
$$(b \to a \to (P||Q||R)) = (b \to a \to P)|||(a \to Q)|||(a \to R), \qquad b \notin \alpha Q \cup \alpha R$$

which is easily proved correct in TCSP, may be interpreted as describing a design transformation step that refines a sequential process into a set of concurrently active processes implementing the same behaviour as the original process.

The lack of a comparable refinement mechanism for value communications seems rather arbitrary and, more importantly, hinders design refinement and may result in over-constrained and inefficient programs. There are two aspects to the computation performed by a process. The first is the internal state computation, which may be specified by the possible sequences of state transformations. The second is the external behaviour in terms of the communications between the process and its environment. Design refinement aims to implement the internal state transformations by an optimal in some sense structure of processes by a sequence of program transformations, while preserving the external behaviour from one level of design to another. This, however, may not be a trivial task if the designer is not given the right tools.

Since there may be only one process connected to any of the ends of a communication channel, a communication of a process P with its environment cannot be used to synchronise concurrently active component processes of P. Therefore, although the sole purpose of external communications is to create a consistent data structure which is to be communicated to the environment or updated by the value received from the environment, there are no means to describe this in a natural way, when the components of the data structure are distributed over concurrently active processes. The program control structure that implements the internal state transformations will also have to implement the necessary synchronisations. As the following example demonstrates, this may over-constrain the execution order and/or result in storage and speed inefficiencies.

Example: design of a sequence generator. Assume that a process P must be implemented with the following specification S



$channels(P) = \{a!, b!\}$, where

a : chan of $\{f : T, g : T\}$; and b : chan of $\{g : T, h : T\}$;

$tr(P) = (a!, b!)*$

The external behaviour of P is specified by the set of its possible sequences of communications or traces $tr(P)$. It interacts with the environment by communicating on a and b, starting on a, and alternating between a and b in a strict sequence.

In addition, the values output must satisfy the following recurrence equations

$$a.f_1 = C_f \qquad a.g_1 = C_g \qquad b.h_1 = C_h$$
$$a.f_i = F(a.f_{i-1}) \qquad a.g_i = G_a(b.g_{i-1}) \qquad b.h_i = H(b.h_{i-1})$$
$$b.g_i = G_b(a.g_i)$$

where $a.f_i$, for example, denotes the f component of the i^{th} communication on channel a.

Perhaps, the simplest and most obvious solution which we may be tempted to try is

$$\begin{aligned} P \;=\; & f, g, h := C_f, C_g, C_h; \\ & *(\; a!\{f, g\}; \\ & \quad f, g := F(f), G_b(g); \\ & \quad b!\{g, h\}; \\ & \quad g, h := G_a(g), H(h) \\ & \;) \end{aligned}$$

There may be at most two concurrent processes active at any time and these are denoted by the concurrent assignments. However, consider the state transformation graph, derived from the recurrence equations and without the additional synchronisations imposed by the behavioural specification, given in figure 1.a. The nodes represent states and the arcs

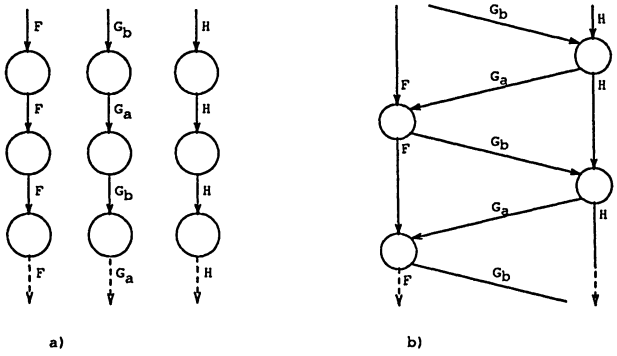


Figure 1: a) State transformation graph. b) Synchronisation graph

represent state transformations. There are three independent concurrent computations and, therefore, our first implementation of P has over-constrained the execution order.

We may hope that further refinement will result in an implementation that does not constrain the execution more than specified by S. Consider now figure 1.b, which shows the state transformation graph with the synchronisations imposed by the outputs. There may still be three concurrently active processes. The graph is, however, not single-entry, single-exit and cannot be constructed using || and ; alone. The additional synchronisations between the three concurrent processes must be implemented by using a pair of local channels, as shown in figure 2. Process P_g computes the g component of the internal state and

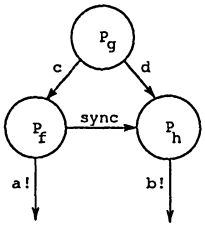


Figure 2: Sequence generator with hidden channels.

communicates it in alternating order to P_f and P_h on channels c and d, respectively. P_f and P_h output the whole pairs to the environment. Channel *sync* is used for synchronising P_f and P_h to ensure that the outputs on a and b occur in a strict sequence.

Although this implementation exploits all the concurrency available in the original specification, it also introduces notable inefficiencies both in storage and in speed. There are three more channels and the values of g are explicitly communicated from P_g to P_f and P_h. These values must also be stored in both P_f and P_h; storage for g is therefore duplicated twice.

In addition, the design transformation involved is not trivial, although it may appear as such in this simple example. ♣

3 Communication Refinement

The requirement that communication channels are used between two processes only is unnecessarily restrictive. The communication of a data value has a very natural internal structure, which matches the structure of the data and can be conceived as the result of the simultaneous participation of more than two processes. At the output end of a channel, for example, several concurrently active processes may synchronise and provide different components of the data structure that must be output. The *complete* output event may then be considered as being composed of many *partial* outputs. Similar arguments apply for the input end of the channel, but note that the two ends may be distributed over concurrent processes in different ways. The requirement for overall synchronisation ensures that the data structure communicated on the channel is in a consistent state and that the atomic character of communication remains unchanged.

3.1 Syntax and Semantics

Communication channels are to communicate data and, therefore, all data structuring mechanisms provided by a programming language must be available to structure channels as well. A channel that communicates values of some structured type V may be viewed and, indeed, used both as a single channel of that type and as a structure of channels which communicate the components of V. In addition, the same syntax that is used to denote the components of a data structure should be used to denote the components of a channel.

A communication event is, as before, described by a pair

$c.v$, with $v \in V$

where c is the name of the channel and v is the value of the message.

The representations of complete outputs and inputs remain unchanged and process that first outputs a value v on channel c and then behaves like P is defined as

$$(c!v \rightarrow P) = (c.v \rightarrow P)$$

and a process which is initially prepared to input any value $x \in V$, and then behave like $P(x)$, is defined as

$$(c?x \rightarrow P(x)) = []_{v \in V}(c.v \rightarrow P(v))$$

If the communicated value v is of a record type V with

$$V = \{a_1 : T_1, \ldots, a_n : T_n\}$$

then a process that is initially prepared to participate in communication on c by outputting the value $u \in T_i$ $(i \in \{1, .., n\})$ as the a_i component of v and then behave as P is defined as

$$(c.a_i!u \rightarrow P) = []_{x \in V'}(c.x \rightarrow P),$$

$$\text{and } V' = \{x_1, \ldots, x_{i-1}, u, x_{i+1}, \ldots, x_n \mid x_1 \in T_1, \ldots, x_n \in T_n\}$$

i.e. it is prepared to engage in all communication events $c.\{x_1, \ldots, x_{i-1}, u, x_{i+1}, \ldots, x_n\}$, which have u as the value of their a_i component.

A process which is initially prepared to participate in a communication on c, input any value $x \in T_i$ $(i \in \{1, .., n\})$ as the a_i component of v, and then behave like $P(x)$, is defined as

$$(c.a_i?x \rightarrow P(x)) = []_{y \in V}(c.y \rightarrow P(y.a_i)$$

i.e. it is initially prepared to engage in any communication event on the channel c, but its future behaviour is determined by the value of the a_i component alone.

Partial inputs on channels of array type, which are also product types, have similar

semantics, but the syntax differs. For example, given c : chan of array$[n,n]$ of T

$$(c[i,j]!v \rightarrow P)$$

is a process that outputs the i,j component of a matrix in the atomic communication of the whole matrix on channel c.

If the value u communicated is of a union type U with

$$U = \{b_1 : T_1 +\!+ \cdots +\!+ a_n : T_n\}$$

then a process that is initially prepared to output a value e from the b_i $(i \in \{1,..,n\})$ component of U and then behave as P is defined as

$$(c.b_i!e \rightarrow P) = (c.\{b_i : e\} \rightarrow P)$$

The only event in which it is initially prepared to engage is the communication event $c.\{b_i : e\}$.

As with TCSP, we will insist that a channel has the same alphabet at both ends and that an inputting process is prepared to accept any communicable value. This means that we cannot write for c defined as above

$$(c.b_i?x \rightarrow P(x))$$

as this may allow for obscure deadlock situations.

A convention is imposed that a component of a channel may be with one process only at each end and communication is in one direction only. In the rest of this paper the language with structured channels will be called CR-CSP, which derives from *C*ommunication *R*efinement CSP. TCSP restricted to binary named channels as the only interaction mechanism will be refered to as CSP.

3.2 Some equivalences

The translation of the input and output commands of CR-CSP into TCSP allows the easy proof of a set of equivalences between process, which we will intuitively expect to hold. Obviously, the equivalences given in [5] for binary value communications will hold in CR-CSP unchanged.

First consider two laws for record types. Assume $V = \{a : T_a, b : T_b\}$.

L1 $\quad (c.a!u \rightarrow P)|||(c.b!v \rightarrow Q) = (c!\{u,v\} \rightarrow (P||Q))$

L2 $\quad (c.a?x \rightarrow P(x))|||(c.b?y \rightarrow Q(y)) = (c?\{x,y\} \rightarrow (P(x)||Q(y)))$

i.e. a process which at one level of abstraction is taken to be atomic and connected to one end of the channel c may, at another level of abstraction, be decomposed into two concurrent processes each connected to a component of the channel c.

A channel c of a union type

$$U = \{a : T_a \ ++ b : T_b\}$$

can communicate values of any of the types T_a or T_b.

L3 $\quad (c.a!u \rightarrow P) = (c!\{a_i : e\} \rightarrow P)$

Note, that there is no equivalent law for the input end of c. It may appear that component selection for channels of union type does not add any expressive power, but this is not so. It provides the means to denote the distribution of the output end of a channel for union subtypes that are themselves of some structured type, as the following example illustrates.

$$c : \text{chan of } \{a : T_a \ ++ b : \{f : T_f \ , b : T_b\}\}$$
$$(c.b!\{f : u, g : v\} \rightarrow (P||Q)) = ((c.b.f!u \rightarrow P) \ || \ (c.b.g!u \rightarrow Q))$$

3.3 Examples

The sequence generator revised. The ability to use partial outputs to synchronise concurrently active process connected to the output end of a channel allows a straightforward implementation of the sequence generator. It consists of three concurrent processes implementing the state transformation graph in the obvious way and cooperating to output to the environment.

$P = (P_f||P_g||P_h)$ where,

$P_f =$ $f := C_f;$ *($a.f!f;$ $f := F(f)$)

$P_g =$ $g := C_g;$ *($a.g!g;$ $g := G_b(g);$ $b.g!g;$ $g := G_a(g)$)

$P_h =$ $h := C_h;$ *($b.h!h;$ $h := H(h)$)

It is clear that this program, while of a single entry, single exit control structure, implements the flow graph of figure 3. It imposes no additional restrictions on the execution order, specifies only the minimum synchronisation needed, and avoids all inefficiencies of which the other implementations suffered. Surprisingly, perhaps, it is even more efficient than the first implementation, which has to create contexts for the assignments after every external communication. ♣

Generic adder. Assume we have designed a simple process A to repeatedly input two real numbers x and y from channels a and b, respectively, add the numbers together,

and output the result on another channel c. In addition, we will assume that process definitions can be parameterised in the obvious way.

$A(a, b, c : \text{chan of real}) = (x, y : \text{real};\ *(a?x; b?y; c!(x + y)))$

The process A can then be used as a building block to implement, for example, a process M that must repeatedly add two $n \times n$ matrices, each input from the environment in one atomic communication, and output the result back, again in one atomic communication.

$a, b, c : \text{chan of array}[n, n] \text{ of real};$

$M = ||_{i,j=1}^{n} A(a[i, j], b[i, j], c[i, j])$

This illustrates how communication abstraction improves our ability to write modular programs. For example, a self-contained design module with an input port, or *interface*, of type T_1 can be used to build a bigger system with a composite interface c of type $C = \{a : T_1, b : T_2\}$, say. ♣

3.4 Implementation and Efficiency

It must be obvious that the class of CR-CSP programs subsumes the class of CSP programs and a programmer can, therefore, use binary communications wherever and whenever she or he deems necessary. In other words, in CR-CSP we can write programs that are at least as efficient as their CSP equivalents. In many cases, however, the ability to preserve local context upon multiway synchronisation allows us to do even better.

At a very general and abstract implementational level, a program in CR-CSP seems to always specify either the same or less synchronisation, data movement, and context creation than its equivalent CSP program. For example,

if $c : \text{chan of } \{a_1 : V_1, \ldots, a_n : V_n\}$ then

$P_a = (||_{i=1}^{n} P_i);\ c!\{v_1, \ldots, v_n\};\ (||_{i=1}^{n} P_i')$ and

$P_b = ||_{i=1}^{n} (P_i;\ c.a_i!v_i;\ P_i')$

obviously implement the same behaviour. Before the output on c all n processes P_i must synchronise. In P_a this is upon termination and in P_b, which can only be written in CR-CSP, this is upon communication. After the output, however, P_a will have to spawn n new processes P_i' with the associated cost of context creation, which in some implementations may be even higher than the cost of communicating on c.

In other cases where a CR-CSP program has no trivial translation into CSP, the contrast is even greater. For, collecting the components of a data structure together in order to perform an atomic communication certainly involves no less synchronisation and perhaps always more data movement and storage duplication.

4 Multiway Communication

Communication, both binary or structured, has so far been considered to be a *directed* activity, where one or more processes either *send* or *receive* a value or its components. However, the concept of communication as the occurrence of an event $c.e$ from a set of events $C = \{c.v \mid v \in V\}$ as a result of the cooperation of several concurrently active processes, with some of them partially specifying the value e and others using these choices to determine their future behaviour, is more powerful than that. A participating process can both make some or none of the choices and use some or none of the choices made by the other processes to determine its future behaviour.

This introduces a view of communication as a multiway *undirected* activity which may involve an arbitrary number of processes. The participating processes interact by each possibly *exposing* a value of some type and/or *reading* the values exposed by the other processes in an arbitrary way. The overall data structure created from the individual contributions determines the type of the interaction.

4.1 A Novel Multiway Communication Primitive

The interaction patterns that served as a model for the syntax introduced in Section 3.1 constitute a set richer than the one denoted by the input and output commands used so far. A more expressive syntax is needed that can be used to denote all interaction patterns from that set. It should, ideally, retain the simplicity of the input and output commands to denote directed communications and provide a proper extension to the original language.

Let us consider more carefully the three most primitive ways in which a process can participate in the communication of a value $v \in V$ on a channel c.

synchronisation a process may simply synchronise on the occcurrence of the communication event, without making or allowing any of the choices made to determine its future behaviour; it, therefore, need not even know the type of the communicated value.

output a process may output or *expose* either a component of the value v or the whole of v.

input a process may input or *read* arbitrary components of v or, indeed, the whole of v. This reflects the fact that an input does not influence which event from the set actually occurs, but the process that performs the input may use the event to determine its future behaviour in any way it likes.

Multiway communication means that a process may engage in an arbitrary combination of inputs and outputs and a multiway communication primitive must be sufficiently expressive to denote all such combinations.

The following convention on the use of composite names will be observed

Given c : chan of V; The prefix $(c \rightarrow P)$ will be defined as

$$(c \rightarrow P) = []_{v \in V}(c.v \rightarrow P)$$

The name c will be interpreted as a partially specified event name which may denote any event from the set $\{c.v \mid v \in V\}$. The equivalences

L4 $(c \rightarrow P) \,||\, (c!e \rightarrow Q) = (c!e \rightarrow (P \,||\, Q))$

L5 $(c \rightarrow P) \,||\, (c?x \rightarrow Q(x)) = (c?x \rightarrow (P \,||\, Q(x)))$

will then be valid for all $e \in V$.

The expression

$$(c \rightarrow P)$$

denotes a process which is initially prepared to engage in any communication on c without actually observing the value which is communicated. This will be used for pure synchronisations.

As before

$$(c!v \rightarrow P)$$

will denote the process which is initially prepared to engage in the communication of the value v on c. Similarly,

$$(c?x \rightarrow Q(x))$$

will denote the process which is initially prepared to engage in any communication on c and will store in the variable x the value actually communicated.

If the communicated value is of some structured type, eg. $V = \{f : V_f, g : V_g, h : V_h\}$, then more complicated input/output actions are possible. For example, the process

$$(c\{f!e, g?x\} \rightarrow Q(x))$$

is initially prepared to engage in the communication on c by outputting the f component of the value v and will store in the variable x its g component. Similarly for

$$(c\{f!e, g?x, h?y\} \rightarrow Q(x, y))$$

The description of a communication action is then a partial description of the structure of the value that is to be communicated, with the components of the value being either fixed by '!' or observed and copied by '?', in a manner reminiscent of the classical parammeter passing mechanisms.

4.2 Examples

Merger. The new communication primitive does, indeed, retain the denotational simplicity of the conventional input and output commands. The (possibly unfair) merger process M will be defined by an expression which can be written in CR-CSP and, indeed, CSP.

$$M(a, b, c : \textbf{chan of } T) = (x : T;\ *((a?x;\ c!x)\ []\ (b?x;\ c!x)))$$

where T is some type. ♣

Generic synchroniser. A process which simply synchronises on a communication need not know the type of the communicated value, which may, indeed, be different in different contexts. The process

$$S[t](a, b : \textbf{chan of } t) = *(a;\ b)$$

may then be used to constrain the communications on any two channels to occur in a strict sequence, as in

$$a, b, c : \textbf{chan of real};$$
$$M(a, b, c) \ ||\ S[\textbf{real}](a, b)$$

where $S[t](\ldots)$ denotes a generic process definition, which may be instantiated with different types substituted for t, as in $S[\textbf{real}](a, b)$. A distinct denotation is used to ephasise that types are not treated as first class values.

In this case it may seem simpler to design from scratch a new merger process M' with the required behaviour. In many other cases, however, this may be a non-trivial task, especially when the internal structure of M is not known and/or it has a more complex external behaviour. Practice has convincingly shown that component reusability is a powerful design tool. ♣

Swap. Two processes swapping values can be defined as

$$c : \textbf{chan of } \{f : T_f, g : T_g\};$$
$$(c\{f!x, g?x\} \rightarrow P(x)) \ ||\ (c\{f?y, g!y\} \rightarrow Q(y))$$

Similarly for a ring like swap which involves three processes.

$$c : \textbf{chan of } \{f : T_f, g : T_g, h : T_h\};$$
$$(c\{f!x, g?x\} \rightarrow P(x)) \ ||\ (c\{g!y, h?y\} \rightarrow Q(y)) \ ||\ (c\{f?z, h!z\} \rightarrow R(z))$$

♣

Grid relaxation. Multiway communication admits a very natural solution to the classical 2-D relaxation problem. This examples demonstrates how this novel communication

primitive bridges the gap between data concurrency and process concurrency.

$$
\begin{array}{l}
k : \mathbf{integer} = 1; \\
m : \mathbf{array}[n,n]\mathbf{of\ real}; \\
c : \mathbf{chan\ of\ array}[n,n]\mathbf{of\ real}; \\
P = ||_{i,j=1}^{n}(k \leq N) * (c![i,j]!m[i,j], \\
\qquad\qquad\qquad\qquad [i,j-1]?l, \\
\qquad\qquad\qquad\qquad [i-1,j]?u, \\
\qquad\qquad\qquad\qquad [i,j+1]?r, \\
\qquad\qquad\qquad\qquad [i+1,j+1]?d\,]; \\
\qquad\quad m[i,j] := (l+u+r+d)/4; \\
\qquad\quad k := k+1; \\
\qquad)
\end{array}
$$

In a single atomic action each $P_{i,j}$ exposes its current grid value and reads the values of its neighbours. The index expressions can be suitably modified to take into account boundary effects and wrap-around. The equivalent program with binary communications is not as intuitive and the possibility of deadlock must be considered explicitly.

5 Conclusions

Binary value communication is too restrictive a communication primitive. It is an abstraction over synchronised point-to-point message passing, and as such is more appropriate for an assembly language to program distributed systems than for a high-level concurrent programming language. It makes the description of intricate synchronisation patterns and complex transfers of data difficult to program.

This paper advocates the view that communication has a very natural internal structure, which matches the structure of the communicated data and can be conceived as the result of the simultaneous participation of an arbitrary number of processes. A number of concurrent processes may cooperate to provide and/or consume the components of the data structure in an arbitrary way.

TCSP is sufficiently powerful to model this novel concept in terms of its more primitive and well studied concepts. The result is a communication primitive that subsumes directed, both binary and structured, and undirected communications in a very elegant way. It provides a means of communication abstraction and refinement, which is close to our intuitions and complements the means for data and process abstraction and refinement.

There have been some recent research efforts to define a multiway communication primitive. Most of them define an abstraction which encapsulates representation details and processing activity on the combined state during an interaction. This, however, violates the intuitive notion of a *communication primitive* and is more appropriate for process abstraction. These and other issues are discussed in a recent paper [3] which formulates a set of criteria to view and define an interaction as a generalised communication *primitive*, albeit in a somewhat informal way. It is an undirected activity, which synchronises all

participants to create a global state, where each participant may provide and use state variables.

Although similar in some respects to the communication primitive introduced here, it, however, fails on several points. Firstly, communication is via arbitrary reads of the local variables of the participants, which does not admit any information hiding, elegant parameterised process definitions, nor communication abstraction. This seems to be counter-intuitive as a process should be allowed upon communication to expose as much of its internal state and hide the rest as necessary. Secondly, again because of the explicit interprocess access to variables, it fails to model directed communications in a natural and elegant way. The familiar CSP programs will look very cumbersome indeed, if expressed in this notation. In addition, considering communication to be orthogonal to interaction does not admit a natural model in terms of more primitive and abstract concepts.

Acknowledgments

I would like to acknowledge the support of my Supervisor of Studies Prof. C.R. Jesshope, now at the University of Surrey, and fellow research student P.R. Miller during the course of this work. It has also greatly benefited from discussions with Bill Roscoe and Mark Josephs at PRG, Oxford University, and Andy Gravell and Denis Nicole at Southampton University.

On the financial side, the support of the Bulgarian Ministry of Education and the IEE must be gratefully acknowledged.

References

[1] ANSI/MIL-STD 1815A, **Ada^{TM} Reference Manual.**

[2] Brookes, S.D., Hoare, C.A.R., Roscoe, A.W., *A theory of Communicating Sequential Processes*, Journal of ACM, Vol. 31, pp560-599,1984.

[3] Evangelist, M, Francez, N, Katz, S, *Multiparty Interactions for Interprocess Communication and Synchronisation*, IEEE Tran. on Software Engineering, Vol.15, No.11, pp1417-1426, 1989.

[4] Hoare, C.A.R., *Communicating Sequential Processes*, Communications of ACM, Vol 21, pp666-677,1978.

[5] Hoare, C.A.R., **Communicating Sequential Processes**, Prentice Hall, 1985.

[6] Inmos Ltd. **occam^{TM} Programming Manual**, Prentice-Hall, 1984.

[7] Inmos Ltd. **The Transputer Reference Manual**, Prentice-Hall, 1988.

[8] Milner R., **Communication and Concurrency**, Prentice-Hall, 1989.

On the semantics of languages for massively parallel SIMD architectures

(Extended Abstract)

Luc Bougé
LIP, ENS Lyon,
46 Allée d'Italie,
F-69364 Lyon Cedex 07, France
Email: bouge@lip.ens-lyon.fr

Abstract

We define a small language which embodies the main concepts of parallel languages for massively parallel SIMD architectures such as the Connection Machine, and give its Structured Operational Semantics. It yields two related program equivalences, corresponding respectively to two views of SIMD architectures. They are studied in detail. We then extend the language with parallel counterparts of non-local control transfer structures such as break and continue in C, and show that many classical control structures in real SIMD languages can be expressed thanks to them.

Key words Semantics — Parallel languages — SIMD — Massively parallel architecture — Connection Machine — Structured Operational Semantics — Program equivalence — The C language

Parallel ARchitectures and Languages have been the subject of an impressive amount of work in the last recent years (not only in Europe!) This work has led to definite progress, for both parallel MIMD architectures and SIMD ones. In this latter case, we can cite MPP (Massively Parallel Processor, [1]), DAP (Distributed Array Processor, [9]), CLIP4 (Cellular Logic Image Processor, [5]), GAPP (Geometric Arithmetic Parallel Processors, [10]), and the popular Connection Machine ([7], [6]).

Yet, this does not carry over when it comes to languages. Much effort has been devoted to designing and implementing languages for MIMD architectures. Consider for instance the achievements of Ada and Occam. This has resulted in considerable advances in semantics and programming models, based in particular on Petri nets and Milner's CCS process algebra. In contrast, very little has been said about SIMD languages and their programming models. In fact, this aspect is hardly even mentioned in surveys on processor arrays. Many processor arrays are still programmed in Fortran-derived ad-hoc languages, or even assembly code. Of course, some languages have been proposed. We can cite Actus for Illiac IV ([14]), Parallel Pascal for MPP ([15]), DAP Fortran initially designed for the DAP, which inspired the Fortran 8X standard ([8]), *Lisp, an extension of Common Lisp for the Connection Machine ([17]), ParIS, the macro-assembler for the Connection Machine ([13]) and POMPC, an extension of C currently under design and implementation for Connection Machine-like architectures ([12]). But very little work has been done on their formal semantics and their abstract programming model.

This paper is a step towards redressing this imbalance. It is organized as follows. We first describe the SIMD model we have in mind, basically the Connection Machine in its NEWS mode. Then, we define a small programming language called L which encapsulates the basic concepts of this model, and give its semantics. It gives birth to two program equivalences which will be studied in detail. Finally, we show that advanced control structures such as non-local escapes can be defined in these semantics, and can serve as a basis for expressing the parallel extension of the control structures of the C language. An extended version of this paper can be found in [2].

1 The L language

1.1 The programming model

We have in mind an SIMD model similar to the Connection Machine (CM), as described in [4]. It is composed of a number of parallel processors (64000 in the largest configuration of the CM) or *processing elements* (PEs), each of them managing its private store. All PEs are controlled by a unique external *sequencer*. At each point in the computation, the sequencer broadcasts to all PEs the common instruction to be executed. Each PE may then update its private store accordingly, at the constant address included in the instruction. Yet, each PE is equipped with an additional flag, called the *context*. The local store of a PE is updated only if the local context is in its *active* state. This possibility is crucial in implementing conditional branching in a SIMD discipline where each PE "does the same thing at the same time". Also, a *feedback* bus passes to each PE and conveys to the sequencer the or-ing of all local elements of a specified boolean array value. This is crucial in implementing conditional iterations, as the sequencer has to get informed that all PEs have satisfied their exit condition in order to stop repeatedly broadcasting the loop body and continue with the rest of the program.

The Connection Machine can operate in two communication modes. In the general

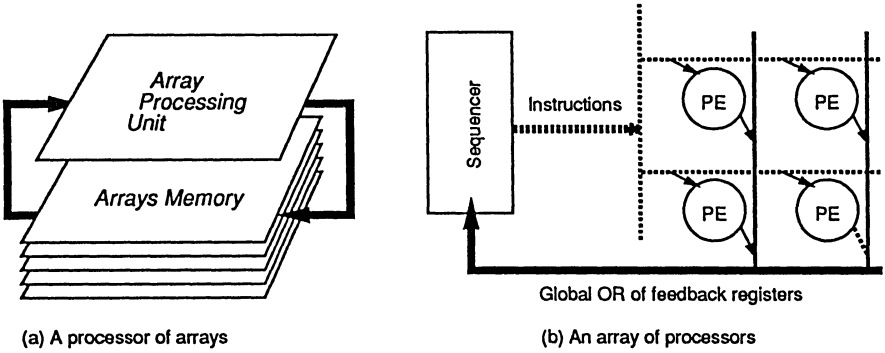


Figure 1: Two views of a massively parallel SIMD architecture

mode, each PE can send a message to any other PE through the hypercube network. Specialized *routers* attached to each group of PEs are in charge of conveying messages to their destination. In the NEWS mode (for North, East, West, South), the PEs are (possibly virtually) interconnected by a grid network. Communications are carried out by shifting values on the grid in a given direction. In this paper, we only consider this latter mode, as it is much more basic and moreover common to most processor arrays. For simplicity, we will consider an infinite square NEWS grid so as to avoid problems with borders, but everything would extend to regular topologies of higher dimension without borders. Observe finally that, in a SIMD discipline, there is no difference between sending and receiving. If everyone sends, then everyone is implicitly ready to receive too.

As pointed out by Steele and Hillis ([16]), there are two different views of such an SIMD architecture.
In the *macroscopic* view (see figure 1(a)), it appears as a sophisticated sequential processor with the capability of operating on arrays instead of scalars. Parallelism is thus on the data level. This is the user's viewpoint: *a processor of arrays.*
In the *microscopic* view (see figure 1(b)), it appears as an array of elementary sequential processors operating in parallel on their private scalar data. An external sequencer is in charge of synchronizing them. Parallelism is thus on the control level. This is the implementer's viewpoint: *an array of processors.*

The next sections show how this duality can be expressed at an abstract level. Steele and Hillis actually proposed in [16] a quite elegant functional framework, called CM-Lisp, where this duality is caught by an algebraic relation between operators: $\bullet\alpha = \alpha\bullet =$ Identity. In contrast, we propose in this paper an operational approach. As will be seen, this approach is much closer to "real" languages than the functional one, and it gives many interesting insights into them.

1.2 The language

We now describe the L language. We adopt the following convention. Identifiers with an initial uppercase letter denote arrays of objects, whereas those with an initial lowercase letter denote scalar objects. Array locations will be denoted u, v etc. If X is an array of objects, then $X|_u$ denotes its local element located at u.

No action The instruction `skip` simply does nothing.

Local store management The instruction $X := E$ stores into variable X the value of expression E. We stress that E should evaluate *elementwise*: its value at location u depends on the value of its variables at location u only. For instance, $E = f(Y, Z)$, where f denotes a scalar function, is of this type; think of matrix addition $X := Y + Z$ as opposed to matrix multiplication.

Communication As specified above, we consider only a global shifting of some array along some direction. The instruction `shift` X `along` d shifts the array X along the constant common direction d. Each PE stores into its local variable $X|_u$ the contents of the local variable $X|_{\bar{d}(u)}$ of its neighbor along direction $\bar{d}$, where $\bar{d}$ is the direction opposite to d. We stress that d is a syntactic constant `North`, `East`, `West` or `South`.

Sequencing The construct $P; Q$ executes P then Q.

Iteration The construct `while` B `do` P `end` iterates program P up to a point where the (elementwise) boolean expression B evaluates to false at *each* PE. This implicitly includes a communication, namely a global reduction of all elements of B's value to a single value.

Conditioning The construct `where` B `do` P `end` inhibits during the execution of program P those PEs such that the elementwise boolean expression B evaluates locally to false.

1.3 Two examples

Consider the following problem. Let A be an image, identified with an array variable: $A|_u$ is true iff u belongs to image A. Assume that all connected components of A are finite, even though A is infinite in general. Consider a point P, identified with an array variable: $P|_u$ is true iff P is located at u. The problem is to determine the connected component C of P in A. We proceed as follows. We build C by dilating a wave originating at P within the image A. Let C' be the last value if C. The result is found when the wave has not

extended in the current step: $C = C'$.

```
C := A ∧ P;
C' := ff;
while (C' ≠ C) do
    C' := C;
    dilate C;
    C := A ∧ C
end
```

Dilation is obtained by shifting a copy of the image in each direction and accumulating it.

```
dilate C   ::  dilate C to North;
               dilate C to East;
               dilate C to West;
               dilate C to South

dilate C to d  ::  Aux := C;
                   shift Aux along d;
                   C := C ∨ Aux
```

The connected component is found in a number of steps proportional to its diameter.

In order to illustrate conditioning, consider the following problem. Say each PE holds a number $1 \leq n \leq \alpha$, where α is some fixed bound. We want to compute each factorial $n!$ and assign it to variable f. The idea is to let all PEs iterate the multiplications, inhibiting them once they have reached $n = 1$.

```
F := 1;
while (N > 1) do
    where (N > 1) do
        F := F * N;
        N := N - 1
    end
end
```

All factorials are computed in a number of steps proportional to the bound α, whatever their number.

2 Semantics and program equivalences

2.1 The microscopic semantics

We describe a Structured Operational Semantics for L, according to the microscopic view which reflects the implementer's viewpoint. It considers a program in L as the (implicit) parallel composition of the (identical) local program of each PE of the array. This is

thus basically the semantics of an ordinary sequential centralized language, except that values are arrays of values, one for each PE, instead of scalars. Environments σ map (array) variables X to arrays of values $\sigma(X)$. As usual $\sigma[V/X]$ denotes the environment equal to σ everywhere except at X where it has value V. $\sigma|_u$ denotes the projection of environment σ on the PE located at location u: $\sigma|_u(x) = \sigma(X)|_u$, with $x = X|_u$. Observe that an environment is determined by its projections. Environments extend as usual to expressions: $\sigma(E)$ denotes the value of expression E in environment σ. The condition of being elementwise can be expressed by

$$\forall \sigma, \sigma', u \qquad \sigma|_u = \sigma'|_u \Rightarrow \sigma(E)|_u = \sigma'(E)|_u.$$

The main feature is that each PE can access and explicitly manipulate its context. In fact, a *stack* of contexts is needed to cope with nested conditioning. Its top value is the *current* context.

The Operational Semantics associates to each program a *transition system.* It describes the sequence of internal states of some (abstract) machine running the program. These states are triples $\langle P, \sigma, s\rangle$, where P denotes the program remaining to be executed, σ the current environment and s the context stack. The empty stack is denoted ε. By convention, we have $top(\varepsilon) = tt$, and $pop(\varepsilon) = \varepsilon$. The PE located at u is *active* if $top(s)|_u = tt$. Symbol $\bullet$ denotes the terminated program. The transition system is defined by a syntax-directed induction on the program.

skip

$$\langle \texttt{skip}, \sigma, s\rangle \longrightarrow \langle \bullet, \sigma, s\rangle$$

Assignment We express the idea that an inhibited PE remains passive and does not execute the assignment.

$$\langle X := E, \sigma, s\rangle \longrightarrow \langle \bullet, \sigma'', s\rangle$$

with $V = \sigma(E)$, $\sigma' = \sigma[V/X]$, $\sigma''|_u = \sigma'|_u$ if $top(s)|_u$, and $\sigma''|_u = \sigma|_u$ elsewhere.

Communication Similarly, an inhibited PE does not take part in any communication, though it may let other PE access its memory. Let $\sigma'(X)|_u = \sigma(X)|_{\bar{d}(u)}$ and $\sigma'(T)|_u = \sigma(T)|_u$ for $T \neq X$.

$$\langle \texttt{shift } X \texttt{ along } d, \sigma, s\rangle \longrightarrow \langle \bullet, \sigma'', s\rangle$$

with $\sigma''|_u = \sigma'|_u$ if $top(s)|_u$, and $\sigma''|_u = \sigma|_u$ otherwise.

Sequencing

$$\frac{\langle P, \sigma, s\rangle \longrightarrow \langle P', \sigma', s'\rangle \mid \langle \bullet, \sigma', s'\rangle}{\langle P; Q, \sigma, s\rangle \longrightarrow \langle P'; Q, \sigma', s'\rangle \mid \langle Q, \sigma', s'\rangle}$$

Iteration

$$\frac{\bigvee_u \sigma(B)|_u = tt \mid ff}{\langle \texttt{while } B \texttt{ do } P \texttt{ end}, \sigma, s\rangle \longrightarrow \langle P; \texttt{while } B \texttt{ do } P \texttt{ end}, \sigma, s\rangle \mid \langle \bullet, \sigma, s\rangle}$$

For conditioning, we adopt a mode similar to call-by-value. On entering a conditioned block, each PE evaluates the (elementwise) condition, and pushes the result onto the stack. It is then popped on exiting the block. We introduce a new syntactic construct `begin` P `end` to keep track of the blocks. For simplicity, we adopt the following notation. Let U be a boolean array value and s a stack. The stack obtained from s by the operation $push(U \wedge top(s), s)$ is denoted $U.s$.

Conditioning

$$\langle \texttt{where } B \texttt{ do } P \texttt{ end}, \sigma, s\rangle \longrightarrow \langle \texttt{begin } P \texttt{ end}, \sigma, \sigma(B).s\rangle$$

$$\frac{\langle P, \sigma, s\rangle \longrightarrow \langle P', \sigma', s'\rangle \mid \langle \bullet, \sigma', s'\rangle}{\langle \texttt{begin } P \texttt{ end}, \sigma, s\rangle \longrightarrow \langle \texttt{begin } P' \texttt{ end}, \sigma', s'\rangle \mid \langle \bullet, \sigma', pop(s')\rangle}$$

2.2 Derived program equivalences

The semantics above gives birth naturally to program equivalences. We consider here *functional equivalence*, as introduced in [11]: two programs are equivalent if they generate the same output result from the same initial state. We follow here a *total correctness* approach: divergence *can* be observed, it yields a special form of output denoted $\perp$. Observe that the semantics is deterministic: if a program generates a result from a given data, then this result is unique. At this point, some notation will be useful. We write $S \xrightarrow{*} S'$ between two states of the Structured Operational Semantics if there exists a *finite* (possibly null) derivation $S = S_0 \longrightarrow S_1 \longrightarrow \ldots \longrightarrow S_n = S'$, $n \geq 0$. We write $S \xrightarrow{\infty}$ if there exist an *infinite* derivation $S = S_0 \longrightarrow S_1 \longrightarrow \ldots$ from S. Throughout the paper, subscript M stands for "macroscopic" and m for "microscopic".

Microscopic equivalence We say that P terminates with result σ' from data σ and context stack s if $\langle P, \sigma, s\rangle \xrightarrow{*} \langle \bullet, \sigma', s'\rangle$, and that P diverges from data σ and context stack s if $\langle P, \sigma, s\rangle \xrightarrow{\infty}$, and we define $[\![P]\!](\sigma, s)$ respectively as σ' and $\perp$. Finally we define the *microscopic* semantics by

$$P \equiv_m Q \iff \forall \sigma, s \quad [\![P]\!](\sigma, s) = [\![Q]\!](\sigma, s)$$

Macroscopic equivalence If we adopt the macroscopic viewpoint of the user, we have no control on the context stack. The only thing we can do is to run a program with various inputs, but from a trivial initial context where all PEs are active. It is therefore meaningful to define that two program are equivalent in this sense if they yield the same outputs from the same inputs when started in this "coldbooted" context stack ε. We define thus

$$P \equiv_M Q \iff \forall \sigma \quad [\![P]\!](\sigma, \varepsilon) = [\![Q]\!](\sigma, \varepsilon).$$

Relativization In the sequel, we will often need to introduce *auxiliary variables* to hold temporary values. The final contents of such variables are not significant, only their side-effects are. We define thus a slight refinement of the equivalences above to take this fact into account: two programs are equivalent if they generate from the same input data the same result data, disregarding the final values of the auxiliary variables.

Let V be a set of variables, the auxiliary variables. Let σ^V be the environment σ where all variables of V are set to some fixed dummy value. For completeness, define $\bot^V = \bot$. The relativized versions of the equivalences above are then defined as follows.

$$P \equiv_M^V Q \iff \forall \sigma \quad ([\![P]\!](\sigma, \varepsilon))^V = ([\![Q]\!](\sigma, \varepsilon))^V$$

$$P \equiv_m^V Q \iff \forall \sigma, s \quad ([\![P]\!](\sigma, s))^V = ([\![Q]\!](\sigma, s))^V.$$

Observe that the original definition is obtained with $V = \emptyset$. More generally, assume that $W \subseteq V$. Then the following holds.

$$P \equiv_M^W Q \Rightarrow P \equiv_M^V Q$$

$$P \equiv_m^W Q \Rightarrow P \equiv_m^V Q$$

3 Studying program equivalences

3.1 Consistency of the equivalences

Our goal is now to express that the macroscopic and the microscopic equivalences are related in an intrinsic way, without reference to the underlying semantics. We have in fact the following theorem.

Theorem 1 *(i)*

$$P \equiv_m^V Q \Rightarrow P \equiv_M^V Q$$

(ii) Let B be a variable which appears neither in P nor in Q.

$$\texttt{where } B \texttt{ do } P \texttt{ end} \equiv_M^V \texttt{where } B \texttt{ do } Q \texttt{ end} \Rightarrow P \equiv_m^V Q$$

Observe that the converse of *(i)* is false as shown by the following example.

P :: `shift` X `to North; shift` X `to South`
Q :: `skip`

We have $P \equiv_M Q$. Let U be such that $U|_{(0,0)}$ is true and U is false everywhere else. Let $\sigma(X)|_{(x,y)} = x$ and $s = push(U, \varepsilon)$. Let $\sigma' = [\![P]\!]_m(\sigma, s)$ and $\sigma'' = [\![Q]\!]_m(\sigma, s)$. We have $\sigma'(X)|_{(0,0)} = 1$ and $\sigma''(X)|_{(0,0)} = 0$. The original value of $X|_{(0,0)}$ gets lost, as the neighbor PE is passive and forgets any incoming value.

Also, observe that the restriction on B is crucial, as shown by the following example.

$$\begin{array}{lll} P & :: & \texttt{where } \neg B \texttt{ do } X := 0 \texttt{ end} \\ Q & :: & \texttt{where } \neg B \texttt{ do } X := 1 \texttt{ end} \end{array}$$

Then $\texttt{where } B \texttt{ do } P \texttt{ end} \equiv_M \texttt{where } B \texttt{ do } Q \texttt{ end}$ holds, but $P \equiv_m Q$ is obviously false.

3.2 The microscopic equivalence is stable

The microscopic equivalence (m-equivalence) enjoys some interesting properties of *stability*: under certain smooth conditions, substituting a piece of a program with another m-equivalent piece yields a new program which is m-equivalent to the original one. We say that a program P *does not depend* on a set of variables V if

$$\forall \sigma, \sigma' \quad \forall X \notin V \ \sigma(X) = \sigma'(X) \Rightarrow [\![P]\!](\sigma) = [\![P]\!](\sigma')$$

This is obviously the case as soon as no variable of V appears in P. A similar notion can be defined for expressions B.

Theorem 2 *Assume $P \equiv_m^V Q$. Then the following holds.*

$$\begin{array}{rl} P; R \equiv_m^V Q; R & \text{provided } R \text{ does not depend on } V \\ R; P \equiv_m^V R; Q & \\ \texttt{while } B \texttt{ do } P \texttt{ end} \equiv_m^V \texttt{while } B \texttt{ do } Q \texttt{ end} & \text{provided } B, P \text{ and } Q \text{ do not depend on } V \\ \texttt{where } B \texttt{ do } P \texttt{ end} \equiv_m^V \texttt{where } B \texttt{ do } Q \texttt{ end} & \end{array}$$

Observe that if we have no auxiliary variable ($V = \emptyset$), then $\equiv_m^V$ is a *congruence* in the usual sense. Observe that the restriction on B for the `while` is crucial. Let $V = \{B\}$, $P :: B := tt$ and $Q :: B := ff$. Then $P \equiv_m^V Q$, but it is obviously false that $\texttt{while } B \texttt{ do } P \texttt{ end} \equiv_m^V \texttt{while } B \texttt{ do } Q \texttt{ end}$.

Apart from being stable, the m-equivalence enjoys a very interesting property: any L program can be transformed into a m-equivalent program in *normal form*. In this normal form, the `where` blocks are made of exactly one assignment. In particular, there is no nested conditioning, and no communication is conditioned. In some sense, all `where`'s have been "pushed" inside and "merged" into the program. Thanks to the stability properties of the m-equivalence m-equivalence, we can describe the normalization process in an incremental way. Auxiliary variables will play a crucial role in the transformation, but observe that the transformed programs do not depend on them. Let V be an *infinite* set of *new* variables which appear nowhere in the program being normalized. We denote variables of V with primes.

skip

$$\texttt{where } B \texttt{ do skip} \equiv_m^V \texttt{skip}$$

Communication

$$\texttt{where } B \texttt{ do shift } X \texttt{ along } d \texttt{ end} \quad \equiv_m^V \quad \begin{array}{l} B' := B; X' := X; \\ \texttt{shift } X \texttt{ along } d; \\ \texttt{where } \neg B' \texttt{ do } X := X' \texttt{ end} \end{array}$$

(Observe that B may depend on X. Observe also that

$$\begin{array}{l} B' := B; X' := X; \\ \texttt{shift } X' \texttt{ along } d; \\ \texttt{where } B' \texttt{ do } X := X' \texttt{ end} \end{array}$$

does not work: it is $\equiv_M^V$, but not $\equiv_m^V$!)

Sequencing

$$\texttt{where } B \texttt{ do } P; Q \texttt{ end} \quad \equiv_m^V \quad \begin{array}{l} B' := B; \\ \texttt{where } B' \texttt{ do } P \texttt{ end}; \\ \texttt{where } B' \texttt{ do } Q \texttt{ end} \end{array}$$

Iteration

$$\texttt{where } B \texttt{ do while } C \texttt{ do } P \texttt{ end end} \quad \equiv_m^V \quad \begin{array}{l} B' := B; \\ \texttt{while } C \texttt{ do} \\ \quad \texttt{where } B' \texttt{ do } P \texttt{ end} \\ \texttt{end} \end{array}$$

Conditioning

$$\texttt{where } B \texttt{ do where } C \texttt{ do } P \texttt{ end end} \quad \equiv_m^V \quad \texttt{where } (B \wedge C) \texttt{ do } P \texttt{ end}$$

3.3 The macroscopic equivalence

The macroscopic equivalence (M-equivalence) is *not* stable in general. Conditioning two M-equivalent programs may result in programs which are no longer M-equivalent! Yet, this is in some sense the only problem, as expressed by the following theorem.

Theorem 3 *Assume $P \equiv_M^V Q$. Then the following holds.*

$$\begin{array}{rl} P; R \equiv_M^V Q; R & \text{provided } R \text{ does not depend on } V \\ R; P \equiv_M^V R; Q & \\ \texttt{while } B \texttt{ do } P \texttt{ end} \equiv_M^V \texttt{while } B \texttt{ do } Q \texttt{ end} & \text{provided } B, P \text{ and } Q \text{ do not depend on } V \end{array}$$

It is therefore tempting to define a stable equivalence, say $\equiv$, as follows. Let $R[P]$ denote a program R which contains some occurrences of a program P. Then state that for any pair of programs P and Q, $P \equiv Q$ iff for any program $R[P]$, substituting Q for P within the text of $R[P]$ yields a program $R[Q]$ such that $R[P] \equiv_M^V R[Q]$. It is clear that $\equiv$ *is* stable by virtue of its definition, and that it is the largest contained in $\equiv_M^V$. It turns out that this equivalence is also contained in $\equiv_m^V$, the microscopic equivalence!

Theorem 4 *Let $\equiv$ be a stable equivalence contained in $\equiv_M^V$. Then it is contained also in $\equiv_m^V$.*

Proof Assume $P \equiv Q$. Let B be a variable which appears neither in P nor in Q. By the stability property, we have `where` B `do` P `end` $\equiv$ `where` B `do` Q `end`, too. Thus `where` B `do` P `end` $\equiv_M^V$ `where` B `do` Q `end`, and, by Theorem 1, $P \equiv_m^V Q$. □

It is nevertheless interesting to analyze precisely the reason why the M-equivalence is not preserved by conditioning. It turns out that it is due to *interactions* between PEs, either *explicit* (internal) as in the `shift` instruction, or *implicit* (external) as in the `while`. Let us consider three examples.

Example 1 Let β be a boolean array constant such that $\beta|_{(x,y)}$ is true if $x = 0$, and false otherwise. Let ξ be an array constant such that $\xi|_{(x,y)} = x$.

$$\begin{array}{lll} P & :: & \texttt{shift}\ X\ \texttt{along}\ E; \texttt{shift}\ X\ \texttt{along}\ W \\ Q & :: & \texttt{skip} \\ R[.] & :: & X := \xi; B := \beta; \texttt{where}\ B\ \texttt{do}\ [.]\ \texttt{end} \end{array}$$

Then $P \equiv_M^V Q$. But the final value of X at location $(0,0)$ is 1 in $R[P]$, whereas it is 0 in $R[Q]$. In order to preserve M-equivalence, a conditioned program should contain at most one `shift`.

Example 2 As above, consider

$$\begin{array}{lll} P & :: & X := \xi + 1; \texttt{shift}\ X\ \texttt{along}\ E \\ Q & :: & \texttt{skip} \\ R[.] & :: & X := \xi; B := \beta; \texttt{where}\ B\ \texttt{do}\ [.]\ \texttt{end} \end{array}$$

Then $P \equiv_M^V Q$. But the final value of X at location $(0,0)$ is -1 in $R[P]$, whereas it is 0 in $R[Q]$. In order to preserve M-equivalence, a non-atomic conditioned program should not contain any `shift` instruction, except possibly its first one.

Example 3 Let γ be a boolean array constant such that $\gamma|_{(x,y)}$ is true if $x = 1$ and false otherwise. As above, consider

$$\begin{array}{lll} P & :: & \texttt{while}\ C\ \texttt{do}\ C := \beta\ \texttt{end} \\ Q & :: & \texttt{while}\ C\ \texttt{do}\ C := \gamma\ \texttt{end} \\ R[.] & :: & C := \beta; B := \beta; \texttt{where}\ B\ \texttt{do}\ [.]\ \texttt{end} \end{array}$$

Then $P \equiv_M^V Q$, as either they both terminate at once with the same value of C (namely false everywhere) or they both diverge. Yet, $R[P]$ diverges, whereas $R[Q]$ terminates in one iteration. In order to preserve M-equivalence, a conditioned program should not contain any `while` construct. In effect, such a construct implicitly specifies a possibly infinite number of interactions.

More formally, consider the following grammar

$$\begin{array}{rcl} \langle \textit{local-prog} \rangle & :: & X := E \\ & | & \texttt{where } B \texttt{ do } \langle \textit{local-prog} \rangle \texttt{ end} \\ & | & \langle \textit{local-prog} \rangle ; \langle \textit{local-prog} \rangle \\ \\ \langle \textit{safe-prog} \rangle & :: & \texttt{shift } X \texttt{ along } d \\ & | & \langle \textit{local-prog} \rangle \\ & | & \texttt{where } B \texttt{ do } \langle \textit{safe-prog} \rangle \texttt{ end} \\ & | & \langle \textit{safe-prog} \rangle ; \langle \textit{local-prog} \rangle \end{array}$$

and say that a program is *local* if it is a $\langle \textit{local-prog} \rangle$ and *safe* if it is a $\langle \textit{safe-prog} \rangle$. Informally, a local program is a linear program (that is a program without any loop) in L which contains no interaction. A safe program is a linear program which contains at most one interaction, in which case it is the first instruction to be executed. As expected, those programs behave more or less as atomic macro-instructions. In fact, they could as well be run asynchronously as an atomic block. This amounts thus to the well-known SPMD (Single Program Multiple Data) model!

Theorem 5 *Let P and Q be safe, such that $P \equiv^V_M Q$. Then*

$$\texttt{where } B \texttt{ do } P \texttt{ end} \equiv^V_M \texttt{where } B \texttt{ do } Q \texttt{ end}.$$

Corollary 6 *If P and Q are safe, then $P \equiv^V_M Q$ iff $P \equiv^V_m Q$.*

4 Advanced control structures

The microscopic semantics provides a convenient framework for expressing and studying non-trivial control structures through the explicit manipulation of the context stack. in this section, we will focus on the parallel extension of non-local control transfer structures such as `exit`, `return`, `break`, `continue` in `C`, or the `catch`/`exit` pair in Lisp.

4.1 Definition

For this purpose, we define a new construct `escape` n, where n is a syntactic positive constant, with the following effect. On executing `escape` n, each *active* PE becomes passive up to the end of the nth enclosing `where` block. In some sense, it *jumps* to the exit label of this block, and remains there, waiting for the other PEs.

This can be achieved in our semantics by masking the n upper levels of the context stack with the negation of the current context, that is the value at the top. Let s be a context stack, and let U be a boolean array value. We define the following functions.

$$\begin{array}{rcl} \textit{mask}(s, U, 0) & = & s \\ \textit{mask}(s, U, k+1) & = & \textit{push}(\textit{top}(s) \wedge U, \textit{mask}(\textit{pop}(s), U, k)) \\ \textit{escape}(s, k) & = & \textit{mask}(s, \neg\, \textit{top}(s), k) \end{array}$$

We can then give the microscopic semantics rule for escape.

Escape

$$\langle \texttt{escape}\ n, \sigma, s\rangle \longrightarrow_m \langle \bullet, \sigma, \mathit{escape}(s,n)\rangle$$

Observe this instruction is *conditional* in the following sense. Only the active PEs are affected by it. It is thus interesting to define an *unconditional* version of it, called `interrupt`. If *some* active PE executes the `interrupt` n instruction, then *all* PEs become passive up to the end of the nth enclosing `where` block. Define $\mathit{interrupt}(s,n) = \mathit{mask}(s, ff, n)$.

Interrupt

$$\frac{\bigvee_u \mathit{top}(s)|_u = tt \mid ff}{\langle \texttt{interrupt}\ n, \sigma, s\rangle \longrightarrow_m \langle \bullet, \sigma, \mathit{interrupt}(s,n)\rangle \mid \langle \bullet, \sigma, s\rangle}$$

The effect of the `escape` and the `interrupt` instructions is to restrict the context. It is thus important to define an iteration construct whose exiting is determined by the context, in opposite to the `while` iteration construct, whose exiting is determined by the value of the test without regard to the context. We call it `loop`.

Conditional iteration

$$\frac{\bigvee_u \mathit{top}(s)|_u = tt \mid ff}{\langle \texttt{loop}\ P\ \texttt{end}, \sigma, s\rangle \longrightarrow \langle P; \texttt{loop}\ P\ \texttt{end}, \sigma, s\rangle \mid \langle \bullet, \sigma, s\rangle}$$

Finally, observe that the `where` blocks will then play a new role in conjunction with the `escape` and `interrupt` instructions. In particular, the `where true do` P `end` construct will often be used to add an exit label at the end of P. For clarity, we give a specific name to this construct.

Execution

$$\texttt{exec}\ P\ \texttt{end} :: \texttt{where true do}\ P\ \texttt{end}$$

4.2 Examples and applications

Parallel factorials

We can recast the second example of section 1.3 in this new setting.

```
F := 1;
exec
    loop
        where (N ≤ 1) do escape 2 end;
        F := F * N;
        N := N − 1
    end
end
```

Parallel iteration

The POMPC language ([12]) defines an iteration construct called `whilesomewhere` B `do` P `end` which is both value- and context-determined. At each iteration, only those active PEs such that B evaluates to true execute the body P. The iteration stops when B evaluates to false at *all currently active* PEs. This effect can be achieved as follows.

```
exec
    loop
        exec
            where B do interrupt 2 end;
            interrupt 2
        end;
        where B do P end
    end
end
```

Parallel non-local loop exits

On executing the `break` instruction, active PEs become passive up to the end of the `whilesomewhere` construct. On executing the `continue` instruction, they become passive up to the end of the current iteration. Let k be the current number of enclosing `where` blocks within P at the instruction under consideration. We have

$$\begin{aligned} \texttt{break} &:: \texttt{escape}\,(k+2) \\ \texttt{continue} &:: \texttt{escape}\,(k+1) \end{aligned}$$

Parallel switching

This language defines a switching construct (for simplicity, we consider 3 cases only, with a default case).

`switchwhere` X `do case` $A_1 : P_1$ `case` $A_2 : P_2$ `case` $A_3 : P_3$ `default:` P_4 `end`

The intended effect is the following. The current value of X is evaluated and tested successively against the expression A_1, A_2 and A_3. Each PE then enters the body $P_1; P_2; P_3; P_4$

at the `case` label corresponding to its first match, `default` if none is found.

```
exec
  exec
    exec
      exec
        exec
          where (X = A1) do escape 2 end;
          where (X = A2) do escape 3 end;
          where (X = A3) do escape 4 end;
          where true do escape 5 end
        end;
        P1
      end;
      P2
    end;
    P3
  end;
  P4
end
```

The effect of a `break` in P_i is achieved by the instruction `escape` $(5 - i) + k$, where k is the current number of enclosing `where` blocks within P_i at the instruction under study.

Parallel conditional branching

This language also provides a parallel equivalent of the usual `if` B `then` P `else` Q `end` construct. It is denoted `where` B `do` P `elsewhere` Q `end`, and the intended effect corresponds to

```
B' := B;
where B' do P end;
where ¬B' do Q end
```

where B' is a *new* auxiliary variable (in practice, a temporary register value). We can easily express this construct in our setting as follows.

```
exec
    where B do P; escape 2 end;
    Q
end
```

In fact, the context stack is used here to hold the temporary value. Observe that P and Q do not have the same level of nesting, which may be error-prone. The following

symmetric expression is therefore probably better.

```
exec
    where B do P; escape 2 end;
    where true do Q end
end
```

Returning and exiting

The net effect of the `return` or `exit` instructions is to inhibit any activity in the currently active PEs up to the end of the (sub-)program. This effect can be easily achieved in our setting, thanks to an interesting side-effect of the *mask* function. Consider the expression $mask(s, U, n)$, where n is greater than the size of the stack s. The result is a stack *greater* than s, namely of size n. This holds because $pop(\varepsilon) = \varepsilon$, so that $push(top(\varepsilon), pop(\varepsilon))$ is of size 1, and not 0! In consequence, escaping n levels of `where` blocks, with n greater than the current number of enclosing blocks, will let the stack grow downwards. Everything happens as if the program were surrounded by an unbounded number of `exec` blocks, and the resulting effect is exactly as wanted. In this sense, we can express the `exit` and `return` instructions by `escape` $(k+1)$, where k is the number of enclosing `where` blocks.

5 Conclusion and future work

This work shows that it is possible to define a clean and simple programming model for massively parallel SIMD architectures, and to study it at a sufficiently abstract level. Section 4.2 demonstrate that it is general enough to model many control structures found in "real" SIMD languages. Nevertheless, it can be shown that it is minimal in the sense that none of its constructs can be simulated (up to $\equiv_m^V$) by some combination of the others. A main feature of this model is to give an account of the two views of SIMD architectures through program equivalences.

This work is a preliminary attempt and it could be improved in many aspects. The definition of the `while` B `do` P `end` construct of L should probably be modified so that the iteration stops when B evaluates to false at each *active* processor. Also, the `escape/interrupt` construct should probably use symbolic addresses instead of nesting depth. This would correspond to the notion of *tagged exception* in Lisp and other languages. The `break` and `continue` constructs would then be simulated by using two different tags. Also, the normal form of section 3.2 should be studied in more details. In particular, it is not clear whether it carries over to the advanced control structures. We are currently working on these problems.

A number of research directions remain to be explored. One could add to L scalar variables besides array variables, and define an interface between the two worlds through a pair of instructions `broadcast` x and `reduce` X `with` $\oplus$. We have considered only

`shift` communications. We could consider general communications too, as provided in the Connection Machine for instance. A possible approach is to equip the language L′ with a pair of instructions: `get` X `from` A and `send` X `to` A `with` $\oplus$. Finally, a major challenge is to use this work in proving the correctness of SIMD programs. The definition of the microscopic semantics and the existence of simple normal forms up to microscopic equivalence, as shown in section 3.2, is a definite step in this direction.

This study gives some insight into certain constructs found in "real" languages. The semantics of L′ guarantees that the memory of a passive PE cannot be modified, and that a passive PE may not become active again within the current block. Yet, real programming languages do not enjoy this property in general. In POMPC ([12]), a passive PE may receive a message from an active PE by the `send` instruction. In *Lisp ([17]), the form `*all` selects all PEs whatever their current context is. We believe that such behaviors should be avoided, and that languages should adhere to this discipline as a basic safety requirement. This is *not* to say that those constructs may not be useful under certain circumstances. The situation is rather reminiscent of the celebrated `GOTO` statement 20 years ago. A safe and structured programming discipline should probably avoid such irregular constructs.

Acknowledgments

The author wishes to thank Patrick Garda, Nicolas Paris and Gérard Berry for numerous and stimulating discussions. This work was done while the author was working at the LIENS, Paris. The support of the CNRS Coordinated Research Program C^3 and the comments of anonymous referees are gratefully acknowledged.

References

[1] Batcher K (1979) The design of a Massively Parallel Processor. IEEE Trans. on Computers C-29, 9, pp 836–840.

[2] Bougé L (1990) On the semantics of languages for massively parallel architectures. Rept. No. 90-13, LIENS, Paris, 1990.

[3] Bougé L, Garda P (1990) Towards a semantic approach to SIMD architectures and their languages. In: Semantics of Systems of Concurrent Processes, Proc. 18*th* Spring School of the LITP, Lect. Notes Comp. Science 469, Springer, pp 142–175.

[4] Connection Machine Model CM-2 Technical Summary (1987) Techn. Rept. HA 87-4, Thinking Machine Corp.

[5] Fountain TJ (1981) CLIP4: A Progress Report. In: Duff MJB, Levialdi, S (eds) Languages and Architectures for Image Processing, Academic Press, pp 283–291.

[6] Frenkel K (1986) Evaluating two massively parallel machines. Comm. ACM 29, 8, pp 752–758.

[7] Hillis WD (1985) The Connection Machine. MIT Press, Cambridge, Mass.

[8] Hockney RW, Jesshope CR (1988) Parallel Computers 2: Architectures, Programming and Algorithms. IOP Publishing Ltd.

[9] Hunt DJ (1981) The ICL DAP and its application to image processing. In: Duff MJB, Levialdi S (eds) Languages and Architectures for Image Processing Academic Press, pp 275–282.

[10] Lua KT, Wong WF (1987) Geometric Arithmetic Parallel Processor — An Evaluation. Proc. Interdepartment Seminar on Supercomputers and Applications, Publ. TRIO/87, Dept. Information Systems and Comp. Science, Nat. Univ. Singapore, pp 44–61.

[11] Olderog E-R, Apt K (1988) Fairness in parallel programs: the transformational approach. ACM Trans. on Progr. Lang. and Systems 10, 3, pp 420–455.

[12] Paris N (1990) Définition de POMPC (version 1.5). Typescript, LIENS, Paris.

[13] ParIS: The C Interface. Reference Manual (1987) Thinking Machine Corp.

[14] Perrott RH (1979) A language for array and vector processors. ACM Trans. Progr. Lang. 1, 2, pp 177–195.

[15] Reeves RW (1985) Parallel Pascal and the Massively Parallel Processor. In: Potter JL (ed) The Massively Parallel Processor), MIT Press, pp 230–260.

[16] Steele GH, Hillis WD (1986) Connection Machine Lisp : Fine-Grain Parallel Symbolic Processing. Proc. 1986 ACM Conf. on Lisp and Funct. Progr., Cambridge, Mass., pp 279–297.

[17] *Lisp Language Reference Manual (1988) Thinking Machine Corp.

A Denotational Real-Time Semantics for Shared Processors

Jozef Hooman*

Dept. of Mathematics and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
e-mail: wsinjh@win.tue.nl

Abstract

To describe the real-time behaviour of an Occam-like real-time language with concurrency and synchronous message passing, a denotational semantics is presented. New in this paper is the generalization of the maximal parallelism model, where each process has its own processor, to multiprogramming where several processes may share a processor and statements are scheduled according to dynamic priorities. Our compositional semantics based on several assumptions about the scheduling policy and the communication mechanism. These assumptions are motivated by an operational description of program execution.

1 Introduction

The aim of this paper is a formal, denotational, semantics to describe the timing behaviour of programs. Determining this real-time behaviour requires more information about implementation details than is usual in non-real-time models. For instance, in a non-real-time semantics one can abstract from details about the execution time of statements and the implementation of parallelism (such as scheduling policies). This abstraction is no longer possible when time can be observed. The timing of a parallel program clearly depends on the assignment of processes to processors and on the scheduling policy.

As a starting point for the formal description of distributed real-time systems, several papers have used the *maximal parallelism* model where it is assumed that each process has its own processor. This assumption is used in [KSR+88] where a denotational semantics for a real-time version of CSP is given, based on the linear history semantics of [FLP84]. A fully abstract version of this semantics has appeared in [HGR87]. Reed and Roscoe [RR87] give a hierarchy of timed models, based on a complete metric space structure. A fully abstract timed failure semantics for an extended CSP language has

*Partially supported by ESPRIT-BRA project 3096 (SPEC).

been developed in [GB87]. To prove that a program satisfies a specification expressing real-time properties, compositional proof systems have been given in [HW89,Hoo90], all based on the maximal parallelism assumption.

In practice, however, many applications deal with uniprocessor implementations where several processes share a single processor and actions are scheduled according to some scheduling policy. As a first study to investigate the precise timing behaviour of such implementations, we consider in this paper a programming language with a construct to express that (part of) a program, possibly containing parallel processes, is executed on a single processor. By means of this construct we can distinguish between parallel processes executing on a single processor and concurrent processes each executing on their own processor. Parallelism on one processor is in principle modelled by an arbitrary interleaving of atomic actions. This interleaving can be restricted by assigning priorities to statements.

The ultimate aim of our work is to develop a formal framework for the specification and compositional verification of real-time properties of programs. Compositionality can be considered as a prerequisite of hierarchical, structured, program derivation. It requires that the properties of a compound programming construct can be derived from properties of its constituent parts without any further information about the internal structure of these parts. A good starting point for such a compositional proof theory is the formulation of a denotational, and hence compositional, semantics. In such a semantics the meaning of a program (or program-part) must be defined without any information about the environment in which it will be placed. Hence, this semantics must characterize all possible executions of the program in any environment. When composing the program with (part of) its environment, a number of these executions can be removed, since more information about the environment becomes available.

To be able to describe all potential computations of a statement and to select the correct executions at composition, it is often needed to add, so called, non-observable entities to the denotations. For instance, in the maximal parallelism model we must be able to express when a program is waiting for a communication. (The need for this additional information in a compositional framework follows from the fully abstract semantics given in [HGR87].) In general, any influence of the environment on the behaviour of a program must be made explicit in the semantics of that program. The introduction of shared processors and priorities strongly increases the dependency of a program on its environment. E.g., certain statements that are ready to execute will not be executed, since they have a low priority and at most one statement can be executed at a time on a uniprocessor. Modelling the timing behaviour of such statements requires that the semantics contains primitives to state explicitly when a statement is executing and when it is requesting processor-time with a certain priority. By adding this information, we achieve in this paper a denotational real-time semantics for our programming language. Such a semantics requires several assumptions, for instance, about the execution time of atomic statements, the implementation of the communication mechanism, and the

scheduling strategy. These assumptions are motivated by an operational description of the execution of programs. This operational meaning has been inspired by the implementation of Occam-programs on transputers.

This paper is structured as follows. Section 2 contains the syntax of our real-time programming language. The informal meaning of programs is discussed, including a number of questions about the precise execution model. To answer these questions, we describe an implementation of the language in Section 3. To emphasize the essential new points of our framework, in Section 4 we remove the program variables from the programming language. There we also mention the basic timing assumptions. A denotational semantics for the programming language can be found in Section 5. Section 6 contains a conclusion and an indication of future work.

2 Real-Time Programming Language

In this section we describe our real-time programming language. Section 2.1 contains the syntax of this language and an informal semantics. In Section 2.2 we show that this informal description leads to a number of questions about the precise meaning of programs.

2.1 Syntax and Informal Semantics

Our programming language is akin to OCCAM [Occ88b], with communication by synchronous message passing, that is, both sender and receiver wait until a corresponding partner is available. Furthermore, the language includes *delay*-statements by which a process can release the processor for a certain period of time. By allowing such delay-statements to occur in the guard of a guarded command, we can program a time-out, i.e., restrict the waiting period for a communication.

Let $CHAN$ be a nonempty set of channel names. We assume that channels are unidirectional and that they connect exactly two processes. Let VAR be a nonempty set of (names for) program variables, and $\mathbb{N}$ be the set of natural numbers (including 0). The syntax of our programming language is given in Table 1, where e is an expression, yielding a non-negative value, b and b_i are boolean expressions, $n \in \mathbb{N}$, $n \geq 1$, $c, c_1, \ldots c_n \in CHAN$, and $x, x_1, \ldots x_n \in VAR$.

Table 1: Syntax Programming Language

Statement	$S ::=$	$\mathbf{skip} \mid x := e \mid \mathbf{delay}\ e \mid c!e \mid c?x \mid$
		$S_1; S_2 \mid G \mid *G \mid \mathbf{prio}\ e\ (S) \mid S_1 \,\|\|\, S_2$
Guarded-Command	$G ::=$	$[]_{i=1}^{n} b_i \rightarrow S_i] \mid [][]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i \,[]\, b; \mathbf{delay}\ e \rightarrow S]$
Network	$N ::=$	$\ll S \gg \mid N_1 \| N_2$
Program	$P ::=$	$S \mid N$

Informally, the statements of our programming language have the following meaning:

Atomic statements

- **skip** terminates immediately.
- $x := e$ is used to assign the value of expression e to the variable x.
- **delay** e suspends execution for (the value of) e time units.
- $c!e$ is used to send the value of expression e on channel c as soon as a corresponding input command is available. Since we assume synchronous communication, such an output statement is suspended until a parallel process executes a statement $c?x$.
- $c?x$ is used to receive a value via channel c and assign this value to the variable x. Similar to the output command, such an input statement has to wait for a corresponding partner before a communication can take place.

Compound statements

- Sequential composition $S_1; S_2$.
- Guarded command $[[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i \,[]\, b; \textbf{delay}\ e \rightarrow S]$. A guard (the part before the arrow) is *open* if the boolean part evaluates to true. If none of the guards is open, the guarded command terminates after the evaluation of the booleans. Otherwise, wait until an input-command of an open input-guard can be executed and continue with the corresponding S_i. If the delay-guard is open (b evaluates to true) and no input-guard can be taken within e time units after evaluation of the booleans, then S is executed.
- Iteration $*G$ indicates repeated execution of guarded command G as long as at least one of the guards is open. When none of the guards is open $*G$ terminates.
- **prio** e (S) assigns priority e to statement S. Statements without such an explicit priority assignment obtain priority 0. A higher number corresponds to a higher priority.
- $\ll$ S $\gg$ is called processor closure; it expresses that program S has its own processor and no process outside S executes on this processor.
- $P_1 \parallel P_2$ indicates parallel execution of P_1 and P_2. If this operator occurs inside the brackets $\ll$ and $\gg$ of processor closure, then it expresses uniprocessor parallelism and the statements P_1 and P_2 are executed on the same processor. Otherwise, the networks P_1 and P_2 are executed concurrently on disjoint sets of processors. No variable should occur in both P_1 and P_2.

Observe that parallel processes are not allowed to have shared variables. Hence, to achieve a uniform framework, parallel processes that are executed on a single processor may only communicate by synchronous message passing. An implementation of this communication mechanism could, however, use shared variables. Further, for **prio** e (S) we require that S does not contain any parallel composition operator.

Henceforth we use $\equiv$ to denote syntactic equality. For a guarded command $[[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i \,[]\, b; \textbf{delay}\ e \rightarrow S]$, we write $[[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i]$ if $b \equiv \mathit{false}$. Further, boolean expressions equivalent to *true* are omitted in guards.

2.2 Examples and Questions

Observe the difference between $\ll x := 5 \,||\, y := 6 \gg$, expressing an interleaved execution of the two assignments, and $\ll x := 5 \gg \,||\, \ll y := 6 \gg$, expressing true concurrency. Although we have given an informal explanation of programs, there are a number of questions about the precise meaning of programs. For instance,

- When and how are statements interrupted to allow the execution of statements with a higher priority?
- How are communications performed? By the main processor or by special IO-devices?
- Is there any distinction between internal communications (within a single processor) and external communications that connect two processors?
- What is the priority of internal communications?
  Example 2.1 Consider $\ll$ **prio** 1 $(c!0) \,||$ **prio** 2 $(x := 2) \,||$ **prio** 3 $(c?y) \gg$.
  Should the priority of the c-communication be the maximum of the priorities of the two partners, or should it be the minimum—first executing the assignment? □
- What is the priority of external communications? What is the relation between priorities on different processors?
  Example 2.2 Consider the following program
  $\ll$ **prio** 1 $(c!0) \,||$ **prio** 5 $(d?x) \gg \,||\, \ll$ **prio** 4 $(c?y) \,||$ **prio** 3 $(d!1) \gg$.
  In which order are the two communications executed, or are they performed concurrently? Or maybe this program leads to deadlock? Is it significant that the maximum of the priorities of the statements for the d-communication is higher than the priorities of the statements for the c-communication?
  Compare this program with the following one:
  $\ll$ **prio** 1 $(c!0) \,||$ **prio** 2 $(d?x) \gg \,||\, \ll$ **prio** 2 $(c?y) \,||$ **prio** 1 $(d!1) \gg$.
  Is there any difference between the execution of the two programs above?
  Consider also the program
  $\ll$ **prio** 1 $(c!0) \,||$ **prio** 2 $(x := 4) \gg \,||\, \ll$ **prio** 3 $(c?y) \,||$ **prio** 2 $(z := 5) \gg$.
  Is the c-communication performed before or after the assignments? □
- What is the precise meaning of delay-guards and delay-statements?
  Example 2.3 Consider the following program:
  $\ll$ **prio** 1 $([c?x \to S_1 \,[]\, \textbf{delay } 4 \to S_2]) \,||$ **prio** 3 $(y := 1;\, *[y < 6 \to y := y + 1]) \gg$
  $||\, \ll c!0 \gg$.
  Note that the first process contains a time-out of 4 time units. When does this time-out period exactly start? Is S_2 always executed, assuming that the second process (which has a higher priority) takes more than 4 time units? Compare it with the following program where the high-priority process contains a delay-statement:
  $\ll$ **prio** 1 $([c?x \to S_1 \,[]\, \textbf{delay } 4 \to S_2])$
  $\quad ||$ **prio** 3 $(y := 1\,; *[y < 6 \to y := y + 1; \textbf{delay } 1]) \gg \,||\, \ll c!0 \gg$.
  Will the second process, with priority 3, release the processor by the delay-statement, thus allowing the execution of statements with a lower priority? □

- Finally, there are the usual questions about fairness assumptions.
  Example 2.4 Consider $\ll *[true \rightarrow c?x] \,||\, d?y \gg$. Is it possible to execute the first loop forever, neglecting the d-communication forever? □

3 Operational Behaviour

To answer the questions from the previous section about the informal meaning of programs, we give an operational description of program execution. This operational interpretation of programs also motivates the assumptions made in our formal denotational semantics. Our operational model is inspired by the implementation of Occam on transputers. Therefore we briefly describe the main ideas behind this implementation in Section 3.1. The execution model is then given in Section 3.2. Based on this operational description, the questions from Section 2.2 are answered in Section 3.3.

3.1 Occam Implementation on Transputers

In this section we describe an Occam implementation on transputers (see, for instance, [Occ88a]) as far as it is relevant for our operational semantics. A transputer is a processor with internal memory and (four) *communication links* for connection with other transputers. Each link implements two channels (in opposite direction). Communication links are connected to the main processor via *link interfaces*. These interfaces can, independently, manage the communications of the link, including direct access to memory. As a result of this architecture, a transputer can simultaneously communicate on all links (in both directions) and execute an internal statement. Much of the power of the transputer comes from this facility.

On transputers there is a clear distinction between the implementation of internal channels, which are within a single transputer, and external ones that must be mapped onto links. Assume each statement has a unique identification number (*id*). Internal channels are implemented by a single word of storage. Initially, this memory location is loaded with a special value *nil* (distinct from any *id*). The first process that is ready to communicate finds this value *nil*, places its *id* in the location and becomes suspended. The second process that is ready to communicate finds the *id* of the first process and performs the communication. External channels are implemented as follows. When a process on the main processor tries to communicate on an external channel its execution becomes suspended and the main processor delegates responsibility to the autonomous link interface. If the interfaces on both processors are ready to communicate, the message is transferred and both processes involved become executable.

3.2 Execution Model

We describe an execution model for our programming language which will be based on ideas from the previous section, but differs in many respects from the Occam implemen-

tation on transputers. For instance, on transputers there are only two levels of priority which are statically assigned to processes, whereas we use dynamically changing priorities ranging over the value domain of program variables. Further, we do not distinguish between internal and external channels. Conceptually the mechanism of external channels is used, assuming that all communications are performed by special link interfaces.

We describe the execution of $\ll S \gg$, i.e., the execution of program S on a single processor. First, all skip-statements are removed from S, and all priority assignments are distributed such that for each statement there is an expression that determines its priority. The priority of a statement is derived from the closest surrounding priority assignment that yields a positive value. If the statement is not embedded in a priority assignment then the priority is 0. These transformations lead to a program P.

Execution of a program is defined in terms of *basic statements*, that is, assignments, delay-statements, input- and output-statements (io-statements), and guarded commands. Assume that for a given program P each occurrence of a basic statement in P has a unique identification number (*id*). We use $s, \hat{s}, s_0, s_1, \ldots$ to denote such a number, and $id(S)$ to denote the identification number of S. Henceforth we will often identify an *id* and the corresponding statement.

Let s be a basic statement in P. During execution of P we can determine the priority of basic statement s in the current state, denoted by $Prio(s)$. Similarly, we can determine the set of basic statements that can be executed in the current state immediately after the execution of s. Let $Follow(s)$ denote the set of *id*'s of these statements. For instance, for a guarded command this set is defined to consist of the statements that are ready to be executed when all the boolean guards evaluate to false. For a statement S in P (S need not be a basic statement), we define $First(S)$ as the set of the *id*'s of the first basic statements from P that can be executed in the current state when control is immediately before S. Let $Value(e)$ and $Value(b)$ yield the value of expressions e and b, respectively, in the current state. Assume that the current time can be read using the variable *current_time*. On each processor we have

- A set *requesting* of pairs (s, p), where s is an *id* and p is a priority. This set represents the executable statements that request processor-time.
- A set *delayed* of pairs (s, exp_time) and triples (s_1, s_2, exp_time) where s, s_1, and s_2 are *id*'s and *exp_time* is the expiration time of the statement; a triple (s_1, s_2, exp_time) corresponds to a delay-statement s_2 in a guarded command s_1.
- For each output channel c there is a special variable $send(c)$ which is either *nil* or some *id*.
- For each input channel c there is a special variable $rec(c)$ which is either *nil*, or some *id* s, or a pair of *id*'s (s_1, s_2). Here (s_1, s_2) corresponds to an input-guard s_2 in a guarded command s_1.

Note that if a delay-guard s_2 in a guarded command s_1 expires, which corresponds to a triple (s_1, s_2, exp_time) in *delayed* with $exp\_time \leq current\_time$, then all input attempts of guarded command s_1 must be removed. That is, all variables $rec(c)$ which

have a value (s_1, s_3) must be set to *nil*. By means of $Follow(s_2)$ the next executable statements are determined. Similarly, when a communication along channel c is performed and $rec(c) = (\hat{s}, s_2)$, then all other input attempts of guarded command $\hat{s}$ are discarded and any triple $(\hat{s}, s_4, exp\_time)$ in *delayed* must be removed. Hence the main processor and the link interfaces may all change the variables $rec(c)$ and the set *delayed*. To guarantee that the decision by a link interface to perform a communication and the updating of the *rec* variables and *delayed* are performed atomically, we use statements LOCK and RELEASE to obtain exclusive read/write access to these *rec* variables and the set *delayed*.

The execution of the program P is described by the following algorithm:

$requesting := \{(s, Prio(s)) \mid s \in First(\hat{S})\}$; $delayed := \emptyset$;
For all output channels c, $send(c) := nil$; For all input channels c, $rec(c) := nil$;
REPEAT
IF $requesting \neq \emptyset$ THEN non-deterministically select a pair $(s, p) \in requesting$
such that $p = max(\{p_0 \mid (s_0, p_0) \in requesting\}$;
$requesting := requesting - \{(s, p)\}$;
CASE s corresponds to the following statement:

- $x := e$ DO execute $x := e$;
  $requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s)\}$
- **delay** e DO $delayed := delayed \cup \{(s, current\_time + value(e))\}$
- $c!e$ DO $send(c) := s$
- $c?x$ DO LOCK $rec(c) := s$ RELEASE
- $[]_{i=1}^{n} b_i \rightarrow S_i]$ DO
  IF for all $i \in \{1, \ldots, n\}$, $Value(b_i) = false$
  THEN $requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s)\}$
  ELSE non-deterministically select an i such that $Value(b_i) = true$;
  $requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in First(S_i)\}$
- $[]_{i=1}^{n} b_i; c_i?x_i \rightarrow S_i \,[]\, b;$ delay $e \rightarrow S_0]$ DO
  IF for all $i \in \{1, \ldots, n\}$, $Value(b_i) = false$ and $Value(b) = false$
  THEN $requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s)\}$
  ELSE for all $i \in \{1, \ldots, n\}$; LOCK
  IF $Value(b_i) = true$ THEN $rec(c_i) := (s, id(c_i?x_i))$
  IF $Value(b) = true$ THEN $delayed := delayed \cup$
  $\{(s, id(\text{delay } e), current\_time + value(e))\}$
  RELEASE

ENDCASE
For each $(s, exp\_time) \in delayed$ with $exp\_time \leq current\_time$:
$requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s)\}$;
$delayed := delayed - \{(s, exp\_time)\}$;

LOCK For each $(s_1, s_2, exp\_time) \in delayed$ with $exp\_time \leq current\_time$:
for all c, IF $rec(c) = (s_1, s_3)$, for some s_3 THEN $rec(c) := nil$;
$requesting := requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s_2)\}$;
$delayed := delayed - \{(s_1, s_2, exp\_time)\}$
RELEASE
UNTIL $requesting = \emptyset$ and $delayed = \emptyset$ and for all c: $send(c) = rec(c) = \emptyset$

In parallel with this algorithm on the main processor, the link interfaces perform a communication as soon as both partners are ready, i.e. have set their *send* and *rec* variables. Let $P_k.send(c)$, $P_k.rec(c)$, $P_k.requesting$ and $P_k.delayed$ be the variables on processor P_k, then the interfaces establish the following:
LOCK
IF $P_i.send(c) = s_1 \neq nil$ and either $P_j.rec(c) = s_2 \neq nil$ or $P_j.rec(c) = (\hat{s}, s_2)$
THEN IF $P_j.rec(c) = (\hat{s}, s_2)$ THEN for all channels d:
IF $P_j.rec(d) = (\hat{s}, s_3)$, for some s_3 THEN $P_j.rec(d) := nil$;
$P_j.delayed := P_j.delayed - \{(\hat{s}, s_4, exp\_time) \mid$ for all s_4 and $exp\_time\}$;
$P_i.send(c) := nil$; $P_j.rec(c) := nil$
RELEASE
The communication along c is performed;
$P_i.requesting := P_i.requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s_1)\}$;
$P_j.requesting := P_j.requesting \cup \{(s_0, Prio(s_0)) \mid s_0 \in Follow(s_2)\}$
ELSE RELEASE

Note that, for uniformity, we allow $i = j$ although such an internal communication might be implemented more efficiently.

3.3 Answers to Questions

Using the execution model from the previous section, the questions from Section 2.2 can be answered. Observe that priorities are only taken into account at the start and termination of the execution of basic statements; then a statement can only execute if there are no other statements with a higher priority on the same processor which request processor-time. The execution of an assignment, a delay-statement, or an io-statement cannot be interrupted. After the execution of an io-statement there are two possibilities: either it waits for a partner or it starts the communication if a partner is available. In both cases other statements can be executed simultaneously. Observe that:

- The priority of an internal communication is the minimum of the priorities of both communication statements, since both partners must have been executed before the communication takes place.

  Example 3.1 Consider again $\ll$ prio 1 $(c!0) \parallel$ prio 2 $(x := 2) \parallel$ prio 3 $(c?y) \gg$. According to the algorithm above, first the input statement is executed because it has the highest priority. It will update the variable $rec(c)$. Next the assignment has the highest priority and is executed. Finally, the output statement is the only

requesting statement and after its execution the communication along channel c takes place. □

- In case of equal priorities a nondeterministic choice is made (to abstract from specific scheduling policies). No fairness assumptions are made.

  Example 3.2 In the program $\ll *[true \rightarrow c?x] \parallel d?y \gg$ it is possible that the input statement $d?y$ is never executed. □

- The priorities of statements on different processors are incomparable. Only the relative ordering of priorities on a single processor determines the execution order on that processor.

  Example 3.3 Consider
  $\ll$ **prio** 1 $(c!0) \parallel$ **prio** 5 $(d?x) \gg \parallel \ll$ **prio** 4 $(c?y) \parallel$ **prio** 3 $(d!1) \gg$.
  Then first the two input statements are executed on each processor. Next the two output statements are executed and, assuming that the execution times of statements on both processors are equal, the two communications are performed simultaneously. The program above has exactly the same behaviour as
  $\ll$ **prio** 1 $(c!0) \parallel$ **prio** 2 $(d?x) \gg \parallel \ll$ **prio** 2 $(c?y) \parallel$ **prio** 1 $(d!1) \gg$.
  Compare this with
  $\ll$ **prio** 1 $(c!0) \parallel$ **prio** 2 $(x := 4) \gg \parallel \ll$ **prio** 3 $(c?y) \parallel$ **prio** 2 $(z := 5) \gg$
  where first the assignment to x is performed concurrently with the execution of the input statement. As soon as the assignment to x has terminated the output statement is executed, and then—when the input statement has terminated—the communication takes place. Note that, depending on the execution times, the c-communication might overlap in time with the assignment to z. □

4 Programming Language and Timing Assumptions

To highlight the essential points of our denotational semantics, we remove the variables from the programming language. The techniques to deal with program variables are well-known (see e.g, [Zwi89]), and our semantics can be extended easily with states that give the values of the variables (see also [Hoo91]). Hence we use $c!$ for an output action and $c?$ for input. Instead of assignments, we use a statement **atomic**(t) that represents an atomic action with an execution time of t time units. Let $TIME$ be some denumerable domain of time points, and $PRIO$ a set of priority values. The syntax of our programming language is given in Table 2, where $c, c_i \in CHAN$, $t \in TIME$, $p \in PRIO$, and $n \in I\!N$.
Let $DCHAN$ be the set of channels extended with, so called, directed channels:
$DCHAN = CHAN \cup \{c! \mid c \in CHAN\} \cup \{c? \mid c \in CHAN\}$.

Definition 4.1 (Channels Occurring in Statement) The set of (directed) channels occurring in a statement S, notation $dch(S)$, is defined as the smallest subset of $DCHAN$ such that if c is an output channel of S then $\{c, c!\} \subseteq dch(S)$, and if c is an input channel of S then $\{c, c?\} \subseteq dch(S)$.

Table 2: Syntax Programming Language

Statement	$S ::=$	$\mathbf{skip} \mid \mathbf{atomic}(t) \mid \mathbf{delay}\, t \mid c! \mid c? \mid$
		$S_1; S_2 \mid G \mid *G \mid \mathbf{prio}\ p\ (S) \mid S_1 \,\|\|\, S_2$
Guarded-Command	$G ::=$	$[]_{i=1}^{n} c_i? \rightarrow S_i \,[]\, \mathbf{delay}\, t \rightarrow S]$
Network	$N ::=$	$\ll S \gg \mid N_1 \| N_2$
Program	$P ::=$	$S \mid N$

4.1 Basic Timing Assumptions

To determine the real-time behaviour of programs, we have to make assumptions about the execution time of atomic statements and the overhead needed for compound constructs. For simplicity we make in this paper the following assumptions: **skip** does not take any execution time, priority assignments do not take any execution time, **atomic**(t) takes exactly t time units, and a process that has executed a **delay** t statement starts requesting processor-time for the next statement after exactly t time units. Assume given constants $K_e > 0$ and $K_c > 0$ such that the execution of io-statements and delay-statements requires K_e time units, the overhead for a guarded command is K_e, there is no overhead for other compound statements, and the actual communication (i.e., without waiting) takes K_c time units. Observe that this is indeed a simplification, since these assumptions do not correspond to our operational model from Section 3.2. For instance, the execution of a statement may have to be postponed to guarantee mutual exclusion. The semantics can be easily adapted for the case that the execution times are be given by lower and upper time bounds (see [Hoo91]).

Moreover, bounds must be given on how long a process is allowed to wait with the execution of a primitive statement when a processor is available, and with the execution of an io-statement when a communication partner is available. Based on the operational description from Section 3, we assume *maximal progress* which means that a process never waits unnecessarily; if execution can proceed it will do so immediately. There are two possible reasons for a process to wait:

- Wait to execute an io-statement because no communication partner is available. Since we assume that for each channel special link interfaces are available, we have maximal progress for communications. Hence two statements $c!$ and $c?$ are not allowed to wait simultaneously. As soon as both partners are available the communication must take place. Thus maximal progress implies *minimal waiting* for communications.
- Wait to execute an atomic statement because the processor is not available. The maximal progress constraint implies that if a processor is idle (that is, no statement is executing) then also no statement on that processor requests execution time. Hence we also have minimal waiting for processor-time.

5 Denotational Semantics

5.1 Computational Model

Our formal model of real-time communication behaviour consists of a mapping from points of time to sets of channel names, indicating the channels along which messages are being transmitted at that time, and directed channel names to record information about those processes waiting to send or waiting to receive messages on these channels. Using this information, the formalism enforces minimal waiting for communications by requiring that no pair of processes is ever simultaneously waiting to send and waiting to receive on a shared channel. In addition to this information, which is also present in maximal parallelism models, we now also record at each point of time whether or not a process is executing with a certain priority and, moreover, the priorities of processes that request execution time.

For simplicity, the time domain used in the semantics equals the domain *TIME* which is used in the syntax. In this paper we take a dense time domain; we assume that *TIME* equals the nonnegative rationals, i.e., $TIME = \{\tau \in \mathcal{Q} \mid \tau \geq 0\}$, where $\mathcal{Q}$ is the set of rational numbers. For notational convenience, a special value ∞, $\infty \notin (TIME \cup PRIO)$, is used with the usual properties. Further, assume that $0 \leq p$, for all $p \in PRIO$. For a set A, the powerset of A (the set of all subsets of A) is denoted by $\wp(A)$. A model of a real-time computation is defined as follows:

Definition 5.1 (Model) Let $\tau_0 \in TIME \cup \{\infty\}$. A *model* is a mapping
$\sigma : [0, \tau_0) \rightarrow \wp(DCHAN) \times \wp(PRIO) \times \wp(PRIO \cup \{\infty\})$.

Hence, for all $\tau \in TIME, \tau < \tau_0$, we have that $\sigma(\tau) = (comm, req, exec)$ with $comm \subseteq DCHAN$, $req \subseteq PRIO$ and $exec \subseteq PRIO \cup \{\infty\}$. Henceforth, we refer to the three fields of $\sigma(\tau)$ by $\sigma(\tau).comm$, $\sigma(\tau).req$, and $\sigma(\tau).exec$, respectively.

Definition 5.2 (Length of a Model) For a model σ with domain $[0, \tau_0)$ the *length* of σ, denoted $|\sigma|$, is defined by $|\sigma| = \tau_0$.

Informally, if $|\sigma| = \infty$ then σ represents a non-terminating computation, and if $|\sigma| < \infty$ it corresponds to an execution which terminates at time $|\sigma|$. For a point of time τ, with $\tau < |\sigma|$, the fields of $\sigma(\tau)$ have the following meaning in the semantics of program S:

- $c \in \sigma(\tau).comm$ if a communication takes place along channel c at time τ;
- $c! \in \sigma(\tau).comm$ if S is waiting to send along channel c at time τ;
- $c? \in \sigma(\tau).comm$ if S is waiting to receive along channel c at time τ.
- $\sigma(\tau).exec$ is either the empty set (when S is not executing at τ) or a singleton, containing the priority of the currently executing statement from S at time τ. At the start of the execution this will be the priority assigned to the statement, and during execution – when the execution can not be interrupted – we use priority ∞.
- $\sigma(\tau).req$ is the set of priorities from all statements in S which are requesting to be executed at time τ.

Definition 5.3 (Concatenation of Models) Define *concatenation* of two models σ_1 and σ_2, denoted $\sigma_1\sigma_2$, by $|\sigma_1\sigma_2| = |\sigma_1| + |\sigma_2|$ and

$$\sigma_1\sigma_2(\tau) = \begin{cases} \sigma_1(\tau) & \text{for all } \tau < |\sigma_1| \\ \sigma_2(\tau - |\sigma_1|) & \text{for all } |\sigma_1| \leq \tau < |\sigma_1| + |\sigma_2| \end{cases}$$

Definition 5.4 (Concatenation of Sets of Models) For two sets of models Σ_1 and Σ_2, we define the concatenation of these sets as follows:

$SEQ(\Sigma_1, \Sigma_2) = \{\sigma_1\sigma_2 \mid \sigma_1 \in \Sigma_1, \text{ and } \sigma_2 \in \Sigma_2\}$.

Since concatenation of models is associative, so is SEQ. We use $SEQ(\Sigma_1, \ldots, \Sigma_n)$ to denote $SEQ(\Sigma_1, SEQ(\Sigma_2, SEQ(\Sigma_3, \ldots, \Sigma_n)\cdots))$.

5.2 Formal Semantics

In this section we define the semantic function $\mathcal{M}$ which assigns to each statement a set of models. Since we give a compositional semantics, the meaning of a statement in isolation must characterize all potential computations of the statement. Thus the semantics represents the behaviour of a statement in any arbitrary environment. When composing this statement with (part of) its environment, the semantic operators must select the models corresponding to the computations that are still possible and remove all other models.

Skip A skip-statement terminates immediately and requires no execution time.

$$\mathcal{M}(\mathbf{skip}) = \{\sigma \mid |\sigma| = 0\}$$

Atomic To formulate the semantics of an $\mathbf{atomic}(t)$ statement, we observe that there are in general two periods (see Figure 1): first the statement is requesting processor-time (with default priority 0), and then the statement is executed for a certain period. At

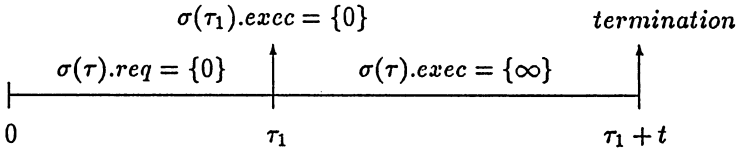


Figure 1: A model σ from of $\mathcal{M}(\mathbf{atomic}(t))$

the start of this execution period, it claims that it has the highest priority by placing its priority in the *exec*-field. When composing processes in parallel we require that there is no statement requesting processor-time with a higher priority at that point of time. Once the execution has been started, it cannot be interrupted—not even by a requesting statement with a higher priority. This is modelled by using the special priority value ∞, which is higher than any other priority, in the *exec*-field. Formally, using *Request* and *Execute*(t) defined below,

$$\mathcal{M}(\textbf{atomic}(t)) = SEQ(Request, Execute(t))$$

where $Request = \{\sigma \mid$ there exists a $\tau_1 \in TIME \cup \{\infty\}$, such that $|\sigma| = \tau_1$, for all $\tau < |\sigma|$: $\sigma(\tau).req = \{0\}$ and $\sigma(\tau).comm = \sigma(\tau).exec = \emptyset\}$

$Execute(t) = \{\sigma \mid \sigma(0).exec = \{0\}$, for all τ, $0 < \tau < t$: $\sigma(\tau).exec = \{\infty\}$, for all $\tau < t$: $\sigma(\tau).comm = \sigma(\tau).req = \emptyset$, and $|\sigma| = t\}$.

Note that models from *Request* need not terminate, and hence the statement is allowed to wait forever. At processor closure, however, we require that no statement is requesting to be executed when the processor is idle.

Delay A **delay** t statement first requests processor-time. If processor-time is available, this statement is executed during K_e time units. After this execution period there is a delay period of exactly t time units.

$$\mathcal{M}(\textbf{delay } t) = SEQ(Request, Execute(K_e), Delay(t))$$

with $Delay(t) = \{\sigma \mid$ for all $\tau < t$: $\sigma(\tau).comm = \sigma(\tau).req = \sigma(\tau).exec = \emptyset$, and $|\sigma| = t\}$.

Input and Output For an io-statement we also have the two periods mentioned above, i.e., first it requests processor-time, and then is executed during K_e time units. There are, however, two additional periods (see Figure 2): after its execution an io-statement starts to wait for a corresponding partner and when such a partner is available the communication takes place (during K_c time units).

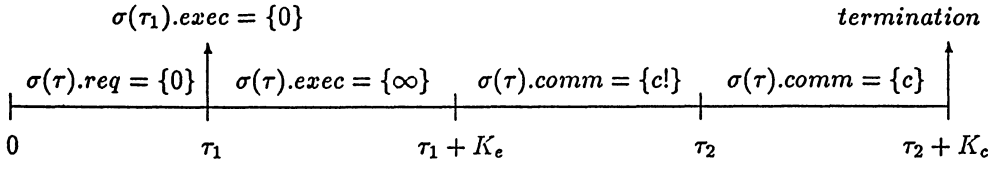


Figure 2: A model σ from $\mathcal{M}(c!)$

Using $Wait(c!)$ and $Comm(c)$ defined below, this leads to the following definition

$$\mathcal{M}(c!) = SEQ(Request, Execute(K_e), WaitSend(c), Comm(c))$$

where $WaitSend(c) = \{\sigma \mid$ there exists a $\tau_2 \in TIME \cup \{\infty\}$, such that $|\sigma| = \tau_2$, and for all $\tau < |\sigma|$: $\sigma(\tau_1).comm = \{c!\}$, $\sigma(\tau).req = \sigma(\tau).exec = \emptyset\}$

$Comm(c) = \{\sigma \mid$ for all $\tau < |\sigma| : \sigma(\tau).comm = \{c\}$, $\sigma(\tau).req = \sigma(\tau).exec = \emptyset$, and $|\sigma| = K_c\}$

Note that *Request* and $WaitSend(c)$ allow non-terminating models which corresponds to a process which is, respectively, continuously waiting to be executed and continuously waiting for a communication partner. Similarly, we define

$$\mathcal{M}(c?) = SEQ(Request, Execute(K_e), WaitRec(c), Comm(c))$$

where $WaitRec(c)$ is defined similar to $WaitSend(c)$.

Sequential Composition Using the SEQ operator, as defined above, we have

$$\mathcal{M}(S_1;S_2) = SEQ(\mathcal{M}(S_1), \mathcal{M}(S_2))$$

Guarded Command For a guarded command $G \equiv [[]_{i=1}^{n} c_i? \rightarrow S_i \,[]\, \mathbf{delay}\, t \rightarrow S]$ there are two possibilities after the usual requesting and executing periods:

- Either a communication along one of the c_i can be performed, represented by $Comm(G)$ below, before t time units have elapsed. Then this communication is preceded by a period shorter than t time units during which G is ready to communicate on all channels $c_1, \ldots, c_n$.
- Or there is a time-out, that is, no communication is possible within t time units. Then there is a waiting period of t time units, followed by the execution of S.

This leads to the following definition:

$$\begin{aligned} \mathcal{M}(G) &= SEQ(Request, Execute(K_e), LimitedWait(G), Comm(G)) \\ &\cup\ SEQ(Request, Execute(K_e), TimeOut(G), \mathcal{M}(S)) \end{aligned}$$

where $LimitedWait(G) = \{\sigma \mid \sigma \in Wait(G) \text{ and } |\sigma| < t\ \}$ and

$TimeOut(G) = \{\sigma \mid \sigma \in Wait(G) \text{ and } |\sigma| = t\ \}$ with

$Wait(G) = \{\sigma \mid$ there exists a $\tau \in TIME \cup \{\infty\}$ such that for all $\tau_1 < |\sigma|$:
$\sigma(\tau_1).comm = \{c_1?, \ldots, c_n?\}$, $\sigma(\tau_1).req = \sigma(\tau_1).exec = \emptyset$ and $|\sigma| = \tau\}$

$Comm(G) = \{\sigma \mid$ there exists a $k \in \{1, .., n\}$ such that $\sigma \in SEQ(Comm(c_k), \mathcal{M}(S_k))\ \}$.

Iteration Every computation of $*G$ either consists of a finite number of computations from G where the last one is non-terminating, or it consists of an infinite number of computations from G. Hence, the semantics of the iteration construct $*G$ is defined as

$$\begin{aligned} \mathcal{M}(*G) &= \{\sigma \mid \text{there exists a } k \in I\!N, k \geq 1 \text{ and models } \sigma_1, \ldots, \sigma_k \text{ such that} \\ &\qquad \sigma = \sigma_1 \cdots \sigma_k, \text{ with } \sigma_i \in \mathcal{M}(G), \text{ for } i \in \{1, .., k\}, \\ &\qquad |\sigma_i| < \infty, \text{ for } i \in \{1, .., k-1\}, \text{ and } |\sigma_k| = \infty\ \} \\ &\cup\ \{\sigma \mid \text{there exists an infinite sequence of models } \sigma_1, \sigma_2, \ldots \text{ such that} \\ &\qquad \sigma = \sigma_1\sigma_2 \cdots, \text{ with } \sigma_i \in \mathcal{M}(G) \text{ and } |\sigma_i| < \infty, \text{ for } i \geq 1\ \} \end{aligned}$$

Observe that $\mathcal{M}(*G) = \mathcal{M}(G;*G) = SEQ(\mathcal{M}(G), \mathcal{M}(*G))$, that is, $\mathcal{M}(*G)$ is a (non-empty) fixed point of the function $F(X) = SEQ(\mathcal{M}(G), X)$. Moreover, we can prove that $\mathcal{M}(*G)$ is the unique non-empty fixed point of F. The proof is based on the observation that $|\sigma| \geq K_e$ for all $\sigma \in \mathcal{M}(G)$. (By this property also an infinite loop in finite time is avoided.)

Priority Assignment Let $\sigma[p/0]$ be the model obtained from σ by substituting p for each occurrence of priority 0 in the *req*- and *exec*-fields of σ. Formally, $|\sigma[p/0]| = |\sigma|$ and for all $\tau < |\sigma|$, $\sigma[p/0](\tau).comm = \sigma(\tau).comm$,
$\sigma[p/0](\tau).req = \{p' \mid p' \in \sigma(\tau).req \wedge p' \neq 0\} \cup \{p \mid 0 \in \sigma(\tau).req\}$, and
$\sigma[p/0](\tau).exec = \{p' \mid p' \in \sigma(\tau).exec \wedge p' \neq 0\} \cup \{p \mid 0 \in \sigma(\tau).exec\}$. Then we have

$$\mathcal{M}(\text{prio } p\ (S)) = \{\sigma[p/0] \mid \sigma \in \mathcal{M}(S)\}$$

Parallel Composition For notational convenience, assume $\sigma(\tau).comm = \sigma(\tau).req = \sigma(\tau).exec = \emptyset$, for all $\tau \geq |\sigma|$. Then we can define the pointwise union of two models σ_1 and σ_2, denoted $\sigma_1 \cup \sigma_2$, as follows: $|\sigma_1 \cup \sigma_2| = max(|\sigma_1|, |\sigma_2|)$, and for all $\tau < |\sigma_1 \cup \sigma_2|$, $(\sigma_1 \cup \sigma_2)(\tau).comm = \sigma_1(\tau).comm \cup \sigma_2(\tau).comm$, $(\sigma_1 \cup \sigma_2)(\tau).req = \sigma_1(\tau).req \cup \sigma_2(\tau).req$, and $(\sigma_1 \cup \sigma_2)(\tau).exec = \sigma_1(\tau).exec \cup \sigma_2(\tau).exec$. The semantics of $P_1 \parallel P_2$ consists of all models σ such that there exists $\sigma_1 \in \mathcal{M}(P_1)$ and $\sigma_2 \in \mathcal{M}(P_2)$ with $\sigma = \sigma_1 \cup \sigma_2$. Note that this includes $|\sigma| = max(|\sigma_1|, |\sigma_2|)$, which corresponds to the notion that the parallel composition of two processes terminates when both processes have terminated. Furthermore we have the following requirements:

1. If that P_1 and P_2 are executed on a single processor, then at most one of the processes should execute at any point of time. This condition (and the next one) will be trivially fulfilled for networks (see the definition of processor closure below).
2. The priority of an executing statement should be greater than or equal to the priority of any executing statement.
3. Since we have synchronous communication, P_1 and P_2 should communicate simultaneously on shared channels, connecting the two processes. Note that these joint channels are represented by the set $dch(P_1) \cap dch(P_2)$.
4. In our execution model we assume minimal waiting for communications, that is, two processes should not be simultaneously waiting to send and waiting to receive on a shared channel.

Hence, the semantics of parallel composition is formulated as:

$$\begin{aligned}\mathcal{M}(P_1 \parallel P_2) = \{\sigma \mid\ & \text{there exist } \sigma_1 \in \mathcal{M}(P_1) \text{ and } \sigma_2 \in \mathcal{M}(P_2) \text{ such that } \sigma = \sigma_1 \cup \sigma_2, \\ & \text{for all } \tau < |\sigma| \text{ and for all } c \in dch(P_1) \cap dch(P_2): \\ & \quad \sigma_1(\tau).exec = \emptyset \vee \sigma_2(\tau).exec = \emptyset, \\ & \quad p_1 \in \sigma(\tau).exec \wedge p_2 \in \sigma(\tau).req \rightarrow p_1 \geq p_2, \\ & \quad c \in \sigma_1(\tau).comm \leftrightarrow c \in \sigma_2(\tau).comm, \text{ and} \\ & \quad \neg(c! \in \sigma(\tau).comm \wedge c? \in \sigma(\tau).comm)\ \}\end{aligned}$$

Parallel composition is commutative and associative.

Processor Closure In the semantics of processor closure we require that if the processor is idle at time τ, represented by $\sigma(\tau).exec = \emptyset$, then no statement is requesting, i.e., $\sigma(\tau).req = \emptyset$. On a model σ that satisfies this requirement we then apply

the network abstraction operator, $netw(\sigma)$, which removes all executing and requesting information from σ, since this should not be visible on the network level. Define $|netw(\sigma)| = |\sigma|$ and, for all $\tau < |\sigma|$, $(netw(\sigma))(\tau).exec = (netw(\sigma))(\tau).req = \emptyset$, and $(netw(\sigma))(\tau).comm = \sigma(\tau).comm$. Then we define

$$\mathcal{M}(\ll S \gg) = \{netw(\sigma) |\ \sigma \in \mathcal{M}(S), \text{ for all } \tau < |\sigma|\colon\ \sigma(\tau).exec = \emptyset \rightarrow \sigma(\tau).req = \emptyset\}$$

Note that the programs $\ll$ **prio** 1 $(c!) \,\|\, $ **prio** 5 $(d?) \gg$ and $\ll$ **prio** 2 $(c!) \,\|\, $ **prio** 3 $(d?) \gg$ obtain the same semantics.

6 Concluding Remarks

We have defined a denotational semantics to describe the real-time behaviour of an Occam-like language where several processes may share a single processor. The interleaving of atomic actions on one processor is restricted by dynamic priorities which are explicitly assigned to statements in the program. Communications are performed by separate link interfaces. Observe that the maximal parallelism assumption, where each process has its own processor, can now be expressed in our programming language. In general, given the design of a parallel program, we can express and compare different assignments of processes to processors.

Closely related to our work is the research done by Gerber and Lee. In a recent paper [GL90] they describe a resource based execution model with interleaving on a single resource and true concurrency among multiple resources. Their formal framework, however, is based on process algebraic techniques. Further, they have defined an operational semantics, although this semantics does not deal with priorities. Such a semantics could serve as a basis for comparison, since we also plan to define an operational semantics for our language and to prove that it is equivalent to our denotational semantics. Future research also includes the definition of a fully abstract semantics for our programming language and to compare this with fully abstract models for maximal parallelism. The main point is the influence of requesting and executing information on the abstractness of the semantics.

Based on the denotational semantics given in this paper, we have formulated in [Hoo91] two compositional proof systems for multiprogramming. In the first proof system, which is an extension of [HW89], one can prove that a program satisfies a specification written in a real-time version of temporal logic. The second proof system is based on extended Hoare triples (i.e., pre-condition, program, post-condition), using a first-order language with references to time (similar to [Hoo90]). In future work we intend to investigate the application of these formalisms to realistic examples.

Acknowledgements

Willem-Paul de Roever is gratefully acknowledged his stimulating interest in this work and many useful comments on the semantic model. Thanks also to Amir Pnueli for his illuminating remarks during the meetings in the context of Esprit projects DESCARTES and SPEC. Frank Zoontjens is thanked for his contribution to a preliminary version of the semantics. The referees are acknowledged for their accurate comments.

References

[FLP84] N. Francez, D. Lehman, and A. Pnueli. A linear history semantics for distributed programming. *Theoretical Computer Science*, 32:25–46, 1984.

[GB87] R. Gerth and A. Boucher. A timed failures model for extending communicating processes. In *Proceedings in the 14th International Colloquium on Automata, Languages and Programming*, pages 95–114. LNCS 267, Springer-Verlag, 1987.

[GL90] R. Gerber and I. Lee. CCSR: a calculus for communicating shared resources. In *CONCUR '90*, pages 263–277. LNCS 458, Springer-Verlag, 1990.

[HGR87] C. Huizing, R. Gerth, and W.P. de Roever. Full abstraction of a real-time denotational semantics for an OCCAM-like language. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 223–237, 1987.

[Hoo90] J. Hooman. Compositional verification of distributed real-time systems. In *Proceedings Workshop on Real-Time Systems - Theory and Applications*, pages 1–20. North-Holland, 1990.

[Hoo91] J. Hooman. *Specification and Compositional Verification of Real-Time Systems.* PhD thesis, Eindhoven University of Technology, 1991.

[HW89] J. Hooman and J. Widom. A temporal-logic based compositional proof system for real-time message passing. In *Parallel Architectures and Languages Europe*, volume II, pages 424–441. LNCS 366, Springer-Verlag, 1989.

[KSR+88] R. Koymans, R.K. Shyamasundar, W.P. de Roever, R. Gerth, and S. Arun-Kumar. Compositional semantics for real-time distributed computing. *Information and Computation*, 79(3):210–256, 1988.

[Occ88a] INMOS Limited. *Communicating process architecture*, 1988.

[Occ88b] INMOS Limited. OCCAM *2 Reference Manual*, 1988.

[RR87] G. Reed and A. Roscoe. Metric spaces as models for real-time concurrency. In *Proceedings Workshop on the Mathematical Foundations of Programming Languages Semantics*, pages 331–343. LNCS 298, Springer-Verlag, 1987.

[Zwi89] J. Zwiers. *Compositionality, Concurrency and Partial Correctness.* LNCS 321, Springer-Verlag, 1989.

CONCURRENT CLEAN

Nöcker E.G.J.M.H., Smetsers J.E.W., Eekelen M.C.J.D. van, Plasmeijer M.J.

Faculty of Mathematics and Computer Science,
University of Nijmegen,
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
E-mail: clean@cs.kun.nl

Abstract

Concurrent Clean is an experimental, lazy, higher-order parallel functional programming language based on term graph rewriting. An important difference with other languages is that in Clean graphs are manipulated and not terms. This can be used by the programmer to control communication and sharing of computation. Cyclic structures can be defined. Concurrent Clean furthermore allows to control the (parallel) order of evaluation to make efficient evaluation possible. With help of sequential annotations the default lazy evaluation can be locally changed into eager evaluation. The language enables the definition of partially strict data structures which make a whole new class of algorithms feasible in a functional language. A powerful and fast strictness analyser is incorporated in the system. The quality of the code generated by the Clean compiler has been greatly improved such that it is one of the best code generators for a lazy functional language. Two very powerful parallel annotations enable the programmer to define concurrent functional programs with arbitrary process topologies. Concurrent Clean is set up in such a way that the efficiency achieved for the sequential case can largely be maintained for a parallel implementation on loosely coupled parallel machine architectures.

1.0 Introduction

Historical context

Concurrent Clean (Eekelen et al. (1990)) is an experimental, lazy, higher-order functional programming language based on term graph rewriting (Barendregt et al. (1987a)). The first work on Clean started in 1984 in the Dutch Parallel Reduction Machine project (Barendregt et al. (1987), Brus et al. (1987)) in which the feasibility of the realization of a parallel reduction machine was investigated. The Nijmegen research focussed on the fundamentals of graph reduction and its implementation on sequential and parallel architectures. The fundamental idea is that graph reduction should not be considered as merely an optimisation in the implementation of functional languages, but that graph reduction is a fundamental basis for any implementation and that graph reduction itself must be investigated and optimised. In this context together with the University of East-Anglia a more general non-functional computational model, Generalized Graph Rewriting Systems (GGRS's) has been designed (Barendregt et al. (1987b)) of which the semantics and pragmatics currently are further investigated in the Esprit Basic Research Action "Semagraph". The Dactl-language used in the declarative UK-Flagship projects is based on GGRS's (Glauert et al. (1987)) as well as the jointly with the University of East-Anglia (UEA) defined language Lean (Barendregt et al. (1987b,1988)). Based on restricted GGRS's the functional graph rewriting language Clean (Brus et al. (1987)) was developed as an intermediate language for the compilation of functional languages. Implementations (compilers and interpreters) of Clean (Brus et al. (1987), Nöcker (1989), Smetsers (1989)) have been developed as well as a Miranda-to-Clean conversion program (Koopman & Nöcker (1988)). Concurrent Clean is partly developed as a part of the Esprit TIP-M Tropics project (Eekelen et al. (1989)).

The language Concurrent Clean

In this paper the language Concurrent Clean is presented that extends the sequential language Clean to a concurrent language suited for efficient code generation for both sequential and parallel machine architectures. Concurrent Clean has many features in common with other lazy, higher-order functional

languages, such as a Milner/Mycroft based polymorphic type system (including algebraic types, synonym types and abstract types). A key aspect of the language is that the object that is manipulated is a graph and not a term. Consequently, the programmer can explicitly control sharing of computation. For instance, cyclic data structures can be created. The most important aspect of Concurrent Clean discussed in this paper is the way in which the order of evaluation can be controlled. Lazy evaluation can be locally changed in eager evaluation. Eager evaluation has the advantage that in general it can be implemented considerably more efficiently than lazy evaluation. Even more speed-up can be achieved by changing sequential evaluation into parallel evaluation.

Changing lazy into eager evaluation

An important feature of the Concurrent Clean system is that strictness annotations are generated automatically by a strictness analyser. This analyser has been designed and implemented based on the concept of abstract set reduction (Nöcker (1990)). The strictness analyser is an efficient as well as powerful analyser that can deal with arbitrary data structures and higher-order functions. To change the default lazy reduction order into eager also the programmer can put strictness annotations in the function definition themselves or in their type definition (Smetsers (1989)). Furthermore, considerable efficiency improvements can be realized by defining a special kind of data types: partially strict data types (Nöcker & Smetsers (1990)) that enable composite data structures to be handled on the stack completely without any usage of the heap.

Changing sequential into parallel evaluation

In Concurrent Clean the programmer can control the parallel evaluation of the functional program with help of only two annotations (Eekelen et al. (1991)). The annotations enable the programmer to allocate processes to parts of the graph in such a way that arbitrary, possibly cyclic, process topologies can be specified.
With the same two annotations the programmer can specify that, when communication takes place, a value has to be communicated or that the expression to compute the value has to be shipped. Communication between processes takes place implicitly on demand via the concept of lazy copying (Eekelen et al. (1991)).
Concurrent Clean is designed for the evaluation on loosely coupled parallel machine architectures. As a special case multi-processing on a single processor can be expressed. Complicated parallel algorithms which can go far beyond divide-and-conquer like applications can be specified. The design of Concurrent Clean is such that the sequential optimisations mentioned above can still be applied in the parallel case. A local reservation/locking mechanism is required that introduces a neglectable overhead.

In this paper an overview is given of the main features of the language Concurrent Clean (Section 2). In more detail it is explained how the (parallel) reduction order is controlled (Section 3). The sequential (Section 4) and parallel (Section 5) implementation of Concurrent Clean is treated. Performance figures are given in Section 6.

2.0 Overview of the Language

In this section we briefly introduce the flavour of Concurrent Clean by showing how some well-known functional programs are written down in this formalism. The first example shows how the factorial function can be specified in Clean:

```
MODULE Fac;

IMPORT delta;

RULE
::   Fac INT   ->   INT                  ;
     Fac 0     ->   1                    |
     Fac n     ->   *I n (Fac (--I n))   ;

::   Start     ->   INT                  ;
     Start     ->   Fac 20               ;
```

A Clean program is composed of modules. Modules are hierarchical. The top-most module is the main module. In the main module a `Start` rule should be declared of which the left-hand-side consists of the symbol `Start` and the right-hand-side corresponds with the initial expression to be computed.

With the `IMPORT` statement all predefined functions (delta rules) and predefined types are imported. `--I` (integer decrement) and `*I` (integer multiplication) are such predefined functions defined on the basic type `INT`.

Rules starting with `::` are either new type definitions or type specifications of rewrite rules. In the latter case the type of the corresponding function is specified. In a Clean program all the rules for a certain function are called the alternatives for that function. It is required that all the alternatives are grouped together. The reader will have inferred that the rule alternatives of a function definition have a priority: they are applied in textual order.

```
MODULE Map;

||   Example of how to use higher order
||   functions in Concurrent Clean

FROM deltaI IMPORT *I;

RULE
::   Square INT     -> INT                    ;
     Square x       -> *I x x                 ;

::   Map (=> x y) [x] -> [y]                  ;
     Map f []         -> []                   |
     Map f [a|b]      -> [f a|Map f b]        ;

::   Start  -> [INT]                          ;
     Start  -> Map Square [42,43,44]          ;
```

In Clean comments can be specified via preceding the comment with `||`. This has to be done on every line in which a comment is given. Square brackets are used for denoting lists: `[]` is an empty list, `[a,b,c]` a list containing the three elements `a`, `b` and `c` and `[a | f]` denotes a list consisting of a list `f` prefixed with an element `a`.

The example also shows that higher order functions can be used freely. There is no difference between the use of full and partial (curried) applications of functions. Types of higher order functions are specified using => (prefix notation) which corresponds to -> (infix notation) in languages like Miranda[1].

The following example is a solution for the Hamming problem: it computes an ordered list of all numbers of the form 2^n3^m, with $n,m \geq 0$. Note that with the explicit nodeid `x`, defined in the right hand side, a cyclic graph is created that allows the use of computations already performed.

```
MODULE Ham;

FROM deltaI    IMPORT *I                     ;
FROM Map       IMPORT Map                    ;
FROM Merge     IMPORT Merge                  ;

RULE
::   Ham  -> [INT]                                          ;
     Ham  -> x:[1 | Merge (Map (*I 2) x) (Map (*I 3) x)]    ;
```

2.1 Type System

Concurrent Clean is a strongly typed language. It is, however, not required to declare the types of functions explicitly: types are deduced by the compiler from the information in the program. The (polymorphic) type scheme that is used for this purpose is based on the a combination of the well-known Milner (1978) and Mycroft (1984) schemes.

---

[1] Miranda™ is a trademark of Research Software Ltd.

The predefined types in Concurrent Clean and examples of denotations and predefined functions are listed below:

Basic types:	INT, REAL, BOOL,CHAR,STRING, FILE
Examples of denotations:	2, 0.31415E1,TRUE,'a',"monkey"
Predefined functions:	+I,<R,NOT,=C,SLICE,FOpen
List and tuple types:	[**T**], ($\mathbf{T}_1$, ... ,$\mathbf{T}_n$) for types **T** and $\mathbf{T}_i$
Denotations for lists and tuples:	[1,2,3,4], [], [2|[]], (1,'?',FALSE)

Defining New Types

There are three mechanisms to introduce new types: *algebraic* type definitions, *synonym* type definitions and *abstract* type definitions.

Synonym types allow the user to define a new name for an already existing type. These types are specified by means of a type rule having exactly one alternative of which the right-hand-side is a type instance.

A *type instance* is either a type variable or an acyclic graph that has a root symbol that is a *type symbol* of which all the arguments are type instances. A type symbol is either a basic type symbol or a user-defined type symbol.

An example of a synonym type definition:

```
TYPE
::   Stack x      ->   [x]        ;
```

With the aid of algebraic types it is possible to introduce a new concrete data type based on free algebras. These types are specified by means of a type rule whereof each alternative has a right-hand-side with a unique root symbol: the *constructor*. The constructor is said to be of that specific *type*. All the arguments of the constructor are type instances.

Example of algebraic type definitions . The types `Nat` and `List` are being defined. The constructors `Zero` and `Succ` are said to be of type `Nat`, `Cons` and `Nil` of type `List x`.

```
TYPE
::   Nat        ->   Zero                  |
     Nat        ->   Succ Nat              ;

::   List x     ->   Cons x (List x)       |
     List x     ->   Nil                   ;
```

Abstract types offer the possibility of hiding the representation of a certain type. To distinguish an abstract type definition from an ordinary type definition a special kind of type block is provided called an `ABSTYPE`-block.

Example of abstract type definition in Clean:

```
ABSTYPE
::   Stack x   ;
```

Abstract type definitions are only allowed in definition modules (see the section on modules). In the implementation module the abstract type should either be a synonym type or an algebraic type. The realisation of the type is invisible for the outside world.

Typing Functions

Each rewrite rule can be typed explicitly by the programmer. This type specification must immediately precede the corresponding rewrite rule.

When typing partial functions one has to ensure that the function symbol itself can be used as a constructor by giving an appropriate algebraic type definition for it. An error is generated at run-time if this has not been

indicated properly (note that in general it cannot be detected at compile-time whether a function is partial). First an example that leads to a run-time type error:

```
RULE
::   F INT  ->   INT  ;
     F 0    ->   0    ;

::   Start  ->   INT  ;
     Start  ->   F 1  ;
```

Although the Clean program is correctly typed, the function `F` applied in the start-rule cannot be matched and therefore `F 1` will not yield the required type: `INT`. At run-time, an error is generated.

The second example shows how partial functions should be typed in order to avoid run-time errors:

```
TYPE
::   Num  ->   Zero        |
     Num  ->   Succ Num    |
     Num  ->   Pred Num    ;

RULE
::   Succ Num       ->   Num  ;
     Succ (Pred n)  ->   n    ;

::   Pred Num       ->   Num  ;
     Pred (Succ n)  ->   n    ;

::   Start          ->   Num                  ;
     Start          ->   Succ (Succ Zero)     ;
```

The graph `Succ (Succ Zero)` in the start rule will not match any rule. Still it is correct because the graph is indeed of the wanted type (`Num`). Notice that `Succ` and `Pred` are used both as functions as well as constructors. As constructors they may appear in the right-hand-side of type definitions and are of type `Num`. As functions they also yield type `Num`.

2.2 Modules

A Concurrent Clean program may be split in several modules that can be compiled separately. A Concurrent Clean program consists of *definition modules* and *implementation modules*. An implementation module consists of a set of type and rule definitions that can be *exported* to other modules via its definition module. The latter consists only of a set of type rules, possibly including strictness information, for exported types and for exported functions. Special definition modules, which are called *system modules*, indicate that the corresponding implementation module does not contain ordinary rewrite rules but (abstract) machine code instead. On demand the compiler will substitute the code of a function *in-line* at the place where this function is called.

2.3 Input and Output

To achieve an efficient implementation of IO facilities in Concurrent Clean the type FILE has been predefined. Besides that, a number of basic operations on files can be imported from a predefined module called deltaIO. This module contains functions to create files, to read characters or strings from files, to write characters or strings to files and to re-open write-files for reading.

The efficiency of the IO functions is obtained by implementing `FILE`'s not as (lazy) lists of characters but by using strict tuples. This allows the Concurrent Clean compiler to generate code for these IO functions wherein a fast call by value like mechanism of parameter passing and returning results is used.

3.0 Controlling Reduction Order

3.1 Graph Rewriting

A Clean program basically consists of a number of graph rewriting rules which specify how a (program) graph has to be rewritten. The program graph, which initially consists of a single `Start` node, is rewritten

according to these rules. The part of the graph that matches the pattern of a certain rewrite rule is called a *redex*. A *rewrite* of a redex consists of replacing the redex in the graph by an instance of the right-hand-side of the corresponding rewrite rule

3.2 Reduction Strategies

A reduction strategy repeatedly determines which redex is going to be reduced next. The strategy of Concurrent Clean, the so-called *functional strategy*. Reducing graphs according to this strategy resembles very much the way execution proceeds in most other lazy functional languages: if there are several rewrite rules for a particular function the rules are tried in textual order; patterns are tested from left to right; evaluation of arguments is forced when it is tried to match an actual argument against a non-variable part in the pattern.

In Concurrent Clean the functional strategy may be locally influenced by the use of annotations. When this strategy encounters an annotation it changes its default reduction order which will influence the way in which a result is achieved. Changing the order is in particular important if one wants to optimise the time and space behaviour of the reduction process.

Currently, two kinds of annotations are possible:
- *strict annotations* to locally change lazy evaluation into *eager evaluation*;
- *process annotations* to define *interleaved evaluation* on the same or *parallel evaluation* on another processor.

3.3 Sequential Annotations

The sequential flow of control can be influenced by means of strict annotations. If a strict annotation is encountered, the evaluation of the indicated subgraph is forced. This forced evaluation will also follow the functional strategy yielding a root normal form. After the forced evaluation has delivered the root normal form, the reduction process continues with the ordinary reduction order following the functional strategy. So, annotations let the reduction strategy deviate from the default functional evaluation order making the evaluation order partially eager instead of lazy.

We distinguish two kinds of strict annotation, namely, *global* and *local* strict annotations

Global Strict Annotations

The strict annotations in a type specification are called global because they change the reduction order for *all* applications of a particular function. Annotations in a type specification of a certain function are allowed to be placed before the type specification of either an argument on the left-hand-side or an argument of a tuple type appearing in a *strict context*. A tuple type is in a strict context if it has been supplied with a (valid) strict annotation itself or if it appears as the root node on the right-hand-side of the type rule. Intuitively, such a strict annotation indicates that the corresponding argument is always reduced to root normal form before the corresponding rule is applied.

Example of global strict annotation in type rules:

```
::   IF !BOOL  x    x     ->  x       ;
     IF  TRUE  then else  ->  then    |
     IF  FALSE then else  ->  else    ;
```

Strict annotations may also be used in tuple types appearing in a type synonym definitions The meaning of these annotated synonym types can be explained with the aid of a simple program transformation with which all occurrences of these synonym types are replaced by their right-hand-sides (of course, annotations included). These annotated type definitions are a special case of the more general *partially strict data types* which are treated later on in this section.

Local Strict Annotations

Strict annotations in rewrite rules are called *local*. They change only the order of evaluation for a *specific* function application. These annotations appear in the right-hand-side of rewrite rules.

Before the evaluation continues after applying a rewrite rule all strict annotated nodes of the right-hand-side of the applied rewrite rule are evaluated. Strict annotations in rewrite rules can be placed anywhere on the right-hand-side.

Example of strict annotations on the right-hand-side:

```
F x y -> IF x !y (! ++I y)          ;
```

In this particular application of `IF` it is clear that a common part of the then part and else part can safely be reduced.

Partially Strict Data Types

Partially strict data types are obtained by supplying the type definitions or type specifications of functions with additional (global) strictness information. In a type definition this strictness information specifies for each individual part of an instance of such a type whether this part should be evaluated or not (the so called evaluation context of that part). In a type specification of a function the strictness information determines the evaluation contexts of both the parameters and the result. The only partially strict data types that have been implemented in Concurrent Clean are the partially strict tuples (these types were already mentioned in the section on global strict annotations). An example of the use partially strict tuples is the following definition of a complex number:

```
TYPE
::   Complex     -> (!REAL,!REAL)                              ;

RULE
::   +C !Complex !Complex ->   Complex                        ;
     +C (r1,i1) (r2,i2)   ->   (+R r1 r2,+R i1 i2)            ;
```

3.4 Parallel Annotations

The parallel flow of control can be influenced by means of process annotations. Currently only *local process annotations* can be specified in the right-hand-side of rewrite rules.

If a process annotation is encountered, the evaluation of the indicated subgraph is forced as with a strict annotation, following the functional strategy until a root normal form is reached. The important difference with strict annotations is that with process annotations new reduction processes are created that perform the evaluation. These new reduction processes can run interleaved or in parallel with the original reduction process. The original process continues with the evaluation in the ordinary reduction order independently.

Creating parallel processes

The `{P}` annotation (`P` for parallel) creates a new graph, which is a copy of the annotated subgraph, on a remote processor together with a parallel reduction process (a *reducer*) which reduces this new graph to root normal form.

Creating interleaved processes

The `{I}` annotation (`I` for interleaved and internal) creates a new internal process on the annotated subgraph. This new internal reducer reduces the corresponding subgraph interleaved with the other processes of this processor (so no copy is made).

Communication Channels

Communication takes place when the initial graph that is going to be reduced in parallel has to be sent to another processor or when the result of such a parallel reduction is needed by another reducer.

Communication involves the copying of graphs. In Concurrent Clean the concept of *lazy-copying* is used (Smetsers et al. (1991)). When during the copying a subgraph is encountered that is already being reduced by another reduction process this subgraph is not copied at that moment. The copying is deferred until the other reducer has finished the reduction of this graph. The fact that the copying was stopped temporarily is administered with the aid of a special arc, a so-called (*communication*) *channel*, that interconnects the new copy with the subgraph that is currently reduced. The continuation of the copying is triggered when the result of the graph to which a channel refers is needed.

Besides creating channels implicitly via copying there is another way whereby channels come into existence: the initial subgraph of a new parallel reduction process is also connected to the original graph via a channel.

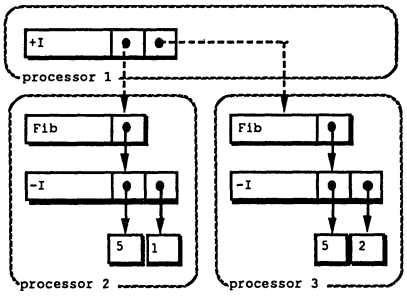
Note that the above-mentioned method of process creation and communication implies that the only interconnections between graphs residing on different processors are channels.

Divide and Conquer Parallelism

In the following example it is shown how a divide-and-conquer parallelism can be specified in Concurrent Clean:

```
Fib 0  ->  1                              |
Fib 1  ->  1                              |
Fib n  ->  +I left right,
           left:  {P} Fib (-I n 1),
           right: {P} Fib (-I n 2)        ;
```

The `{P}` annotations specify that both calls of `Fib` can be evaluated in parallel. The root of the graph on which a process is started, is built on another processing element with copies of subgraphs as arguments. The father reducer is waiting for the results. A copy of a result is made when a subgraph `left` or `right` is in root normal form. The picture below illustrates a possible processor structure after one reduction of `Fib 5`:



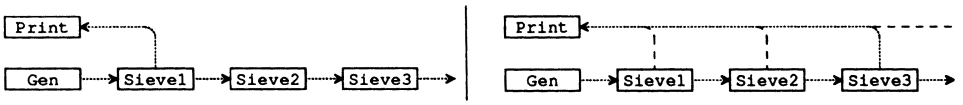
Parallel sieving

The sieve of Eratosthenes is a classical example which generates all prime numbers. A pipeline of `Sieve` processes is created. Those `Sieves` hold the prime numbers in ascending order, one in each `Sieve`. Each `Sieve` accepts a stream of integers as its input. Those integers are not divisible by any of the foregoing primes in the pipeline. If an incoming integer is not divisible by the local prime as well, it is send to the next `Sieve`. A newly created `Sieve` accepts the first incoming integer as its own prime and outputs this prime and the channel of the next `Sieve` to a printing process. After that it starts sieving. A process called `Gen` sends a stream of integers greater than one to the first `Sieve`.

The combination of process annotations and communication via copying provide that the intended behaviour is achieved. Processes are connected to each other by channels through which data is communicated in a demand driven way.

This can be represented in a picture as below (all arrows indicate flow of data on channels). `Sieve1` holds 2 as its own prime, `Sieve2` holds 3, `Sieve3` holds 5, and so on. The printing process one by one receives the

channel identifications from these sieves and collects the corresponding primes. Seen through the time this can be illustrated as follows (all arrows indicate flow of data on channels):



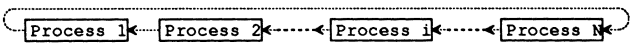
The Sieve:

```
Start                       ->   Print s,
                                 s: {P} Sieve g,
                                 g: {P} Gen 2                    ;

Sieve [pr | stream]         ->   [pr | s],
                                 s: {P} Sieve f,
                                 f: {I} Filter stream pr         ;

Gen n                       ->   [n | rest],
                                 rest: {I} Gen {!} (++I n)       ;

Filter [f | r] pr           ->   IF (=I (MOD f pr) 0)
                                    (Filter r pr)
                                    (NewFilter f r pr)           ;

NewFilter f r pr            ->   [f | rest],
                                 rest: {I} Filter r pr           ;
```

Arbitrary Process Structures

It is beyond the scope of this paper to treat the expressive power of Concurrent Clean very extensively. At this point we only want to claim that it is possible to specify any arbitrary process structure in a Concurrent Clean program. To illustrate this we give an example that shows how a cyclic process structure, i.e. a number of parallel reducers that are mutual dependent, can be created. It is extracted from quite a large program that implements Warshall's solution for the shortest path problem (van Eekelen et al. (1988)).

First the intended reducer topology is given in a picture:



This reducer structure can be specified directly in the following way:

```
Start                  ->   last:CreateProcs NrOfProcs last      ;

CreateProcs 1 left     ->   Process 1 left                       |
CreateProcs pid left   ->   CreateProcs (--I pid) new,
                            new: {P} Process pid left            ;
```

`CreateProcs` is responsible for the generation of all the parallel reducers. This process, which will finally become the first reducer, has initially a reference to itself in order to make it possible to expand it to a cycle of reducers. Each reducer is connected to the next one, i.e. the one with the next `pid` number, by means of a channel. During the creation of the processes this channel is passed as a parameter called `left`.

4.0 Sequential Implementation

Both sequential and parallel implementations of Concurrent Clean are based on the abstract ABC machine. A Concurrent Clean program is compiled to code for this abstract machine. In this way the Concurrent Clean compilation is largely machine independent. Testing the implementations and reasoning about them becomes much easier.

There are two ways in which this code can be executed. First, the ABC code can be interpreted. Second, it can be compiled to code for some concrete machine. The abstract machine can be implemented on various machines relatively easy.

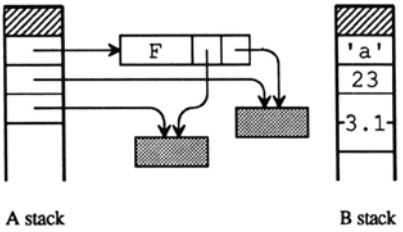
In this section we will outline the basic aspects of the ABC machine. The ABC machine resembles advanced G-machine like architectures (Johnsson (1987), Peyton Jones & Salkild (1989)). The Concurrent Clean compiler exploits all possibilities of the machine. This is discussed in section 4.2. Lastly, we treat how the ABC machine can be implemented on a real machine. More detailed information on these aspects can be found in (Smetsers (1989), Koopman et al. (1990), Groningen (1990))

4.1 The abstract ABC machine

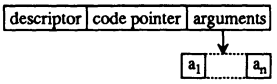
As mentioned before, the abstract ABC machine is similar to G-machine like architectures: it is a stack based graph reduction machine. The main parts of interest are the three stacks (Address, Basic value and Control stack) and the heap.

The C stack is used for storing addresses. The other two stacks are used for evaluating or building expressions, and for passing arguments to functions or returning results from functions. The A stack contains references to nodes in the heap, whereas the B stack contains values of basic types, such as integers or reals. Thus, basic values can be represented in two ways: as node in the heap or as an item on the B stack. Note that a B stack item can occupy more entries, for example, a Real value needs two entries.

Example:



Graphs are stored in the heap. So, the heap contains a collection of nodes. Generally speaking, a node of a Clean graph consists of a symbol with a certain number of arguments. Representing nodes as variable sized object causes problems with updating: the new node doesn't need to fit in the old one. This can be solved by introducing indirection nodes, but this will slow down the access on the contents of a node. In the ABC machine we have chosen to split a node in a fixed and a variable sized part. The fixed size part contains a representation of the symbol (called the *descriptor*), a code pointer and a pointer to a variable sized part.



The descriptor is a representation of a Clean symbol. Normally it is an index or pointer in a descriptor table. The descriptor is used for pattern matching and for printing.
The code pointer points to code with which the node has to be evaluated. During reduction this code pointer can be changed. For example, after entering the node for evaluation a pointer to code that generates an error message can be stored. If the node is entered again (indicating a non-terminating reduction) this code will be executed. If a node is updated with a head normal form value, the code pointer points to special code just containing a return statement.
In the variable sized part the arguments of the node are stored. This means that the arguments always have to be fetched via an extra indirection. On the other hand, updating a node is simple: update the fixed part, and allocate space for the arguments.

Except nodes containing a Clean symbol with the right number of arguments also other kinds of nodes are possible. For such nodes special things are done.

For nodes containing a basic value, e.g. an integer, the descriptor doesn't represent the Clean symbol (that would be the integer value itself!). Instead, all integers share the same descriptor (e.g. INT). The integer value itself is stored in the pointer part. For basic values which are too big (e.g. strings) a pointer to the value (for which space has to be allocated) is stored. As basic nodes are always in normal form, they all contain the head normal form code pointer.
In Concurrent Clean symbols can be applied on too few arguments. Such a partial application can be represented as a spine of applications. In practice, a better way is to build partial nodes, i.e. nodes with a partially filled argument part. Such nodes are built as standard nodes, but with special descriptors. So, for each Clean symbol of arity n, n+1 descriptors are defined. Mostly, the ABC machine sees no difference between such partial nodes and standard nodes. However, if a partial node is applied to another node, a new node with a new number of arguments has to be created.

4.2 The Concurrent Clean compiler

Syntactically, Concurrent Clean is quite an easy language. Therefore, the main task of the Concurrent Clean compiler is to generate efficient code. No complex transformations like lambda lifting, converting ZF-expressions etcetera are necessary.
Many standard optimisation techniques are implemented: tail recursion removal, unnecessary evaluation calls etcetera. In the following we will emphasize only those parts of the compiler that differ from other well-known implementations.

Conceptually, graph reduction is done in the heap: if a node has to be rewritten a new graph is built which will replace the original node. Unfortunately, this scheme will not give efficient code. The goal of the compiler is to generate code in which graph building is omitted as much as possible. For generating such efficient code type and strictness information is necessary. Type information can be fully derived by the type inference mechanism. Strictness information can be given by the programmer, or can be derived by a strictness analyser.
In general, deriving strictness information is very difficult. However, some help from the programmer normally will lead to more information. Certainly annotating data types can lead to much more efficient code.

Strictness Analysis

The strictness analyser in the Concurrent Clean compiler is based on *abstract reduction* (Nöcker (1990)). In abstract reduction a domain of sets of values is defined. Reduction in this domain means reduction of sets. Because this domain of sets is not finite fixed points techniques are not applicable. Problems due to recursive functions are solved with a technique called *reduction path analysis*. With this method also other kinds of strictness can be derived, for example, strictness properties for functions over lists. (however, such information is not used by the Concurrent Clean compiler). It appears that this analyser can find much information. The analysis itself is quite fast. Consider the functions:

```
Append [] y         ->   y                          |
Append [a|r] y      ->   [a | Append r y]           ;

Foldr op r []       ->   r                          |
Foldr op r [a|x]    ->   op a (Foldr op r x)        ;

Catenate l          ->   Foldr Append [] l          ;
```

With strictness analysis based on abstract interpretation for the function `Catenate` a fixed point in a rather complex domain has to be determined. With abstract reduction the right information is found quite easily (see Nöcker (1990) for the analysis).

Nodes in a strict context

There are two ways in which the compiler uses strictness information. First, nodes in a strict context normally don't need to be built. Instead, a call to the code belonging to the function is generated:

```
F x -> +I a b
       a: IF cond 3 b,
       b: IF cond a 4,
       cond: P x       ;
```

As can be seen easily, the node `cond` is in a strict context. In this case a direct call to P can be generated. However, despite the fact that nodes `a` and `b` are in a strict context, nodes for them have to be build because they are on a cycle (in fact, they are in a 'semi-strict' context).

Passing parameters and returning results

The second way in which strictness and type information is used is in passing values as parameters or as results. Values are passed via the A and B stack. The type of the function determines how this is done:

```
:: F INT !(!INT,![CHAR]) -> (INT,!CHAR);
```

The function F with this type is a function which needs two arguments. The first one, is a non-strict integer. This value is passed via the A stack. The second argument is a strict tuple. Both elements of this tuple have to be reduced to head normal form before calling F. The integer has to be passed via the B stack, whereas the character list is passed via the A stack. For the result value similar things have to be done: a (strict) tuple is returned of which the first element, a on-strict integer, will be passed via the A stack, and the second, a strict character, will be returned via the B stack.
If a value is not in the state in which it is needed for a function call a conversion has to be done. In the case of tuples, such a conversion can be quite complex. Such a conversion we will call a *coercion.*

Entry points

The above calling convention is applicable only if nodes appear in a strict context. However, there are three other ways in which a function can be called.
Firstly, a function application might have been appeared in a non-strict context. In this case a node has been built. If this node has to be evaluated first a conversion (in fact a coercion) has to be done before the strict code can be executed: arguments have to be fetched from the heap. If necessary, they have to be evaluated or, in the case of strict tuples, unpacked.
The second way in which a function can be called is if a partial application has been built. If, after that some applications have delivered the remaining arguments, a similar transformation has to be done.
Lastly, also special things have to be done for exported functions. The exported type determines the calling convention outside the module. However, inside the module another calling convention can be more efficiently. This is the case if abstract types are exported (hiding the internal representation), or if the strictness analyser finds more information than is exported. For both cases an additional entry point is needed. This 'external_strict' entry does some conversions according to the extra strictness and continues with the internal strict entry.
So, in general the layout of the code of a function is as follows:

```
apply_entry:
     get arguments
     jump convert_code
lazy_entry:
     get arguments
convert_code:
     convert strict args
     jump_subroutine to strict_entry
     update node
     return
external_strict_entry:
     convert strict args
strict_entry:
   ...
```

For some functions (e.g. many predefined functions like +I etc). special things are done. A call (in a strict context) to an addition would be unnecessary expensive. Instead, the addition code itself will be substituted

directly. This is done by '.inline' directives that are inserted in the strict code part. Such inline code is only searched for in the case of SYSTEM modules. Note that the compiler itself knows nothing about such functions. In this way new basic functions can be added easily. Even functions for which complex code has to be inserted can be expanded inline in this way.

4.3 Realisation on a concrete machine

Basic Aspects

There are two ways of implementing the ABC machine on sequential hardware: by means of an ABC code interpreter and by means of a code generator that compiles ABC-code into target machine code. The section gives a short description of the code generator for the MOTOROLA 680x0 processors. The interpreter is treated in the section on the current status of our research.

Code generation for an M68k processor

A straightforward way of generating concrete machine code is by means of macro expansion: each ABC instruction is considered as a macro application that is substituted by a sequence of M68k instructions. However, the quality of the generated M68k code is mainly determined by the way the registers of this processor are utilised. Since the ABC machine does not contain abstract registers it will be evident that the resulting code is far from optimal. Therefore the current ABC to M68k code generator uses a more intelligent way of generating code than just performing macro expansion. An ABC program is subdivided into *basic blocks* (i.e. sequences of ABC instructions that do not contain any label definitions or jump instructions). The code generator considers each basic block as a specification of how the initial state of the ABC machine (which is determined by the contents of the stacks and the graph store) at the start of the basic block has be converted into the final state at the end of the block. Now the tasks of the code generator becomes to implement such state transitions as efficient as possible, in all likelihood, by using registers. Note that, in contrast with the macro expansion mechanism, the relation between original ABC code and generated M68k code may be difficult to detect.

Besides using registers for computing intermediate results inside the basic blocks registers are also used for parameter passing and returning results between basic blocks.

As an example we give both the ABC code and M68k code generated for the factorial function that has been defined earlier:

ABC-code:

```
Fac.1:
     eqI_b +0 0
     jmp_true lab
     jmp sFac.2
lab:
     pop_b 1
     pushI +1
     rtn
Fac.2:
     push_b 0
     decI
     jsr Fac.1
     push_b 1
     update_b 1 2
     update_b 0 1
     pop_b 1
     mulI
     rtn
```

M68k-code:

```
Fac.1:
   CMP #0,D0
   BNE Fac.2


   MOVE #1,D0
   RTS
Fac.2:
   MOVE D0,(A4)+
   SUB #1,D0
   JSR Fac.1


   MUL -(A4),D0
   RTS
```

The previous example clearly shows that the main task of the Concurrent Clean to ABC-code compiler is to define some order of evaluation in which the B-stack is used if possible. It does not try to optimise the stack manipulations, for instance by avoiding redundant move operations. The last-mentioned kind of optimisations are done by the ABC to M68k code generator.

5.0 Parallel Implementation

Also the parallel implementation is based on the ABC machine. In this section we will present the parallel ABC machine, and its implementation aspects.
The basic assumption we make for this machine is that we have a processor topology, where each processor has its own local memory. On each processor a number of sequential ABC machines can be running. For each new process, created by a {P} or {I} annotation, a new sequential ABC machine (a *reducer*) is started. Each reducer has its own stacks. Reducers on the same processor share the heap of that processor.

The reservation/locking mechanism

Because several reducers on one processor can share subgraphs some reservation mechanism is necessary. In the parallel ABC machine this is done as follows.
A reducer evaluates a node by executing the code pointed to by the code pointer of that node. The first this code does is changing the code field of the node. The new code pointer points to a piece of code with which other reducers that will try to evaluate this node will be suspended:

```
_reserve:   set_wait 0
            suspend
            rtn
```

If a reducer executes this code sequence it puts itself (by the `set_wait` instruction) into the waiting list of the node it wanted to reduce. Thereafter it suspends itself with the `suspend` instruction.
After some time the node will be updated by the first reducer (note that nodes are updated only with head normal forms). Then also the reducers in the waiting list will be released. They all execute the return (`rtn`) instruction and continue as if they has reduced the node themselves.
In first instance it seems as if a waiting list will enlarge the fixed size part of all nodes: each node must have enough room to store a pointer to such a list. However, a node with a waiting list is under reduction and no information of this node is needed anymore. Therefore, in a concrete implementation other fields of the node can be misused.

Communication

There are two moments at which a graph has to be shipped to another processor. First, with the {P} annotation a remote reducer has to be started. The graph this reducer has to evaluate has to be copied to the processor on which the reducer will be started. The second case occurs if a result of a reduction is needed on another processor.
These forms of graph copying are basically the same. Copying a graph is not straightforward, since its structure has to be preserved. So, the copying algorithm has to take account of sharing and cycles. Also special action is needed if reserved nodes or nodes on which a reducer will be started (by an {I} or {P} annotation) risk to be copied. Reserved nodes can be recognised by the code pointer (or, alternatively, a flag might have been set). For nodes on which a reducer will be started a special node, called a *Defer* node, is inserted. In both cases simple copying of these nodes would mean duplication of work. Instead special nodes are created: *channel* nodes. Such a channel node is also created in the case of the {P} annotation: it points to the graph that will be reduced by the new remote reducer.
So, a channel node can be seen as a node containing a pointer to a remote graph. It has a special code pointer:

```
_channel_code:
        set_entry       _reserve 0
        send_request    0
        suspend
        rtn
```

If such a node is evaluated a request will be sent to another processor (by the `send_request` instruction). The reducer sending the executing this code will suspend itself. As soon as the requested graph is in head normal form it will be sent. The channel node will be updated with this graph. Note that a request is sent

only once: the code pointer is set to the reserve code, so other reducers will be suspended immediately. If a channel node is reduced, it is needed. Thus a request is sent only if the channel node is needed. Lastly we note that also channel nodes can be copied. The result will be a copy of the channel node.

6.0 Results

Current Status

Currently, the Concurrent Clean to ABC compiler has been fully implemented on various machines. It includes all aspects mentioned earlier and it compiles quite quickly. On a SUN3/280 it compiles roughly 150 lines of Concurrent Clean code per second. This is without strictness analysis. With strictness analysis compilation time approximately doubles.

For the ABC machine both a simulator and several code generators exist. The simulator is used for testing both sequential and parallel versions of the ABC machine. For the parallel part the simulator has some global knowledge of a real run time system of a parallel machine. In particular, it includes a parallel garbage collector, and a stack reallocation mechanism.
At this moment several versions of an ABC code to machine code compiler are available. The best one generates code for the MC68020 type of processor, and has been implemented both on the MacintoshII as well as on a SUN3. Also for the Transputer a code generator exists. The last one is a preliminary version of a code generator for a parallel machine. In the future this code generator will be extended with the same optimisation techniques as the other ones.

Sequential

We compared the implementation of our system with implementations of Lml, Hope and C on the SUN3 (with a MC68020, 25Mhz processor). The Lml system is considered as a standard implementation of a lazy functional language (notice that we do not present figures for Miranda: most of the benchmarks below do not terminate within reasonable time). The Hope system is an example of a fast implementation of a strict functional language. The imperative languages are represented by C. It should be stated that, if possible, C has been used in an imperative way (i.e. using iteration instead of recursion). The following implementations of these languages where used:

Lml	The Chalmers Lazy ML compiler, version 0.99.2, (90/08/20) (Augustsson & Johnsson (1989)).
Hope	The Hope+ compiler, release 3.2.1, August 1989 (Burstall et al. (1980)).
C	The gnu C compiler, version 1.36 (which generally gives faster code than the standard C compiler).

The following test programs where used:

nfib	the well known nfib program with argument 30.
tak	the Takeuchi function, called with (tak 24 16 8).
sieve	a program which generates the first 10000 primes, using quite an optimal version of the sieve of Eratosthenes (outputs only the last one).
queens	counts all solutions for the (10) queens problem.
reverse	a program which reverses a list of 3000 elements 3000 times.
twice	four times the twice on the increment function.
revtwice	four times the twice of the reverse of a list of 30 elements.
rnfib	again the nfib program, but now working on real numbers, with argument 26.
fastfourier	the fast fourier algorithm, on an array of 8K complex numbers. In the Concurrent Clean program a complex number is defined as a strict tuple of two reals.

	Clean	Lml	Clean (u!)	Hope	C	Clean (-!)
nfib	4.5	25	4.5	5.4	11	30
tak	4.9	40	4.9	7.2	11	36
sieve	8.1	25	6.8	9.1	4.5	12
queens	28	62	14	16	4.1	45
reverse	64	108	50	65	--	51
twice	1.7	SegFault	0.5	0.3	--	1.7
revtwice	27	OutOfHeap	9	12	--	39
rnfib	11	26	11	33	19	19
fastfourier	34	--	19	--	9.0	--

Table 6.1 Performance Overview (All times in seconds cpu time)

The following notes have to be made:

- The Lml versions of twice and revtwice resulted in run-time errors for these values.
- The reverse and twice programs make no sense in the C context. The sieve and fast fourier programs are iterative versions. The other ones are inherently recursive.
- Computing the fast fourier with the other functional languages is impossible: they all would run out of heap space.
- The times needed to generate an executable for the example programs vary widely. On an average, the Concurrent Clean implementation consumes about 3.5 seconds cpu time, the Lml system needs 6 seconds and the Hope system even 15 seconds.

The first two columns of the table compare a standard compilation of Concurrent Clean programs with Lml. The default reduction strategy is lazy, but strictness information is added automatically by the strictness analyser. It is obvious that in all cases Concurrent Clean outruns Lml.
The next two columns present a comparison between user annotated Clean and Hope. User annotations are inserted at some places that are not indicated by the strictness analyser. Some of these annotations can be found automatically by a clever analysis (but not by strictness analysis), as is the case for the sieve and the queens programs. The annotations for the fast fourier (in the type definition of the complex number) *have* to be added by the programmer. Again, Concurrent Clean produces in almost all the cases the fastest code although the differences are not that great anymore. The only case in which Hope is faster is the twice example. This is mainly because Hope uses a smart integer representation. This is indicated by the revtwice program, which also tests the implementation of higher order functions but avoids the use of integers.
The recursive programs written in C appear to be slower than the ones written in Concurrent Clean. However, the iterative versions of the examples written in C are faster. But, in comparison with the past, the difference between execution times of on the one hand the functional languages and on the other hand the imperative languages has significantly decreased.
The last two rows of the table are measurements for real arithmetic. In fact, they show that of the functional languages only Concurrent Clean supports reals seriously.
Finally, the last column gives execution times for Concurrent Clean programs for which no annotations were added, neither automatically by the strictness analyser, nor by the programmer himself. From these figures we can conclude that in general strictness annotations increase the efficiency. The largest gain is achieved in programs which largely manipulate objects of basic types as is the case with tak and fast fourier.

Parallel

Partly funded by the ESPRIT Parallel Computer Action and the Dutch Neural Network Project, recently a beginning has been made with the implementation of Concurrent Clean on a Transputer system composed of 64 Transputers. Currently this implementation supports only multi-processing on a single Transputer. Therefore, it is not yet possible to present performance figures of executions on a real parallel machine. However, with the PABC simulator a number of preliminary observations have been made.

The main results concern the kinds of parallelism which are possible, and how the parallel annotations influence this.
The process annotations are very powerful: it appears that many kinds of parallelism can be created. Also, it appears that the optimisations of the sequential code can be used in the parallel programs. The main problem in here is to assure that the grain size of the tasks is big enough.
The main disadvantage is that often very many reducers are needed to achieve a certain behaviour (for instance, each channel requires a reducer serving it). Also, the process annotations have to be used very carefully. Sometimes they have to be combined with local strictness annotation to provide that processes are created at the moment they are wanted. Some programs tend to behave sequential or create too many reducers if annotation are used wrongly.

7.0 Future work

The efficiency of the sequential code can be further improved by adding a special so-called application depended "strictness" analysis to the system. Such an analysis tries to determine whether eager evaluation of arguments for a certain application is safe because for this specific application it is known that these arguments will be evaluated (inspite of the fact that the applied function is not known to be strict in these arguments for the general case). Program transformations will be investigated that yield larger basic blocks of ABC code such that an optimal use of the new code generator is made.

We hope to demonstrate in the near future that real speed-ups can be achieved on a parallel architecture such as a Transputer system (Kesseler (1990)). At UEA already some promising results have been obtained with a previous version of our Clean system (McBurney & Sleep (1990)).
Furthermore, the presented annotations will be extended in order to enable the fine tuning of load balancing on a parallel machine.
On a higher level of abstraction new annotations are investigated to make parallel functional programming more user friendly (Eekelen & Plasmeijer (1990)).

8.0 Conclusions

The language Concurrent Clean is a lazy, higher-order functional graph rewriting language with as special feature that the sequential and parallel reduction order can be controlled in a general way. In Concurrent Clean arbitrary, dynamically changing process topologies can be specified. Parallel evaluation and communication can be controlled by the programmer. Parallel programs can also be executed on a sequential machine either by using simulated parallelism provided by the Concurrent Clean system or by ignoring the annotations giving rise to standard sequential evaluation.

There are several optimisations incorporated in the compiler such that, after a reasonable compilation time, very efficient execution is obtained for the sequentially evaluated parts of the code. The differences in speed between functional programs written in Concurrent Clean and programs imperatively written in a language like C are now becoming acceptable. Most optimisations are still applicable when code is generated for parallel environments.

The expressive power of the concurrency primitives available in Concurrent Clean make it possible that a new class of parallel algorithms can be expressed adequately in a functional language.

Simulations have shown that the speed obtained for sequential machines can be inherited for parallel architectures such that efficient, parallel functional programming will be possible.

References

Augustsson L., Johnsson T. (1989), 'The Chalmers Lazy-ML Compiler', The Computer Journal, Vol. 32, No. 2 1989.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R., (1987a), 'Term Graph Reduction', Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, The Netherlands, LNCS Vol. 259, pp. 141-158, June 1987.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R., (1987b). Towards an Intermediate Language based on Graph Rewriting. Proceedings of Parallel Architectures and Languages Europe (PARLE), part II, Eindhoven, The Netherlands. *Springer Lec. Notes Comp. Sci.* 259, 159-175.

Barendregt, H.P., Eekelen, M.C.J.D. van, Plasmeijer, M.J., Hartel, P.H., Hertzberger, L.O., Vree, W.G., (1987), 'The Dutch Parallel Reduction Machine Project', Intern. Conf. on Frontiers in Computing, Amsterdam, Dec. 1987.

Barendregt, H.P., Eekelen, M.C.J.D. van, Glauert, J.R.W., Kennaway, J.R., Plasmeijer, M.J., Sleep, M.R. (1988). 'Towards an Intermediate Language based on Graph Rewriting'. Revised version. *Journal of Parallel Computing* 9 with selected papers of the conference on Parallel Architectures and Languages Europe (PARLE), Eindhoven, The Netherlands. North-Holland 163-177.

Brus, T., Eekelen, M.C.J.D. van, Leer, M. van, Plasmeijer, M.J. (1987). Clean - A Language for Functional Graph Rewriting. Proc. of the Third International Conference on Functional Programming Languages and Computer Architecture (FPCA '87), Portland, Oregon, USA, *Springer Lec. Notes on Comp.Sci.* 274, 364 - 384.

Burstall, R.M., MacQueen, D.B., and Sanella, D.T. (1980). Hope: An Experimental Applicative Language. Proceedings of the 1980 LISP Conference, 136 - 143.

Eekelen, M.C.J.D. van, (1988). Parallel Graph Rewriting, Some Contributions to its Theory, its Implementation and its Application. University of Nijmegen. Ph.D. Thesis.

Eekelen, M.C.J.D. van, Plasmeijer, M.J., Smetsers, J.E.W., (1989b). Communicating Functional Processes. University of Nijmegen. Technical Report 89-3.

Eekelen M.C.J.D. van, Nöcker E.G.J.M.H., Plasmeijer M.J., Smetsers J.E.W., (1990). 'Concurrent Clean, version 0.6', Technical Report 90-21, University of Nijmegen, December 1990.

Eekelen, M.C.J.D. van, Plasmeijer, M.J., Smetsers, J.E.W., (1991). 'Parallel Graph Rewriting on Loosely Coupled Machine Architectures' proceedings of the workshop on CTRS'90. Montreal Canada. To appear in 1991.

Eekelen, M.C.J.D. van, Plasmeijer, M.J., (1990). 'Concurrent Functional Programming'. Proceedings of the conference on Unix & Parallelism, NLUUG, may 1990, pp 75-98.

Glauert, J.R.W., Kennaway, J.R., Sleep, M.R., (1987), 'DACTL: A Computational Model and Compiler Target Language Based on Graph Reduction', ICL Technical Journal, May 1987.

Groningen J. van. (1990). 'Implementing the ABC-machine on M680x0 based architectures'. Master Thesis, University of Nijmegen, November 1990.

Johnsson Th. (1987). 'Compiling Lazy Functional Programming languages'. Dissertation at Chalmers University, Götenborg, Sweden. ISBN 91-7032-280-5.

Kesseler M., (1990), 'Concurrent Clean on Transputers', Master Thesis, University of Nijmegen, November 1990.

Koopman, P.W.M., Nöcker, E.G.J.M.H. (1988), 'Compiling functional languages to Term Graph Rewriting Systems'. Technical Report 88 - 1, University of Nijmegen.

Koopman P.W.M., Eekelen M.C.J.D. van, Nöcker E.G.J.M.H., Smetsers S., Plasmeijer M.J. (1990). 'The ABC-machine: A Sequential Stack-based Abstract Machine For Graph Rewriting'. Technical Report, University of Nijmegen.

McBurney, D, Sleep, R. (1990), 'Concurrent Clean on Zapp', Proceedings of the Second International Workshop on Implementations of Functional Languages on Distributed Architectures, University of Nijmegen, November 1990.

Milner, R.A. (1978). Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, Vol. 17, no. 3, 348 - 375.

Mycroft, A. (1984). Polymorphic type schemes and recursive definitions. Proc. of the 6th Int. Conf. on Programming, *Springer Lec. Notes Comp. Sci.* 167, 217 - 228.

Nöcker E.G.J.M.H., (1989). 'The PABC Simulator, v0.5. Implementation Manual'. University of Nijmegen, Technical Report 89-19.

Nöcker E.G.J.M.H. (1990). 'Strictness Analysis based on Abstract Reduction', in Proceedings of the Second International Workshop on Implementation of Functional Languages on Parallel Architectures, pp. 297-321, Technical Report no. 90-16, October 1990, University of Nijmegen.

Nöcker E.G.J.M.H., Smetsers J.E.W., (1990). 'Partially Strict Data Types', Proceedings of the Second International Workshop on Implementations of Functional Languages on Distributed Architectures, University of Nijmegen, November 1990.

Peyton Jones S.L, Salkild J. (1989). 'The Spineless Tagless G-machine'. Proceedings of the Conference on Functional Programming Languages and Computer Architectures, Addison Wesley, pp 184 - 201.

Plasmeijer, M.J., Eekelen, M.C.J.D. van (1989). Functional Programming and Parallel Graph Rewriting. Lecture notes, University of Nijmegen, to appear at Addison Wesley 1991.

Smetsers J.E.W., (1989). 'Compiling Clean to Abstract ABC-Machine Code'. University of Nijmegen, Technical Report 89-20.

Smetsers, J.E.W., Eekelen, M.C.J.D. van, Plasmeijer, M.J., (1991). Operational semantics of Concurrent Clean. University of Nijmegen. Technical Report: in preparation.

Turner D.A. (1985), 'Miranda: A non-strict functional language with polymorphic types'. Proc. of the conference on Functional Programming Languages and Computer Architecture, Springer Lec. Notes Comp. Sci. 201, 1 - 16.

THE SCRIPTIC PROGRAMMING LANGUAGE

André van Delft
Delftware Technology B.V., Gentsestraat 165,
2587 HP Den Haag, The Netherlands.
E-mail: delft@fwi.uva.nl

INTRODUCTION

Over the last years a vast number of parallel languages have been developed. A large part of these are not available on 'normal' computers, or they do not cooperate easily with other languages. This paper describes Scriptic, a parallel language that extends widely used sequential languages (C, C++). Scriptic offers great expressiveness by incorporating many concepts from process theory dealing with concurrency and communication. Its availability on normal computers allows many software developers to experiment with parallel programming.

Basicly, Scriptic extends the C language with powerful and concise alternatives for the sequence operator: instead of the semicolon you can write for instance a comma for a parallel composition and a bar for choice. Scriptic has already been applied successfully in areas such as simulations, language parsers and graphical user interfaces.

This article is intended as a brief introduction to Scriptic programming. It will discuss some theory, explain most Scriptic operators and language features, and then illustrate their use through some real-world examples. Although Scriptic has also been designed as an extension to Pascal and Modula-2, these versions will not be discussed in this paper. Basic knowledge of C is assumed.

THE THEORY

The main flow structure in most programming languages is the sequence. It is often represented by a semicolon such as in Pascal and C. In the last decade a family of process theories has emerged: CSP, CCS, Process Algebra [2, 3, 4, 6, 7]. These suggest that other kinds of flow, such as choice and parallelism, can be covered in a comparable way. So if we already have a semicolon for sequence, then why not add a bar for choice, a comma for parallelism, and others?

Scriptic was founded on Process Algebra (PA). This theory is about *process expressions* and other representations of processes, which are well comparable to programs. In PA *atomic actions* are the building bricks (rather than assignments etc.) and constructs such as sequence and choice act as a kind of *glue*. PA basically defines *axioms* for sequence and (exclusive) choice, e.g.,

```
x|y = y|x
```

This axiom states that choice is symmetric: the order of operands is irrelevant. Other basic axioms state laws of associativity and distribution for sequence and choice. Sequence and choice are somewhat comparable with mathematical multiplication and addition, and with *and* and *or* in logic. Additional axioms in PA define constructs as parallelism in terms of sequence and choice. The atomic actions in parallel processes do not happen synchronously, but in an interleaved mode. They cannot be preempted by other atomic actions.

PA also defines axioms for two special processes: *deadlock* and the *empty process* (or *immediate success*). These correspond with the mathematical *zero* and *one*, or with the truth values *false* and *true* in Boolean Algebra (BA), a mathematical treatment of logic to which PA is closely related.

Currently research is done on combining BA and PA into a single theory dealing both with actions and truth values, or booleans. It seems that deadlock and the empty process are equal to the boolean values *false* and *true*. We see mixtures of actions and truth values already in conventional programming languages, where boolean expressions that are guards in if-statements determine which actions will be executed. The combination of BA and PA appears more directly in Scriptic. It suggests amongst others two kinds of parallelism: *and-parallelism* and *or-parallelism*. These are pure process generalisations of *and* and *or* for truth values. Although born as theoretical piece of work, or-parallelism has shown of significant practical use in Scriptic programs.

PRIMARY LANGUAGE CONSTRUCTS

Scriptic gets Process Algebra to work by letting fragments of C code placed between braces playing the role of atomic actions. (Unlike the situation in C, code fragments need not to end with a semicolon before the right-hand brace.). With symbols as the semicolon for sequence and the bar for exclusive choice you can make *script expressions* which have much in common with statement sequences in C. There is a function-like refinement construct named *script* which can have parameters and local variables. Unlike in C function calls, empty parameter lists in script calls may be omitted. One can specify groups of scripts in a *scripts section*, as in

```
scripts
  Hello   = {printf ("Hello world!")}
  Goodbye = {printf ("Goodbye!")}
  main    = Hello; Goodbye
```

In later examples we will leave out the keyword `scripts`. Various operators are available for Script expressions, as shown by table 1 in their priority order. Several operators denote parallelism:

- the comma for normal parallelism or *each-parallelism*: each operand should succeed in order to let this parallel composition succeed.
- the plus for *or-parallelism*: as soon as one operand terminates successfully the others are discarded.
- the ampersand for *and-parallelism*: as soon as one operand enters a deadlock state the others are discarded.

- The slash denotes *breaking*: operands will be discarded as soon as an operand more to the right starts to execute.

With the operators you can make many variations of the previous program, such as:

```
main =  Hello, (Goodbye | Hello)
```

This script will print "Hello" twice, or "Hello" and "Goodbye" in either sequence. When you specify such a non-deterministic program you do not know what will happen. The Scriptic implementation is then free to choose; its choice does not need to be random. It is better to let alternatives in choices start with input actions so that input will drive the program, or with guards so that you get a kind of if-statements.

operator	description
|	exclusive choice
/	breaking: left terminates when right starts
;	"and-then": normal sequence
+	or-parallelism: terminate on any success
&	and-parallelism: terminate on any deadlock
,	each-parallelism: each operand should succeed

Table 1: primary script operators

PARAMETERS

Like C functions, scripts can have parameters. The parallel constructs in Scriptic suggest a more powerful parameter mechanism. Therefore Scriptic offers three kinds of parameters:

- *Input parameters*, which are the same as in C functions (including passing references to variables). Example:

```
PrintChar('a')
```

- *Output parameters*: these behave like input parameters but upon success of the script the formal value is copied onto the actual variable. Output parameters are suffixed by a question mark, as in:

```
Read(c?)
```

- *forcing* parameters. Instead of a variable as an actual output parameter, you can specify a *forcing* value, which is suffixed by an exclamation mark. Thus

```
Read('x'!)
```

will only pass any success onwards if the formal parameter equals the value 'x'.

The definition header of `Read` shows that its parameter can both be used as output and forcing:

```
Read(char c?!) = .....
```

PROCESS COMMUNICATION

Most parallel languages offer a means to let parallel processes communicate. In some of these, communication just involves data transfer. In Scriptic parallel processes communicate through shared scripts. There are two ways to specify this, one inspired by Process Algebra, and one inspired by CSP:

1. With a *pair of communicating scripts*, one presumable representing a sender and the other a receiver. An example of a definition of two scripts that can communicate:

   ```
   Send,Receive = {printf("We have communication!")}
   ```

   When `Send` and `Receive` have been activated in parallel (as in `main = Send,Receive`) then this may lead to printing the message (*may*, because the program may have alternatives, as in `main = Send,Receive|Read(c?)`). `Send` and `Receive` may also communicate with other actions if you specify so, like

   ```
   Send,Receive1 = {printf("Another communication!")}
   ```

   Like normal scripts, communicating partners may have parameters. They may also have array indexes, e.g.,

   ```
   S[10 i](int j),R[]= {printf("S[%d](%d),R[]",i,j)}
   ```

2. As a *channel*. This is handy when communication mainly involves transferring data from the sender to the receiver. Sending and receiving over a channel involves arrows denoting the direction of the communication. You specify a channel for instance as:

   ```
   aChannel<-->(int i?!) = {printf("int passed!")}
   ```

   Such a communication may happen if two parallel processes have activated sending and receiving over this channel, as in

   ```
   P1 = ..... aChannel<-(1 ) .....
   P2 = ..... aChannel->(i?) .....
   P3 = ..... aChannel->(1!) .....
   main = P1,P2,P3
   ```

   The kinds of communication discussed here take place within a program. Scriptic also supports communication with external entities, which may be programs running on other processors, or devices such as the keyboard. One can for instance declare a *semi channel*: a channel which is declared with only a single arrow:

   ```
   Key->(char c?!) = {.....try to read a character.....}
   ```

   A call as `Key->(c?)` does not need a matching call like `Key<-('x')`; instead the body of the declaration of `Key` will contain code for getting a character from the keyboard buffer. In general, Scriptic's execution mechanism makes semi channels faster than normal scripts for communication with external devices.

CODE FRAGMENTS

Code fragments have several information attached to them, so that program execution can be directed. It is often not required to address these attributes explicitely, but some will be used more frequently in particular application areas (e.g., simulations):

- *priority* - the same notion as in operating systems: only code fragments with the highest priority level are allowed to run
- *preference* - comparable to priority, but it only determines the evaluation order of fragments.
- *duration* - a simulated number of time units, mainly for use in simulations
- *eventtype* - the type of event that the code fragment specifies. This mechanism cooperates with the operating system or windowing managers for input handling etc. For instance, you may define that eventtype 1 corresponds to keyboard input. Then before a code fragment with eventtype 1 is executed, the Scriptic *events management module* will check whether a key has been pressed.
- *success* - a variable which normally is 1. If the code fragment sets `success` to 0 the fragment *fails*: Alternative fragments may then be tried out, possibly with lower priorities. Often `success` will depend on the value of a boolean expression, as in:

  `{success = booleanValue}` and `{someStatements; success = booleanValue}`

  In a shorthand notation for setting `success` you may start the fragment with a question mark, as in:

  `{? booleanValue}` and `{? someStatements, booleanValue}`
  (the comma here is a C operator, chaining multiple expressions!)

  `return` returns from the code fragment - not from the script it occurs in. It is possible to chain code fragments with a question mark; the chaining point returns when `success` equals 0. So

  `{cCode; if (!success)return; moreCode}` is equal to `{cCode}? {moreCode}`

ACTIVATION AND DEACTIVATION

The attributes `priority`, `preference` and `eventtype` have default values 0. If these should be different then they need to be set before a code fragment is executed. This is possible in special *activation code fragments*, which are attached to expressions by means of a less sign. (For a summary of secundary script operators, see table 2.) The attributes are also valid for parts of expressions, and they are inherited downwards upon activation. So in

```
{priority=3;eventtype=1} < ( {priority++ }< A | {eventtype=0}< B)
```

`A` will get priority 4 and eventtype 1, whereas `B` will get priority 3 and eventtype 0.

A *deactivation code fragment* follows an expression and a greater sign. It is executed upon deactivation of the expression. There is some correspondence of the less and greater signs that redirect input and output in DOS and UNIX. You also have the notion of activation and deactivation in C: think of function calls. When a function call begins parameters are evaluated which may involve the execution of other pieces of code. On deactivation the function is removed from the call stack.

CONDITIONS

Scriptic has `if` and `switch` statements like in C, with a few minor differences so that they can be used in parallel and alternative contexts: there is no semicolon between the if-part and the else-part; in the `switch` statement "falling through" between different (non-empty) cases is prohibited.

Guards or *preconditions* offer lower level control in Scriptic. A guard is a code fragment enclosed in braces and followed by a colon. When such a guard fails it blocks the code right to it, creating a deadlock situation which can only be bypassed by executing alternatives. For instance:

```
{? c1}: e1 | {? c2}: e2
```

offers a choice between `e1` and `e2`; each alternative may or may not be blocked. Beware: the notion of blocking is absent in if statements in Scriptic and C (these have a notion of bypassing). Only in Pascal compilers there is a notion of blocking in `CASE` statements (they often gave the run-time error messages when no appropriate case tag was there).

There is a way of saying with guards Scriptic that you bypass `e1` if `c1` is not met: enclose a set of guarded alternatives in brackets to obtain a *guarded selection*. The following phrases are equivalent:

```
if (c1) e1
[{? c1}:e1]
```

A minus specifies an explicit "else" clause, so three Scriptic equivalent for if-else statements are:

```
if (c1) e1    else  e2
[{? c1}:e1 | {?!c1}:e2]
[{? c1}:e1 |        :e2]
```

An empty pair of brackets `[]` denotes the *neutral element*: it means really *nothing*. In sequential and parallel context it does not block. It is the hidden "else" clause in a guarded selection without explicite "else". As a kind of syntactic sugar, Scriptic offers also an if-else construct: `if(c1)e1 else e2`. Note that semicolons are absent.

Postconditions are code fragments attached to script expressions with an exclamation mark. They are executed on success of their operands. When they fail they create a deadlock, as in:

```
SomeScript !{.....C code that leaves success nonzero.....}
```

ITERATIONS

Most programs contain iterations, and most programming languages offer special iteration constructs. Let's first see what basic things could be relevant to specify in the context of iterations, and how they are handled in languages as C:

- The simple fact that a part of the program is an iteration. In C you see this by the keywords `do`, `while`, `for`.

- The type of the iteration: is it sequential, alternative (i.e. does it denote a number of choices of the same type), parallel (does it denote a number of processes of the same type). In C all loop constructs are sequential, but that is not necessary in a language which treates sequence as just one basic construct, among others such as choice and parallelism.

- The *exit points*, or the place(s) where the iteration can halt. For each exit point: *where* does it occur in the iteration, is it *mandatory* or *optional*, and on *what condition* is it selected. An exit from a C for-loop is possible at any position with the keyword `break`, although the for-construct strongly suggests a conditional exit at the entrance of the loop. Moreover, all exit points in C loops are *mandatory*: if an exit condition is reached and it yields true, then the iteration definitely terminates. Compiler generator languages such as Yacc support *optional* exit points, where loop continuation depends on the context instead of explicit conditions.

- The structural differences between subsequent iteration passes. Often the first iteration pass is different from later passes: typically in many loops the first pass initializes control variables which are changed in subsequent passes. The C for-construct offers a convenient means to specify the differences between the first pass and subsequent passes.

- The indexes of the iteration passes, which are in a sense like the indexes of array elements. Often variables are defined just to act as loop counters.

Now, how does Scriptic handle these issues? To begin with, Scriptic offers `while` and `for`, comparable to C. The differences are that `while` and `for` may occur as an operand anywhere in an expression, and the iterations need not be sequential. The type of the iteration appears as an operator next to the `while` or `for` operand. Note that there is no need for a `do-while` construct.

But to make more general iterations, Scripitic has a general iteration symbol: the pair of periods: . . . Any expression with a primary operator (semicolon, comma, bar etc.) is an iteration when at least one of its operands contains the iteration symbol. But the

pair of periods does more than that: it can also specify an exit point for the iteration. Normally this is a mandatory exit point, as in these reusable iterators:

```
UNTIL (int t) =  if( t) ..
WHILE (int t) =  if(!t) ..
EXIT          =    ..
```

Often you will need an *optional* exit point: enclose the pair of periods with brackets. For instance,

```
Hellos = [..]; Hello
```

prints **zero** or more times "`Hello world!`". Omitting the brackets would result in an immediate exit point, so `Hellos` would do nothing. It makes sense to define an *iteration script* such as `SOME = [..]`. **One** or more times "`Hello world!`" would then be `Hello; SOME`. There may be more exit points than one in the loop. Exit points may occur anywhere. For instance, consider the following typical iteration:

```
Prompt = {printf("'h' for Hello; 'x' to exit\n")}
main   = (Prompt; SOME; Key->('h'!); Hello);
                        Key->('x'!); Goodbye
```

This prompts the operating instruction; if the user answers by pressing 'h' the Hello message appears, followed by the operating instruction, and so on. The loop ends when the user answers with 'x' instead of 'h'. Other keys are not accepted by this program.

It now becomes clear how Scriptic simplifies programming tasks. Consider the way you had to achieve interactive, or event-driven, programs with conventional languages. Many of these, especially the ones with graphical user interfaces, contain a *main event loop*. This is a section that repeatedly tests what kind of event has happened, such as a user input action, and then reacts to this event. The reaction may set a variable for quitting the loop. You see this typical frame-work in the following C-version of the previous Scriptic example:

```
void main()
{
  int done=0;
  do {prompt();
      switch (getch()) {
      case 'x':  done=1; break;
      case 'h': hello(); break;
  }   }
  while (!done);
  goodbye();
}
```

The pair of periods can *also* occur as an *operator* with high priority, instead of a kind of operand denoting exit. It then separates a number of script expressions as a *pass selector*, such as in

```
FORi(int i?,int b,int e) =  {i=b}..{i++}< if(i>e) ..
```

This is quite a complicated script, but you can place it in a library and use it without thinking of the internals. The idea is that `FORi` changes upon activation a loop counter `i`, which is also an output parameter. This change depends on the pass in which `FORi` is called: the first time `i` is set to the begin value `b`, and in subsequent passes `i` is

incremented. The loop terminates when `i` exceeds the end value `e`. The pair of periods occurs here both as an operator (pass selector) and as an operand (exit point)!

The pass selector may also be used in a script `MANY` to make an expression an iteration without providing an exit point (as in eternal loops). The first pass the script should do nothing (`= []`), as well as in the second pass. The definition of `MANY` looks like `SOME`, but that is just a coincidence: `MANY = [] .. []`

In iterations the variable `pass` is often useful; `pass` denotes the pass number of the nearest sequence, choice, or parallel construct construct that the code fragment is defined in. The first pass value is 0, like array indexes in C. If the sequence, choice etc. is an iteration, the code fragments that are activated in subsequent passes get increasing pass values. For example, you may specify a script `TIMES` and a script for printing '0' up to '9' as:

```
TIMES (int n) = if(pass>=n) ..
Print0To9     = TIMES(10); {printf ("the pass is: %d\n",pass)}
```

You can make a similar expression that accepts keys '0' up to '9', as an example of an alternative iteration:

```
aKeyIn0To9 = TIMES(10) | Key->('0'+(int)pass !)
```

The foregoing contains the raw material to make your own iterators. This is among the hardest things in Scriptic. Beginners can use a library with the following scripts: `SOME`, `MANY`, `EXIT`, `WHILE`, `UNTIL`, `FOR`, `TIMES`.

operator	description
:	precondition
!	postcondition
?	code chaining
<	activation
>	deactivation
..	pass selection

Table 2: Secundary script operators

EXAMPLE 1: THE GAME OF LIFE

For the first Scriptic example application we look at the interactive part of the game of Life by Conway. In this game a human player defines which cells of a playboard are *alive* or *dead*, so that an initial pattern of living cells appears on a computer screen. The pattern on the board starts to change according to a certain algorithm, as soon as the player commands so. A keyboard is used for command input; the script `Key` accomplishes this.

How to program this in Scriptic? The main program will first perform a number of initializations. Then several processes should become active in parallel: for changing the state of the cells, moving the cursor, controlling the speed, getting help, a "clock that makes the real time pass at the right speed", and a process that tests whether execution has to terminate. This behavior is reflected in the script `main`:

```
main = Init;Changes+Moves+SpeedAdjs+Helps+Clock+Exit
Init = {.....initialization statements.....}
```

Changing the state of the cells can be done by the user (toggling a cell state where the cursor resides) or clearing the entire screen, or by the program, which generates new board patterns according to the Life algorithm in the function `Generation`. Starting and stopping generations by the program is controlled by the scripts `Start` and `Stop`, which react on the enter key or the return key. Between subsequent generations a delay will occur according to the current speed: the script `DELAY` is a sequential loop of an action which as a duration equal to 1. It will end when the number of passes has reached the value `TIME(Speed)` (which happens to be a macro for `256 >> Speed`). It also has an optional exit point, so that the user can escape from it early by triggering `Stop`. With the plus as parallel operator in `Changes` we can easily accomplish that generations end as soon as the user toggles a cell or clears the screen.

```
DELAY       = SOME; {duration=1};
              while (pass<TIME(Speed))
Changes     = MANY; Clear+Toggle+Generations
Generations = Start;(SOME;{Generation()};DELAY);Stop
Clear       = Key->( 'c'!) !{clearboard()}
Toggle      = Key->( ' '!) !{togglecell()}
Start       = Key->('\r'!) | Key->('\n'!)
Stop        = Key->('\r'!) | Key->('\n'!)
```

The exclamation marks in `Clear` and `Toggle` could well have been semicolons, but they are more efficient (the actions `clearboard()` and `togglecell()` will now occur in the same execution pass as the acceptance of the key). The three other main processes, `Moves`, `SpeedAdjs`, and `Helps`, are sequential iterations of simple tasks: accepting a key from the keyboard and then reacting to it. You find them in listing 1.

The `Clock` process couples the "Scriptic time" with the "real time": each duration unit will correspond with `SLEEP` microseconds. Therefore the clock ticks actions with duration equal to 1; after each tick a "real time" delay occurs:

```
Clock = MANY; {duration=1} !{delay(SLEEP)}
Exit  =        Key->('!'!) !{leave("Exit program")}
```

`Exit` is unlike the other processes not an eternal `MANY` loop, so that the program will display a message and terminate when the key 'x' has been pressed.

EXAMPLE 2: A GRAPHICAL USER INTERFACE

Up to now, programming user interfaces à la Macintosh has been considered hard: mostly you have to program a complicated *main event loop* which handles all kinds of user input (menu commands, dragging a window, key presses etc.). With Scriptic these tasks are easily handled by processes. As an example, consider the classic example program for Macintosh computers [1]: a simple text editor within a window, offering basic windowing functions and editing functions.

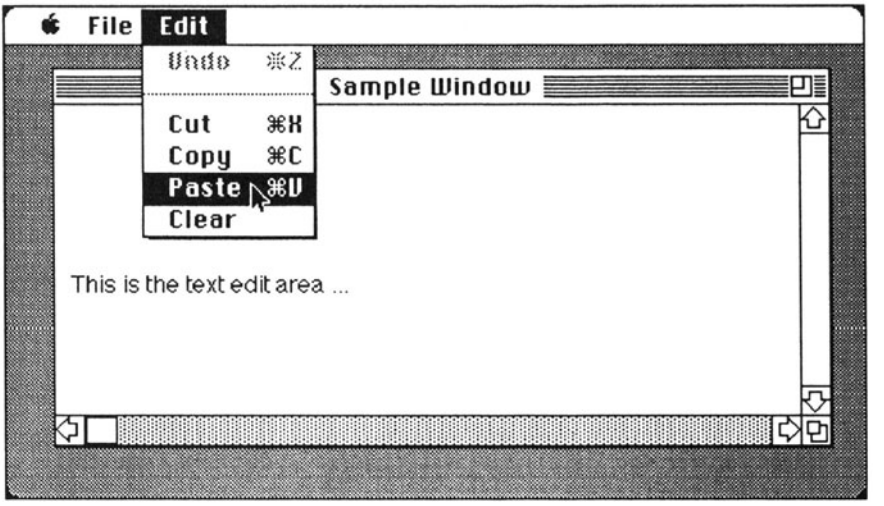


Figure 1: The classic Macintosh example program

In Scriptic you can specify handlers for several kinds of tasks:

- window actions: dragging, growing, zooming, closing, updating, activating, deactivating, clicking the mouse so that the window becomes active, clicking so that the text cursor is moved.
- menu commands; there are three menus: *Apple* (with desk accessoiries), *File* and *Edit.*
- passing key input to the text editor.
- updating the cursor shape.
- handling messages from the operating system.
- letting the text cursor blink when idle.
- termination.

Scriptic enables a programming style that makes it easy to grow and maintain the program, especially as compared to a main event loop:

```
mainscript     = WindowHandler + MenuHandler + KeyHandler
               + CursorHandler + OSHandler   + IdleHandler
               + Exit

Exit           = {? DoneFlag}

WindowHandler = MANY; WindowAction

WindowAction   = Drag | ZoomIn  | GoAway |   Activate
               | Grow | ZoomOut | Update | DeActivate
               | ClickTECursor  | ClickWindow

MenuHandler    = short Menu,Item;
                 MANY; AcceptCmd(Menu?,Item?) !{AdjustMenus()};
                       HandleCmd(Menu ,Item ) !{HiliteMenu(0)}

HandleCmd(short m,short i) = switch(m) (
                             case mApple: AppleCommand(i)
                             case mFile :  FileCommand(i)
                             case mEdit :  EditCommand(i))
```

```
AppleCommand(short i) = short daRefNum; Str255 daNm;
                        if(i==iAbout}) {(void) Alert(rAboutAlert, nil)}
                        else         {GetItem(GetMHandle(mApple),i,daNm);
                                      daRefNum = OpenDeskAcc(daNm)}

FileCommand (short i) = switch(i) (
                        case iNew   : {DoNew()}
                        case iClose: {DoCloseWindow(FrontWindow())}
                        case iQuit  : {DoneFlag=Terminate()}          )

EditCommand (short i) = TEHandle te; WindowPtr w;
                        if(!SystemEdit(I1))
                          {te=((DocumentPeek)(w=FrontWindow()))->docTE}:
                          switch(i) (
                          case iUndo : {DoUndo (te)}
                          case iCut  : {DoCut  (te)}
                          case iCopy : {DoCopy (te)}
                          case iPaste: {DoPaste(te)}
                          case iClear: {DoClear(te)})
                          ! {AdjustScrollbars(w, 0); AdjustTE(w)}

KeyHandler            = char c; MANY;Key->(c?);{TEKey(c)}
```

The complete set of scripts is shown in listing 2.

EXAMPLE 3: A QUEUE SIMULATION

Scriptic is ideally suited for simulations such as discrete event systems and communication protocols. Consider the following example: For a number of days customers arrive at a shop during opening hours according to a Poisson distribution. Then they enter a First-In-First-Out queue for service; a fixed number of servers are available. After service the customers leave the shop.

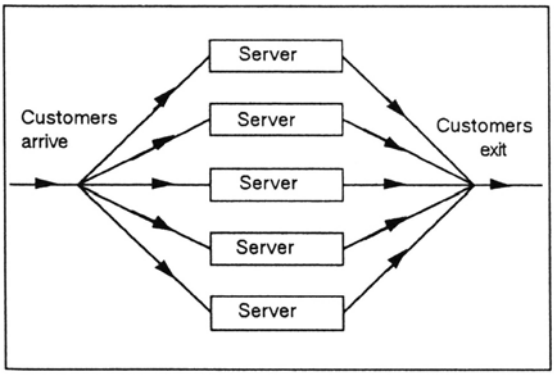


Figure 2: A simple queueing system

Apart from several scripts and C functions for reporting and iterations, a Scriptic program for this case is:

```
scripts

 ServeCust(int s)   ,
 GetServed(int c)   = {duration=RNDNEGEXP(30);
                       printf("%s-%s customer %3u is served by %2u\n",
                              time2str(SimTime),
                              time2str(SimTime+duration),c,s)}

 Customer(int c)    = int t; {t=SimTime  }< Enter(c); GetServed(c);
                        {qtime+=SimTime-t}< Leave(c)

 Day (int i)        = ( (HOURS (9); POISSON (3,Customer))
 /*9 to 5*/           + (HOURS(17); TheShopCloses) )
                    &    HOURS(24); TheDayEnds(i)

 Days (int n)       = TIMES(n); Day((int)pass)

 Info   =  MANY; Key->('?'!);{reports()}
 Exit   =        Key->('!'!);{printf("\nExit \n")}
 main   =  InitShopSim; WORKERS(10,ServeCust)+Days(4)+Info+Exit
```

The script `ServeCust` will communicate with `Getserved`; this will take a negative exponentially distributed amount of time. `RNDNEGEXP` and `Service` are defined in the program as a random function and a tracing function. `POISSON` and `WORKERS` are defined elsewhere as reuasable scripts: `POISSON` "launches" processes with random time intervals; it applies the special bracket pair `<* ..... *>` that marks an expression to be launched. `WORKERS` generates a number of workers; each workers is willing to perform its task for an unlimited number of times.

`Enter,Leave,TheShopCloses,DayEnds` will print messages; `Infos` prints information each time the '?' key is pressed. The plus in `main` is an or-parallel operator: `WORKERS` and `Infos` never want to stop. The program terminates normally after 4 simulated days. Abrupt termination takes place on pressing '!'.

Another plus is present in `Day`: its first operands is a Poisson generator that starts at 9 a.m.. The other operand is ready at 5 p.m., so that the Poisson generator will then be terminated. At 12 p.m. the day will end, due to the and-parallel process `HOURS(24); TheDayEnds(i)`.

SCRIPTIC-C++

Scriptic has recently been combined with the C++ language. That is: scripts will become available as object features, next to variables and functions, thus bringing objects to life. The first results indicate that the expressiveness of both Scriptic and C++ is largely increased, without an extra performance penalty. In particular, Scriptic-C++ is elegant in simulations.

In the shop simulation example, customers may be modeled as C++ objects, with a `Live` script that tells what they will do. The customer class also has a *static* script `Poisson` for launching customers (static items belong to the class as a whole rather than to a single object):

```
scripts

  ServeCust(int s),
  Getserved(int c) = {duration=RNDNEGEXP(20); Service(s,c,duration)}

class Customer {
    int i;
    static int n;  // #customers generated

    Customer(void) {i=n++;} // constructor

  scripts
    EnterShop = {printf("%s enters: %d \n",time2str(SimTime), i)}
    LeaveShop = {printf("%s leaves: %d \n",time2str(SimTime), i)}

    Live = EnterShop; Getserved(i); LeaveShop; {delete this}

  static scripts
    Poisson(int t) =
            for(;;); {duration=RNDNEGEXP(t)};
                     <*{new Customer}->Live*>
};

scripts

  Generator    =  Customer::Poisson(3)
               + {duration=8*hours};
               CloseShop()
// etc.
```

Having a script `Poisson` in the class Customer has the disadvantage of not being generic: it can only launch Live scripts of new customers. Because of its staticness and C++ inheritance rules, Poisson is not available in subclasses. As an alternatve, it would be possible to apply the earlier `POISSON` script that accepted as a parameter the script to be launched. Then, we again have to specify in the second parameter that a new customer process is to be launched, which now also involves allocating a new customer object.

We can use a so-called *script block*, another new feature, comparable to anonymous functions in Lisp and to code blocks in Smalltalk: enclose a script expression between rectangular brackets, let it optionally proceed by a parameter list, and separate these parts with a semicolon, as in `[: {new Customer}->Live]`. We can specifiy this block as a parameter to `POISSON`:

```
  Generator =   POISSON(3, [: {new Customer}->Live])
              + {duration=8*hours};
              CloseShop()
```

THE IMPLEMENTATION

Scriptic is currently available on PC-MS/DOS, Macintosh and Sun computers [5]. The programmer's package comes with a preprocessor (≈110k), a run-time system (≈40k) and an events management module (1...15k, depending on the operating system or windowing system). The latter two must be linked with each Scriptic application.

Execution speed: in the Life example Scriptic overhead is hardly visible. When generating new patterns the life algorithm and the screen output functions are by far the most time consuming. When running the Macintosh example on a Macintosh-plus the user turns out to be the slowest process involved; he does not notice any slower responses by the program due to Scriptic. In the queue example, simulating 1000 customers served by 10 servers takes 4 seconds on a Sun Sparcstation 1.

The preprocessor translates Scriptic-C into C, generating two functions for each script. This approach has enabled:

- importing and exporting scripts between modules
- passing script names as parameters
- script blocks
- scripts in C++ classes

Scriptic has been designed mainly with programmer efficiency in mind. Parallelism in Scriptic means parallel *programming*, not parallel *processing*. The language may already be used on distributed systems, but each processor should then have a private Scriptic program running, and the programmer should take care of communication between the processors; the communication features of Scriptic are valid within the domain of a single processor.

At the moment of writing we are investigating whether Scriptic can give more support for multi-processor architectures. Some of the language features, such as the allowance of variables that appear globally, may be an obstacle for a parallel implementation. Probably we may deal with these issues by making use of an existing parallel extension of C that runs on multi processor architectures, with Scriptic as a more programmer-friendly shell.

NOTES

[1] Apple 1985. *Inside Macintosh, Volume I*, Addison-Wesley, Reading, Massachusetts

[2] Baeten, J.C.M., Weijland, W.P., *Process Algebra*, Cambridge University Press, 1990

[3] Bergstra J.A.,Klop J.W. 1984. "Process algebra for synchronous communication." *Information & Control* 60 (1/3): 77-121.

[4] Bergstra J.A.,Klop J.W. 1986. "Algebra of communicating processes." In *Proceedings CWI Symposium on Mathematics&Computer Science* (de Bakker J.W., Hazewinkel M., Lenstra J.K., eds) North-Holland, 61-94.

[5] Delftware Technology BV, *Scriptic Programmer's Manual*, Den Haag, March 1991.

[6] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall 1985

[7] Milner,R., *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, Springer, 1980

LISTING 1: THE GAME OF LIFE

```
#include "unixem.h" // Scriptic Events Management module
                    //          exports GetChar and iteration scripts
#include "lifei.h"  // functions for screen output and life algorithm
#define SLEEP    25 // # microseconds corresponding with duration==1
#define delay(t) usleep(t*1000)
#define TIME(s)  (256>>s)

short Speed=9;
char *Msg, *helptext[] = {
  "",
  "    Scriptic(r) Demonstration: Conway's Game of Life",
  "",
  "        Copyright (c) 1990, Delftware Technology",
  "                            The Netherlands",
  "  Usage:",
  "  ! = exit                   h     = home",
  "  ? = info (this message)    c     = clear",
  "  > = faster                 space = toggle cell status",
  "  < = slower                 enter = start/stop generation",
  "                            ^",
  "                            |",
  "  cursor movements:     <-- + -->",
  "                            |",
  "                            v" };
scripts
  .
  WAIT        = SOME; {priority++; preference--}< {}
  DELAY       = SOME; {duration=1}; UNTIL (pass>=TIME(Speed))

  Clear       = Key->(  'c'   !) ![clearboard  ()]
  Toggle      = Key->(  ' '   !) ![togglecell  ()]
  Move        = Key->(  'h'   !) ![cursorhome  ()]
              | Key->( homeKey!) ![cursorhome  ()]
              | Key->(   upKey!) ![cursorup    ()]
              | Key->( downKey!) ![cursordown  ()]
              | Key->(rightKey!) ![cursorright ()]
              | Key->( leftKey!) ![cursorleft  ()]
  SpeedAdj    = Key->(  '>'   !) ![if(Speed<9) status(++Speed,Msg)]
              | Key->(  '<'   !) ![if(Speed>0) status(--Speed,Msg)]
  Start       = Key->( '\r'   !)
              | Key->( '\n'   !)
  Stop        = Key->( '\r'   !)
              | Key->( '\n'   !)
  Exit        = Key->(  '!'   !) ![leave("Exit program")]
  Help        = Key->(  '?'   !)
              ![help(helptext); status(Speed,Msg="Press '?' to continue")];
          WAIT; Key->(  '?'   !)
              ![redraw(); status(Speed,Msg="Press '?' for help")]
  Init        = {init();  status(Speed,Msg="Press '?' for help")}

  Generations = Start; (SOME; {Generation()}; DELAY); Stop
  Changes     = MANY; Clear+Toggle+Generations
  Moves       = MANY; Move
  SpeedAdjs   = MANY; SpeedAdj
  Helps       = MANY; Help
  Clock       = MANY; {duration=1; delay(SLEEP)}

  main        = Init; Changes + Moves + SpeedAdjs + Helps + Clock + Exit
```

LISTING 2: A GRAPHICAL USER INTERFACE

```
scripts

mainscript     = WindowHandler + MenuHandler + KeyHandler
               + CursorHandler + OSHandler   + IdleHandler + Exit

Exit           = {? DoneFlag}

CursorHandler = {priority++} < {? AdjustCursor(event.where,cursorRgn),0}
IdleHandler   = {priority--} < {? DoIdle       (),0}

KeyHandler     = char c; MANY;   Key(c?) !{DoKeyDown     (window,c)}

OSHandler      = MANY; OSEvent
OSEvent        = ClickFromSystem !{SystemClick  (&event, window)}
               | SuspendEvent    !{DoDeActivate (FrontWindow())}
               |  ResumeEvent    !{DoActivate   (FrontWindow())}

WindowHandler = MANY; WindowAction
WindowAction   = Drag | ZoomIn  | GoAway |   Activate | ClickWindow
               | Grow | ZoomOut | Update | DeActivate | ClickTECursor

Drag           = MouseDown(inDrag    !) !{ DragWindow (window,event.where,scrn)}
Grow           = MouseDown(inGrow    !) !{DoGrowWindow(window,event.where)}
GoAway         = MouseDown(inGoAway  !) !{?TrackGoAway(window,event.where)}
                                        !{DoCloseWindow(window)}
ZoomIn         = MouseDown(inZoomIn  !) !{?TrackBox (window,event.where,inZoomIn)}
                                        !{ DoZoomWindow(window,inZoomIn )}
ZoomOut        = MouseDown(inZoomOut!)  !{?TrackBox (window,event.where,inZoomOut)}
                                        !{ DoZoomWindow(window,inZoomOut)}

Update         =            UpdateEvent !{  DoUpdate     (window)}
Activate       =          ActivateEvent !{  DoActivate   (window)}
DeActivate     =        DeActivateEvent !{DoDeActivate   (window)}
ClickWindow    =  MouseInPassiveWindow  !{SelectWindow   (window)}
ClickTECursor  =   MouseInActiveWindow  !{DoContentClick(window,&event)}

MenuHandler    = short Menu,Item; MANY;AcceptCmd(Menu?,Item?)!{AdjustMenus()};
                                       HandleCmd(Menu ,Item )!{HiliteMenu(0)}

AcceptCmd (short m?, short i?)
               = char c; long MI; //Menu Item
               ( CmdKey    (c?)        !{AdjustMenus();MI=MenuKey (c)}
               | MouseDown(inMenuBar!)!{AdjustMenus();MI=MenuSelect(event.where)})
               !{m=HiWord(MenuItem);i=LoWord(MI)}

HandleCmd (short m,short i) = switch(m) ( case mApple: AppleCommand(i)
                                          case mFile :  FileCommand(i)
                                          case mEdit :  EditCommand(i))

AppleCommand (short i)   = short daRefNum; Str255 daNm;
                           if(i==iAbout}) {(void) Alert(rAboutAlert, nil)}
                           else           {GetItem(GetMHandle(mApple),i,daNm);
                                           daRefNum = OpenDeskAcc(daNm)}

FileCommand(short i) = switch(i) ( case iNew   : {DoNew()}
                                   case iClose: {DoCloseWindow(FrontWindow())}
                                   case iQuit : {DoneFlag=Terminate()}       )

EditCommand (short i)    = TEHandle te; WindowPtr w;
                           if(!SystemEdit(I1))
                               {te=((DocumentPeek)(w=FrontWindow()))->docTE}:
                              switch(i) ( case iUndo : {DoUndo (te)}
                                          case iCut  : {DoCut  (te)}
                                          case iCopy : {DoCopy (te)}
                                          case iPaste: {DoPaste(te)}
                                          case iClear: {DoClear(te)})
                                                ! {AdjustScrollbars(w, 0); AdjustTE(w)}
```

LISTING 3: A QUEUE SIMULATION

```
#include <math.h>
#include <stdio.h>
#include "scrandom.h" /*various random functions and macros*/
#include "unixem.h"

static int  TotCust; // total number of generated customers
static double qtime; // total queueing times, including service times*/

static void   reports()
{printf ("*********************************************************\n");
 printf ("# customers         = %4u\n", TotCust);
 printf ("total waiting time = %7.11f minutes\n", qtime);
 printf (" mean waiting time = %2.2lf minutes\n", TotCust?qtime/TotCust:0.0);
 printf ("    simulated time = %s days:hours:minutes\n", time2str(SimTime));
 printf ("*********************************************************\n");}


scripts

  Info              = MANY; Key->('?'!); {reports()}
  Exit              =       Key->('!'!) !{printf("\nExit Program\n")}

  InitShopSim       = {TotCust=0; qtime=0.}
  TheShopCloses     = {printf("%s closes\n",  time2str(SimTime)); reports()}
  TheDayEnds(int i) = {printf("%s Day ends\n",time2str(SimTime)); reports()}

  Enter(int c)      = {printf("%s enters: %3u\n",time2str(SimTime),c);TotCust++}
  Leave(int c)      = {printf("%s leaves: %3u\n",time2str(SimTime),c)}

  ServeCust(int s)  ,
  GetServed(int c)  = {duration=RNDNEGEXP(30);
                       printf("%s-%s customer %3u is served by %2u\n",
                        time2str(SimTime),time2str(SimTime+duration),c,s)}

  Customer(int c)   = int t; {t=SimTime  }< Enter(c); GetServed(c);
                        {qtime+=SimTime-t}< Leave(c)

  Day (int i)       = ( (HOURS (9); POISSON (3,Customer))
    /*9 to 5*/        + (HOURS(17); TheShopCloses) )
                    &    HOURS(24); TheDayEnds(i)

  Days (int n)      = TIMES(n); Day((int)pass)

  main              = InitShopSim;
                      WORKERS(10,ServeCust) + Days(4) + Info + Exit
```

Structural Operational Semantics for Kernel Andorra Prolog *

Seif Haridi[1] and Catuscia Palamidessi[1,2]

[1]Swedish Institute of Computer Science,
Box 1263, S - 164 28 KISTA, Sweden

[2]Centre for Mathematics and Computer Science,
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands
email: katuscia@cwi.nl
and
Department of Computer Science, Utrecht University
P.O. Box 80089 3508 TB Utrecht, The Netherlands

Abstract

Kernel Andorra Prolog is a framework for nondeterministic concurrent constraint logic programming languages. Many languages, such as Prolog, GHC, Parlog, and Atomic Herbrand, can be seen as instances of this framework, by adding specific constraint systems and constraint operations, and optionally by imposing further restrictions on the language and the control of the computation model.

We systematically revisit the description in Haridi and Janson [HJ90], adding the formal machinery which is necessary in order to completely formalize the control of the computation model. To this we add a formal description of the transformational semantics of Kernel Andorra Prolog. The semantics of Kernel Andorra Prolog is a set of *or-trees* which also captures infinite computations.

1 Introduction

Kernel Andorra Prolog is language framework that is specifically designed to combine the programming paradigms of Prolog and committed choice languages [HJ91], allowing fully general combinations. The proposed family of languages are guarded definite clause languages, with deep guards, and three guard operators (wait, cut, and commit). In general, the machinery of deep guards is necessary in nondeterministic languages, for selecting a single solution, or collecting all solutions for a given goal. In particular the generalization to deep guards is essential to achieve the goal of simultaneously subsuming Prolog and exploiting independent and dependent parallelism. Deep guards can

*The visit at SICS of Catuscia Palamidessi, during which this work was carried out, has been supported by the project Andorra

also be used to encapsulate nondeterministic transformational parts of a program while maintaining a reactive indeterministic computation at an outer level.

The computation model of Kernel Andorra Prolog (KAP) is a generalization of the Andorra Model for pure definite clauses [War87, HB88]. The Andorra Model exploits implicit and-parallelism in the execution of definite clauses. The generalized model features a carefully controlled nondeterminism, which is available uniformly in a computation.

The framework is parameterized with the constraint system used, and the chosen set of constraint operations with their respective activation conditions. Also, in some specific cases, sequential ordering between goals is necessary to achieve the desired synchronisation effects.

The exposition depends partly on an intuitive understanding of KAP as given in [HJ90]. However we have tried to make the paper as self containted as possible within the size limit.

2 The Basic Andorra Model

The Andorra model is defined for pure Horn clauses. It gives priority to deterministic computation over nondeterministic computation, as nondeterministic steps are likely to multiply work.

The Andorra model divides a computation into *deterministic* and *nondeterministic phases.* First, all atomic goals for which it is known that at most one clause would succeed are reduced using a single clause during the deterministic phase. (These goals can be reduced in and-parallel.) Then, when no such goal is left, some goal is chosen for which all clauses are tried; this is called the nondeterministic phase. The computation then proceeds with a deterministic phase on each or-branch.

The key concept here is the notion of determinacy. An atomic goal is said to be *deterministic* when there is at most one candidate clause that would succeed for the goal. As soon as it is known that an atomic goal has become deterministic, the goal can either be reduced by a single clause, or fail, if it was known that no clause would apply. It is not considered to be an error if the mechanism for detecting the determinacy of goals fails to detect that a goal is deterministic. In general, nothing less than complete execution will establish this property.

The Andorra model has a number of interesting consequences.

Firstly, the Andorra model allows deterministic goals to be run in and-parallel, extracting implicit and-parallelism from the program.

Secondly, the notion of determinacy in the Andorra model gives a reasonably strong form of *synchronization.* As long as a goal is able to produce data deterministically, no consumer of this data is allowed to run ahead (if it does not know what to consume). This allows specification of concurrent processes.

Thirdly, the Andorra model reduces the search space by executing the deterministic goals first. Goals can fail early, and the constraints produced by a reduction can reduce the number of alternatives for other goals. This has been proved to be very relevant for the coding of constraint satisfaction problems [Kor89, Sar89b, BG89, HB88, Yan89].

The Andorra model in items:

- An atomic goal fails if it is known that no clause would succeed for the goal.
- An atomic goal can be reduced using a single clause when it is known that all other clauses would necessarily fail for the goal.
- When no goal is known to be deterministic, all clauses in its definition are tried for some goal.

3 The Extended Computation Model

The extended computation model takes advantage of the principles underlying the Andorra model to control nondeterminism in a "deep" concurrent language. Atomic goals may start in and-parallel, performing local computations, by an operation called *local forking.* Local computations are recursive Andorra computations that are logically independent. A local computation may receive information from its environment, but cannot communicate its results to the uncle goals until a promotion operation is performed. In KAP each clause is divided into a guard and a body. Local computations are performed by the goals of the guards of the candidate clauses. Promotion takes place when one or more guard executions terminate (depending on the guard operator). Promotion has three main forms. *Determinate promotion* is performed when a local computation reduces to a single branch. This generalizes the Andorra determinacy test. *Nondeterminate promotion* is performed when a local computation reduces to several branches; this introduces nondeterminism by creating an or-tree and distributing the uncle goals into each branch of the local computation. *indeterminate* promotion is performed when the guard operator is a pruning operator, selecting only one successful branch of the local computation.

Now, the language and the configurations are defined. Then, the transition rules that start guard execution, perform commit, etc, are described.

3.1 Kernel Andorra Prolog (KAP): the language

Let *Var* be an infinite set of variables, with typical elements $x, y, z, ...$, Let *Con* be a set of n-adic data constructors, with typical elements $a, b, c...$ (constant symbols) and $f, g, h, ...$ (function symbols). Terms (*Term*), $t, u, ...$, atoms (*Atom*), $A, B, ...$ and substitutions (*Subst*) are defined as usual. Elementary atoms $H, K, ...$, are atoms of the form $p(\bar{x})$, where p is a predicate and $\bar{x}$ is a tuple of distinct variables. A clause is an object of the form

$$H \text{ :- } \mathbf{choice}_{\%}(T_1, \ldots, T_n)$$

such that the T_i's are simple guarded goals (see below). The variables in H are called *parameters.* The symbol % stands for a guard operator. There are three possible guard operators, namely '!', '|' or ':'. A *KAP program* is a set of clauses. Each clause represents the definition of one predicate. We assume that each predicate occurring in the program is defined by exactly one clause of the program.

⟨*program*⟩	::=	⟨*set of clauses*⟩
⟨*clause*⟩	::=	⟨*head*⟩ :- ⟨*simple choice box*⟩
⟨*head*⟩	::=	⟨*elementary atom*⟩
⟨*simple choice box*⟩	::=	**choice**⟨*guard operator*⟩(⟨*sequence of simple guarded goals*⟩)
⟨*simple guarded goal*⟩	::=	[⟨*simple guard*⟩]⟨*body*⟩
⟨*simple guard*⟩	::=	∃⟨*set of variables*⟩.**and**(⟨*sequence of atomic goals*⟩;**true**)
⟨*body*⟩	::=	⟨*sequence of atomic goals*⟩
⟨*atomic goal*⟩	::=	⟨*elementary atom*⟩ | ⟨*constraint operation*⟩
⟨*guard operator*⟩	::=	':' | '!' | '|'

For technical reasons, we assume that in a simple choice box

$$\mathbf{choice}_{\%}([\exists Z_1.P_1]\bar{B}_1, \ldots, [\exists Z_n.P_n]\bar{B}_n)$$

the sets of variables $Z_1, \ldots, Z_n$ are pairwise disjoint. Moreover, we assume that $Z_1, \ldots, Z_n$ contain all local variables of a clause, namely those variables occurring in the choice box and not in the head. A program is considered to be closed under all possible variable renaming.

The language is parametrized with a *constraint theory*. The set of formulas of this theory, with typical elements $\vartheta, \sigma, \psi, \ldots$, will be denoted by *Constraints*. The existential closure of ϑ is denoted by $\exists(\vartheta)$. We say that ϑ is *consistent* iff $\models \exists(\vartheta)$, where the symbol $\models$ stands for logical validity with respect to the given theory. We assume that the theory is *decidable*, therefore ϑ is *inconsistent* iff $\models \neg\vartheta$. We say that ϑ *entails* σ iff $\models \vartheta \supset \sigma$.

Unification, and the like, are performed by primitive constraint operations. The notation $op(\psi)$ denotes a primitive operation *op* applied to the constraint ψ. A constraint operation may suspend until its *activation condition* is satisfied. Some primitive operations are described later (see section 6.1).

4 The transition system for Kernel Andorra Prolog

We define the operational semantics of KAP via a transition system. This system is essentially based on the rewriting system defined in [HJ90] for describing the computational model of Andorra Prolog.

The set of configurations is defined by the following grammar

$$
\begin{array}{lcl}
\langle goal \rangle & ::= & \langle or\ box \rangle \mid \langle and\ box \rangle \\
\langle or\ box \rangle & ::= & \mathbf{or}(\langle goal \rangle, \langle goal \rangle) \\
\langle and\ box \rangle & ::= & \exists \langle set\ of\ variables \rangle . \\
 & & \mathbf{and}(\langle sequence\ of\ local\ goals \rangle; \langle constraint \rangle) \\
\langle local\ goal \rangle & ::= & \langle atomic\ goal \rangle \mid \langle choice\ box \rangle \\
\langle choice\ box \rangle & ::= & \mathbf{choice}\langle guard\ operator \rangle(\langle sequence\ of\ guarded\ goals \rangle) \\
\langle guarded\ goal \rangle & ::= & [\langle goal \rangle]\langle body \rangle
\end{array}
$$

We use *Goal*, *LocalGoal*,... etc, to denote the sets generated by $\langle goal \rangle$, $\langle local\ goal \rangle$, ... etc. The symbols A, B, C stand for local goals, P, Q, R stand for goals, and S, T stand for guarded goals. Moreover, *GeneralGoal*, with typical element g, will denote the set $Goal \cup LocalGoal$.

The set of *sequences of existentially quantified constraints Seq*, with typical element s, is the smallest set such that

- $\lambda \in Seq$ (the empty sequence)
- $\forall \vartheta \in Constraints\ \forall s \in Seq\ \forall X \subseteq Var\ \exists X.\vartheta \diamond s \in Seq$

The symbol $\diamond$ stands for sequence concatenation.

The logical meaning Θ_s of a sequence s of existentially quantified constraints is defined as follows:

- $\Theta_\lambda = \mathbf{true}$
- $\Theta_{\exists X.\vartheta \diamond s} = \exists X.(\vartheta \wedge \Theta_s)$

Sometimes we will need the conjuction of constraints with all variables free; for this we use $\hat{\Theta}_s$ defined as follows:

- $\hat{\Theta}_\lambda = \mathbf{true}$
- $\hat{\Theta}_{\exists X.\vartheta \diamond s} = \vartheta \wedge \hat{\Theta}_s$

The notion of consistency and entailment is extended to sequences in the following way. A sequence s is *consistent* iff $\models \exists(\Theta_s)$, and it is *inconsistent* iff $\models \neg\Theta_s$. s *entails* $\exists X.\sigma$ (or, σ does not restrict the environment outside X) iff $\models \hat{\Theta}_s \supset \exists X.\sigma$ (or, equivalently, iff $\models \Theta_s \supset \Theta_{s \circ \exists X.\sigma}$).

The transition system is a pair $\langle Conf, \rightarrow \rangle$, where $Conf = GeneralGoal \cup \{\mathbf{fail}, \mathbf{deadlock}\}$, and $\rightarrow$ is a class of transition relations on $Conf$

$$\{\ell \rightarrow_s^m : \ell \in \{o, u\}, m \in \{F, G\}, s \in Seq\}$$

o, u stand for *ordered* and *unordered* respectively, and refer to the context of the configuration. A transition $c\ \ell \rightarrow_s^m c'$ will be represented as $\ell \vdash c \rightarrow_s^m c'$. When the context information is irrelevant to a particular transition rule, we omit the symbol $\ell \vdash$.

F, G stand for *guess free* and *not guess free* respectively. The subscript s indicates the environment that the configuration is assumed to have when the transition takes place. Namely, $c \rightarrow_s^m c'$ means that it is possible to make a transition from c to c' in the mode m when the environment of c is s.

In the following, we assume the program W to be fixed. In an and-box $\exists X.\mathbf{and}(\bar{A};\vartheta)$, X represents the set of immediate local variables of the box. This information is necessary to deal with execution and suspension of the constraint operations.

A tuple of goals is represented by a bar, so, for instance $\bar{S}, \bar{T}$ stand for tuples of guarded goals, etc. The empty and-box $\exists X.\mathbf{and}(;\vartheta)$ is simply represented by $\exists X.\vartheta$.

We introduce the notion of *variables local to a general goal*, $var(g)$.

- $var(p(x_1, \ldots, x_n)) = \{x_1, \ldots, x_n\}$
- $var(op(\psi)) = var(\psi)$
- $var(\mathbf{or}(P, Q)) = var(P) \cup var(Q)$
- $var(\exists X.\ \mathbf{and}(A_1, \ldots, A_n;\vartheta)) = X \cup var(A_1) \cup \ldots \cup var(A_n) \cup var(\vartheta)$
- $var(\mathbf{choice}_{\%}(S_1, \ldots, S_n)) = var(S_1) \cup \ldots \cup var(S_n)$
- $var([P]\bar{B}) = var(P)$

Computation Rules

Local Forking

An elementary atom A can be transformed into the choice-box associated with the definition of the predicate of A. The parameters are replaced by the arguments of A. This is expressed by the following rule

If $A \equiv p(y_1, \ldots, y_n)$ and $p(x_1, \ldots, x_n)$:- B belongs to W, where $var(B) \cap \{y_1, \ldots, y_n\} = \emptyset$, then

$$A \rightarrow_s^F B\alpha$$

where $\alpha = \{x_1/y_1, \ldots, x_n/y_n\}$ and $B\alpha$ is the application of the substitution α to B.

The structural rules of the transition system will guarantee that variables introduced by the local forking are different from the variables of the global configuration.

Primitive Constraint Rules

Constraint operations are the only ones that can modify the environment. There are three transition rules corresponding to successful execution, failure and suspension.

Some constraint operations, corresponding to the actions of existing languages like Prolog, GHC, Parlog, and Atomic Herbrand, are described in section 6.1.

$$\exists X.\mathbf{and}(\bar{A}, op(\psi), \bar{B}; \sigma) \rightarrow_s^F \exists X.\mathbf{and}(\bar{A}, \bar{B}; \sigma \wedge \psi) \quad \textit{if} \quad \begin{array}{l} activ(op, X, \psi, \sigma, s) \\ \models \exists(\Theta_{s \circ \exists X.\sigma \wedge \psi}) \end{array}$$

i.e. if the activation condition of $op(\psi)$ holds (with respect to X, σ and s) and $\exists X.\sigma \wedge \psi$ is consistent with its environment. The activation condition will depend both on the specific constraint operation and s.

Determinate Promotion

If after the completion of the guard execution only one guarded goal is left within a choice box, then it can be extracted, according to the following rule

$$\exists X.\mathbf{and}(\bar{A}, \mathbf{choice}_{\%}([\exists Y.\psi]\bar{B}), \bar{A}'; \sigma) \rightarrow_s^F \exists X \cup Y.\mathbf{and}(\bar{A}, \bar{B}, \bar{A}'; (\sigma \wedge \psi))$$

$$\textit{if} \models \exists(\sigma \wedge \psi)$$

Quiet Indeterministic Promotion

A guard execution is quiet if it results in an (empty) and-box, whose constraint does not restrict the environment outside the local variables of the box.

Cut

After a successful guard execution of one branch, the cut operator prunes all the branches to the right.

$$\mathbf{choice}_!(\bar{S}, [\exists Y.\sigma]\bar{B}, \bar{T}) \rightarrow_s^F \mathbf{choice}_!(\bar{S}, [\exists Y.\sigma]\bar{B}) \quad \textit{if} \models \hat{\Theta}_s \supset \exists Y.\sigma.$$

Commit

After a successful guard execution of one branch, the commit operator prunes all the other branches.

$$\mathbf{choice}_|(\bar{S}, [\exists Y.\sigma]\bar{B}, \bar{T}) \rightarrow_s^F \mathbf{choice}_|([\exists Y.\sigma]\bar{B}) \quad \textit{if} \models \hat{\Theta}_s \supset \exists Y.\sigma.$$

Or Reduction

The or boxes within a choice box are eliminated according to the following rule

$$\mathbf{choice}_{\%}(\bar{S}, [\mathbf{or}(P, Q)]\bar{B}, \bar{T}) \rightarrow_s^F \mathbf{choice}_{\%}(\bar{S}, [P]\bar{B}, [Q]\bar{B}, \bar{T})$$

Now we will describe the transitions of guessing rules, marked by the G flag. The use of these rules will later be restricted by the control principles of Kernel Andorra Prolog.

Nondeterministic Promotion

A goal is in an *ordered context* if the closest surrounding pruning choice is a cut choice, otherwise it is in an *unordered context.*

Ordered Context

In an ordered context, after the successful execution of the guard, the leftmost branch of a nondeterministic choice within an and-box can be promoted if the computed constraint is consistent with the one of the and-box.

$$o \vdash \exists X.\mathbf{and}(\bar{A}, \mathbf{choice}_{:}([\exists Y.\psi]\bar{B}, \bar{T}), \bar{A}';\sigma) \rightarrow_s^G \mathbf{or}(\ \exists(X \cup Y).\mathbf{and}(\bar{A}, \bar{B}, \bar{A}';(\sigma \wedge \psi)),\ \exists X.\mathbf{and}(\bar{A}, \mathbf{choice}_{:}(\bar{T}), \bar{A}';\sigma))$$

if $\models \exists(\sigma \wedge \psi)$.

Unordered Context

In an unordered context, after the successful execution of the guard, any branch of a nondeterministic choice within an and-box can be promoted if the computed constraint is consistent with the one of the and box.

$$u \vdash \exists X.\mathbf{and}(\bar{A}, \mathbf{choice}_{:}(\bar{S}, [\exists Y.\psi]\bar{B}, \bar{T}), \bar{A}';\sigma) \rightarrow_s^G \mathbf{or}(\ \exists(X \cup Y).\mathbf{and}(\bar{A}, \bar{B}, \bar{A}';(\sigma \wedge \psi)),\ \exists X.\mathbf{and}(\bar{A}, \mathbf{choice}_{:}(\bar{S}, \bar{T}), \bar{A}';\sigma))$$

if $\models \exists(\sigma \wedge \psi)$.

Noisy Indeterministic Promotion

Cut

When the guard execution of the leftmost branch left in a choice-box is completed successfully and the computed constraint is consistent with the constraint of the closest sourrounding and-box, then the cut operator prunes all the other branches (to the right) and the computed constraint is made public.

$$\exists X.\mathbf{and}(\bar{A}, \mathbf{choice}_{!}([\exists Y.\psi]\bar{B}, \bar{T}), \bar{A}';\sigma) \rightarrow_s^G \exists(X \cup Y).\mathbf{and}(\bar{A}, \bar{B}, \bar{A}';(\sigma \wedge \psi))$$

if $\models \exists(\sigma \wedge \psi)$.

Commit

When the guard execution of one branch in a choice-box is completed successfully and the computed constraint is consistent with the constraint of the closest sourrounding and-box, then the commit operator prunes all the other branches and the computed constraint is made public.

$$\exists X.\mathbf{and}(\bar{A}, \mathbf{choice}_{|}(\bar{S}, [\exists Y.\psi]\bar{B}, \bar{T}), \bar{A}';\sigma) \rightarrow_s^G \exists X \cup Y.\mathbf{and}(\bar{A}, \bar{B}, \bar{A}';(\sigma \wedge \psi))$$

if $\models \exists(\sigma \wedge \psi)$.

Structural Rules

The following rules allow us to derive the transitions of the configurations depending on the transitions that can be made by the components of the configurations. The rule for and boxes will be restrained by the condition that the new local variables introduced in a transition of a subgoal must be disjoint

with the variables of the other subgoals. This will ensure that all the local variables of different and components are always disjoint, thus evoiding clashes of variables.

Or boxes

Any transition made by a goal inside an or-box is propagated to the parent box.

$$\frac{\ell \vdash Q \rightarrow_s^m Q'}{\begin{array}{l}\ell \vdash \mathbf{or}(P,Q) \rightarrow_s^m \mathbf{or}(P,Q') \\ \ell \vdash \mathbf{or}(Q,P) \rightarrow_s^m \mathbf{or}(Q',P)\end{array}}$$

Note that, since Q' is a metavariable on *Goals*, $Q' \neq$ **fail**, **deadlock**.

Choice boxes

Any transition made by a goal inside a choice box is propagated to the parent box. An ordered transition can take place if the closest surrounding pruning choice box is a cut choice box. An unordered transition can take place if the closest surrounding pruning choice box is a commit choice box.

To formalize this notion, we introduce the following function $\mathcal{OU} : \{o,u\} \times \{:,!,|\} \rightarrow \{o,u\}$ (ordered-unordered) that filters the context informations out of the guard operator.

- $\mathcal{OU}(\ell,:) = \ell$
- $\mathcal{OU}(\ell,!) = o$
- $\mathcal{OU}(\ell,|) = u$

$$\frac{\ell \vdash Q \rightarrow_s^m Q'}{\ell' \vdash \mathbf{choice}_{\%}(\bar{S},[Q]\bar{B},\bar{T}) \rightarrow_s^m \mathbf{choice}_{\%}(\bar{S},[Q']\bar{B},\bar{T})} \quad \ell = \mathcal{OU}(\ell',\%)$$

And boxes

Any transition made by a goal inside an and box, with the assumption that the external environment is s, generates a transition for the parent box, with a weaker assumption s', such that s is the result of appending the sequence s' and the constraint (existentially quantified with respect to the local variables) of the and box. Intuitively, this model the fact that the environment of the goal consists of the environment and the constraint of the parent and box.

$$\frac{\ell \vdash B \rightarrow_s^m B'}{\ell \vdash \exists X.\mathbf{and}(\bar{A},B,\bar{C};\psi) \rightarrow_{s'}^m \exists X.\mathbf{and}(\bar{A},B',\bar{C};\psi)} \quad \begin{array}{l}(var(B') \setminus var(B)) \cap \\ var(\exists X.\mathbf{and}(\bar{A},B,\bar{C};\psi)) = \emptyset \\ s = s' \diamond \exists X.\psi\end{array}$$

Next we specify the transition rules for *failure* and *suspension*. Having explicitely these rules in the operational model of the language allows to gain in efficiency (the detection of *failure* allows to prune the failing choice branches).

Failure Rules

The following rules describe the transitions that bring to failure.

Constraint operations

Any primitive operation *op* will fail whenever the constraint is not consistent with the environment

$$op(\psi) \rightarrow^{F}_{s \diamond \exists X.\sigma} \textbf{fail} \quad \textit{if} \quad \models \neg\Theta_{s \diamond \exists X.\sigma \wedge \psi}$$

And boxes

An and box fails when the constraint is inconsistent with its global environment. Notice that this is the only rule (apart from the rules on primitives) that depends upon the constraints of the environment in the transition relation.

$$\exists X.\textbf{and}(\bar{A};\sigma) \rightarrow^{F}_{s} \textbf{fail} \quad \textit{if} \quad \models \neg\Theta_{s \diamond \exists X.\sigma}$$

Choice boxes

A choice box fails when there are no alternatives left.

$$\textbf{choice}_{\%}() \rightarrow^{F}_{s} \textbf{fail}$$

Structural Rules for Failure

The following rules describe the propagation of failing transitions from inner goals to the external configurations.

Choice boxes

Choice boxes simply eliminate failing branches.

$$\frac{Q \rightarrow^{F}_{s} \textbf{fail}}{\textbf{choice}_{\%}(\bar{S}, [Q]\bar{B}, \bar{T}) \rightarrow^{F}_{s} \textbf{choice}_{\%}(\bar{S}, \bar{T})}$$

And boxes

An and box fails whenever one of the local goals fails. Again, the environment condition of the internal transition is weakened in the external transition

$$\frac{B \rightarrow^{F}_{s} \textbf{fail}}{\exists X.\textbf{and}(\bar{A}, B, \bar{C};\psi) \rightarrow^{F}_{s'} \textbf{fail}} \quad s = s' \diamond \exists X.\psi$$

Suspension Rules

Here we describe the rules that bring a configuration to be suspended. We only describe *global suspension*, namely the *deadlock* of the whole goal.

Constraint operations

A constraint operation may suspend when the activation condition of the operation is not satisfied and the constraint is consistent with the environment (otherwise it would fail).

$$op(\psi) \rightarrow^{F}_{s \diamond \exists X.\sigma} \textbf{deadlock} \quad \textit{if} \quad \begin{array}{l} \neg activ(op, X, \psi, \sigma, s), \\ \models \exists\Theta_{s \diamond \exists X.\sigma \wedge \psi} \end{array}$$

Structural Rules for Deadlock

The following rules describe the propagation of deadlock from the inner configurations to the outer ones.

Choice boxes

Commit and Wait

If the the guard operator is | or :, then a choice box deadlocks whenever all the branches get deadlocked.

$$\frac{P_1 \rightarrow_s^F \textbf{deadlock}, \ldots, P_n \rightarrow_s^F \textbf{deadlock}}{\textbf{choice}_{\%}([P_1]\bar{B}_1, \ldots, [P_n]\bar{B}_n) \rightarrow_s^F \textbf{deadlock}} \quad \% \in \{|, :\}$$

Cut

If the the guard operator is !, then a choice box deadlocks whenever the first branch gets deadlocked.

$$\frac{P \rightarrow_s^F \textbf{deadlock}}{\textbf{choice}_!([P]\bar{B}, \ldots, \bar{S}) \rightarrow_s^F \textbf{deadlock}}$$

And boxes

An and box deadlocks whenever all the internal local goals deadlock. Observe that the deadlock of one local goal does not cause the deadlock of the whole and box since it can be resumed after the execution of some other local goals. This can be modeled by the following rule.

$$\frac{B_1 \rightarrow_s^F \textbf{deadlock}, \ldots, B_n \rightarrow_s^F \textbf{deadlock}}{\exists X.\textbf{and}(B_1, \ldots, B_n; \psi) \rightarrow_{s'}^F \textbf{deadlock}} \quad s = s' \diamond \exists X.\psi$$

We don't have failure and suspension rules for the or boxes. Internally or boxes are collapsed into choice boxes. For external or boxes, we want to preserve the shape of the tree. This issue will be exposed in details in the section on operational semantics.

5 Control of the computation model

A sequence of applications of the transition rules described in the previous section constitute an unrestricted derivation or computation of the extended Andorra computation. KAP computations are restricted. The control of KAP has been described informally and motivated in [HJ90]. We will just give a brief review to aid the understanding of the following semantic description. KAP always prefer deterministic steps over nondeterministic and noisy indeterministic steps. The F-marked rules: local forking, deterministic promotion, quiet pruning, constraint rules, etc. are called *guess-free* rules. Nondeterministic promotion and noisy pruning are called *guessing* rules.

Central to the control of the model is the notion of stability of and-boxes. Stable boxes are boxes that cannot be affected by computations in its surrounding environment regardless of the context in which it is running. An and-box is called *stable* if no guess-free rules are applicable to or within the box, and no constraint or constraint operation occurring in the box imposes new constraints on variables that are external to the box.

A context is a general goal "with a hole".

Definition 5.1 *The set Context, with typical element $C[\,]$, is the smallest set such that*

1. $[\,] \in \mathit{Context}$
2. *if* $C[\,] \in \mathit{Context}$ *then*
   - $\mathbf{or}(P, C[\,])$, $\mathbf{or}(C[\,], P) \in \mathit{Context}$
   - $\exists X.\ \mathbf{and}(\bar{A}, C[\,], \bar{B}; \vartheta) \in \mathit{Context}$
   - $\mathbf{choice}_{\%}(\bar{S}, [C[\,]]\bar{B}, \bar{T}) \in \mathit{Context}$

Whenever the expression $C[g]$ is legal it will denote the general goal obtained by "filling the hole" of $C[\,]$ with g. Note that g is a general goal.

The next definition formalizes the notion of total constraint of a global goal, which gives the set of all the constraint sequences in all the subgoals.

Definition 5.2 *The function* $tc : \mathit{GlobalGoal} \to \mathcal{P}(\mathit{Seq})$ *(the set of all sets of sequences) is defined as follows*

- $A \in \mathit{Atom} \Rightarrow tc(A) = \{\lambda\}$
- $op(\sigma) \in \mathit{Primitives} \Rightarrow tc(op(\sigma)) = \{\sigma\}$
- $S_i = [P_i]\bar{B}_i \Rightarrow tc(\mathbf{choice}_{\%}(S_1, \ldots, S_n)) = tc(P_1) \cup \ldots \cup tc(P_n)$
- $tc(\mathbf{or}(P, Q)) = tc(P) \cup tc(Q)$
- $tc(\exists X.\ \mathbf{and}(A_1, \ldots, A_n; \sigma)) = \exists X.\sigma \diamond (tc(A_1) \cup \ldots \cup tc(A_n))$

where the operation $\diamond$ *is extended on sets.*

We now introduce the notion of *stability* of and-boxes (with respect to an environment).

$$\mathit{stable}(A, s) \text{ if } \nexists c \in \mathit{Conf}\ [A \to_s^F c] \text{ and } \forall s' \in tc(A)\ [\models \hat{\Theta}_s \supset \Theta_{s'}]$$

The next definition formalizes the notion of *environment*: the environment of a configuration is the environment of the closest surrounding and-box plus its constraint. We define the environment as a function *env* from contexts to sequences of constraints in such a way that $env(C[\,])$ is the environment of the "hole" of $C[\,]$, i.e. the constraint seen by a legal goal g in $C[g]$.

Definition 5.3 (environment of a context) *The function* $env : \mathit{Context} \to \mathit{Seq}$ *is defined as follows:*

- $env([\,]) = \lambda$
- $env(\mathbf{or}(P, C[\,])) = env(\mathbf{or}(C[\,], P)) = env(C[\,])$
- $env(\exists X.\mathbf{and}(\bar{A}, C[\,], \bar{B}; \sigma)) = \exists X.\sigma \diamond env(C[\,])$
- $env(\mathbf{choice}_{\%}(\bar{S}, [C[\,]]\bar{B}, \bar{T})) = env(C[\,])$.

The following definition extends the function $\mathcal{OU}$ so to filter the informations of ordered and unordered out of generic contexts.

Definition 5.4

- $\mathcal{OU}_{\mathbf{ext}}(\ell, [\,]) = \ell$
- $\mathcal{OU}_{\mathbf{ext}}(\ell, \mathbf{or}(P, C[\,])) = \mathcal{OU}_{\mathbf{ext}}(\ell, \mathbf{or}(C[\,], P)) = \mathcal{OU}_{\mathbf{ext}}(\ell, C[\,])$
- $\mathcal{OU}_{\mathbf{ext}}(\ell, \exists X.\mathbf{and}(\bar{A}, C[\,], \bar{B}; \sigma)) = \mathcal{OU}_{\mathbf{ext}}(\ell, C[\,])$
- $\mathcal{OU}_{\mathbf{ext}}(\ell, \mathbf{choice}_{\%}(\bar{S}, [C[\,]]\bar{B}, \bar{T})) = \mathcal{OU}_{\mathbf{ext}}(\mathcal{OU}(\ell, \%), C[\,])$

The restricted Andorra Computation model is defined by the following transition system (based on the previous one). The transition relations are labeled by A and describe *admissible* applications:

- An application of a guess-free rule is always admissible

$$\frac{\ell \vdash g \rightarrow^{F}_{s} g'}{\ell \vdash g \rightarrow^{A}_{s} g'} \quad \textit{where } g' \in \textit{GlobalGoal} \cup \{\mathbf{fail}, \mathbf{deadlock}\}$$

- An application of a guessing rule is admissible iff

  1. it is applied to (or to a subgoal of) a stable and-box, and
  2. there are no admissible applications of guessing rules to proper subgoals of the rewritten box, i.e. it is innermost. Innermost application of G-transition of a goal g is defined as follows:

$$\textit{innermost}(g, \ell\, s) \text{ iff } \forall C[\,] \neq [\,]\, \forall g' \,[g = C[g'] \Rightarrow \nexists c\, (\mathcal{OU}_{\mathbf{ext}}(\ell, C[\,]) \vdash g' \rightarrow^{G}_{s \circ env(C[\,])} c)]$$

$$\frac{\ell \vdash g \rightarrow^{G}_{s} g'}{\ell' \vdash C[g] \rightarrow^{A}_{s'} C[g']} \quad \begin{array}{l} \textit{stable}(C[g], s'), \\ \textit{innermost}(g, \ell, s) \\ \ell = \mathcal{OU}_{\mathbf{ext}}(\ell', C[\,]) \\ s = s' \diamond env(C[\,]) \end{array}$$

- (Compositional Rule) An admissible transition of a stable box can be propagated to outer configuarations.

$$\frac{\ell \vdash g \rightarrow^{A}_{s} g'}{\ell' \vdash C[g] \rightarrow^{A}_{s'} C[g']} \quad \begin{array}{l} \ell = \mathcal{OU}_{\mathbf{ext}}(\ell', C[\,]) \\ s = s' \diamond env(C[\,]) \end{array}$$

Further restrictions of KAP computations are possible, but we regard such restriction as part of the semantics of the particular user languages based on KAP. For discussions of possible user languages see [HJ90].

6 Operational Semantics

We give now the definition of the operational semantics of AP in set-theoretic terms. Intuitively, the operational semantics of a program is defined by the semantics of the and-boxes that can be run under that program, and the semantics of an and-box consists of the set of all the possible collections of answers that can be obtained by running it. In general, all the answers that are delivered under certain particular indeterministic and nondeterministic choices are collected in an or box. The or box has a tree like structure, with the intermediate nodes labeled by **or** and the leaves labeled by and-boxes. The nonempty and-boxes correspond to computations not terminated yet, and we can define the semantics as the limit of the or trees obtained along a certain computation (transition) chain. This limit can be, in general, an infinite tree, as the computation can deliver an infinite set of answers. This allows us to preserve all the information about the computation in the semantic structure, and it leaves space for further abstractions.

Indeed, different implementations may lead to different notions of *observables*. For instance, we may imagine a situation similar to Prolog, in which the answers are presented in the order they are collected by the usual depth-first strategy. A loop along one branch will cause the unobservability of the answers possibly generated at the right of that branch. To model this, we can abstract from our semantics the sequence of answers obtained by collecting left-to-right the leaves of the tree corresponding to a successfully terminated computation. The sequence ends when we get in correspondence of a nonterminating and-box.

Another possibility is to collect in parallel all the answers generated, without any restriction on the order in which they appear. To model this we can abstract from our semantics the set of the leaves corresponding to a successfully terminated computation.

Definition 6.1 *The set $\mathcal{T}$ (the set of or trees) is the minimal set that satisfies the following conditions:*

- $\bot \in \mathcal{T}$
- $\mathbf{fail}, \mathbf{deadlock} \in \mathcal{T}$
- *if* $\vartheta \in$ *Constraints and* $X \subseteq$ *Var then* $\exists X.\vartheta \in \mathcal{T}$
- *if* $t, t' \in \mathcal{T}$ *then* $\mathbf{or}(t, t') \in \mathcal{T}$

The set $\mathcal{T}$ is ordered as follows

Definition 6.2 *The relation $\leq$ is the minimal ordering relation on $\mathcal{T}$ that satisfies the following conditions*

- $\forall t \in \mathcal{T}.\ \bot \leq t$
- $\forall t, t', u, u' \in \mathcal{T}.(t \leq u \wedge t' \leq u') \supset \mathbf{or}(t, t') \leq \mathbf{or}(u, u')$

Let $(P, \leq)$ be an poset (partially ordered set). A *directed set* in P is a subset D of P such that

$$\forall a, b \in D\ \exists c \in D\ [a \leq c\ \wedge\ b \leq c].$$

An *ideal* S is a directed set which is *downward closed*, i.e. such that

$$\forall a \in S\ [b \leq a \Rightarrow b \in S].$$

The set of ideals of P, ordered by set inclusion, we will denote by $(Id(P), \subseteq)$. It is well known that it is a Complete Partial Order (i.e. it has a minimum, and each non-empty set of elements admits a least upper bound) and it contains a sub-CPO isomorphic to $(P, \leq)$. $(Id(P), \subseteq)$ is called *completion by ideals* of $(P, \leq)$. The elements of the subset isomorphic to $(P, \leq)$ will be denoted by the corresponding elements of P.

Definition 6.3 (The domain of interpretation) *The complete partial order $(\mathcal{T}^\omega, \leq)$ (the domain of finite and infinite or trees) is the completion by ideals of the poset $(\mathcal{T}, \leq)$. The least upper bound of a directed subset $\mathcal{D} \in \mathcal{T}^\omega$ will be denoted by $\sqcup\mathcal{D}$.* □

Definition 6.4 (Operational Semantics) *The operational semantics of a program is a function* $\mathcal{O}$:$\{o,u\} \times Goal \to \mathcal{P}(\mathcal{T}^\omega)$, *where Goal is the set of all the goals and* $\mathcal{P}(\mathcal{T}^\omega)$ *is the set of all the subsets of* $\mathcal{T}^\omega$. $\mathcal{O}$ *is defined as follows.*

$$\mathcal{O}_\ell(P) = \{\sqcup_i \mathcal{O}'(P_i) \colon i \in \{0,1,2,\ldots\}, P \equiv P_0,\ \ell \vdash P_i \to^A_\lambda P_{i+1}\}$$

Note that the environment of the whole configuration is always empty. The argument ℓ *indicates the dependence of the operational semantics upon the global computation beeing considered ordered or unordered.* $\mathcal{O}'$:$Goal \to \mathcal{T}^\omega$ *is defined as follows:*

- $\mathcal{O}'(\mathbf{or}(Q,R)) = \mathbf{or}(\mathcal{O}'(Q), \mathcal{O}'(R))$.
- $\mathcal{O}'(\exists X.\mathbf{and}(\bar{A};\vartheta)) = \begin{cases} \mathbf{fail} & \text{if } \exists X.\mathbf{and}(\bar{A};\vartheta) \to^m_\lambda \mathbf{fail} \\ \mathbf{deadlock} & \text{if } \exists X.\mathbf{and}(\bar{A};\vartheta) \to^m_\lambda \mathbf{deadlock} \\ \exists X.\vartheta & \text{if } \bar{A} \equiv \lambda \\ \bot & \text{otherwise} \end{cases}$

Note that, if $P_0 \to^A_\lambda \ldots P_i \to^A_\lambda \ldots$, then (since P_i ranges over goals, hence it cannot be **fail** or **deadlock**), $\{\mathcal{O}'(P_i)\}_{i\geq 0}$ is a chain. Therefore (since a chain is a particular case of directed set), the definition of $\mathcal{O}$ is correct.

If the constraint system is decidable, then the rules for *failure* and *suspension* cover all the cases in which the computation rules are not applicable, excepting the or-boxes. Namely, a configuration is final only if it is of one of the following forms: **fail**, **deadlock**, an empty and-box, or an or-box containing only final configurations. This means that the semantics of a configuration is always a set maximal objects (possible infinite) in $\mathcal{T}^\omega$ (i.e., the leaves are not labeled by $\bot$).

6.1 Primitive Operations

The primitive operations are the only ones that can modify the environment. The four primitives we describe differ in the level of the environment they are allowed to impose constraint on. In particular, **ask** cannot impose any constraint (its constraint must be completely *entailed* by the environment), whilst $\mathbf{tell}_\omega$ can always do it. Between these two extreme cases, there are $\mathbf{tell}_0$ and $\mathbf{tell}_1$. The first can impose contraints on the local variables of the parent and box, whilst the second can impose contraints on the local variables of the "granparent" and box.

- $activ(\mathbf{ask}, X, \psi, \sigma, s) \equiv \models \hat{\Theta}_{s \diamond \exists X.\sigma} \supset \psi$
- $activ(\mathbf{tell}_0, X, \psi, \sigma, s) \equiv \models \hat{\Theta}_{s \diamond \exists X.\sigma} \supset \exists X.\sigma \wedge \psi$
- $activ(\mathbf{tell}_1, X, \psi, \sigma, \lambda) \equiv true$
- $activ(\mathbf{tell}_1, X, \psi, \sigma, s \diamond \exists Y.\vartheta) \equiv \models \hat{\Theta}_{s \diamond \exists Y.\vartheta \diamond \exists X.\sigma} \supset \exists Y \cup X.(\vartheta \wedge \psi \wedge \sigma)$
- $activ(\mathbf{tell}_\omega, X, \psi, \sigma, s) \equiv true$

7 Related Work

Vijay Saraswat defined a language CP [Sar87] that provides among other things deep guards, an operator called "don't know commit", related to our "wait" operator, and the concept of blocks, which are similar to and-boxes. One of the main differences between CP and our work is our control of *when* to promote nondeterminism. This is also true of Saraswat's thesis [Sar89a]. Also, we emphasize fully interleaved execution in a language with deep guards. However, Kernel Andorra Prolog is definitely a concurrent constraint language.

8 Discussion

We presented a formal transformational semantics for Kernel Andorra Prolog. The semantics is transformational since it desribes the final results of a KAP computation (as a set of abstract trees), and there is no notion of interaction with the environment. A number of issues will be addressed in the near future. Firstly, since KAP is intended as a framework for implementing some user oriented languages, by using the proper constraint operations, we like to prove the correctness of the implementation w.r.t a priori given semantics of the user language.

Another issue is proving the properties of the sublanguages given in [HJ90]. This is partly done in [Fra90].

The semantics given treats and-boxes seen at the outermost level as black boxes until the box either succeeds, fails or suspends. This notion is not sufficient, for some of the sublanguages of KAP. In particular the Reactive Andorra Prolog which encapsulates nondeterminism in pruning guards, needs a more refined notion where actions in an and-box can affect its environment, i.e. some sort of a reactive semantics, and observational equivalence based on it.

Finally we remind the reader, that other more important issues, like efficient implementation, of both sequential and parallel machines, and programming methodology and techniques have the highest priority and much effort is devoted to them.

Acknowledgements

The authors wish to thank Sverker Janson, Vijay Saraswat, Torkel Franzén, D.H.D. Warren and Bill Kornfeld for many valuable comments and suggestions. This work is part ofthe PEPMA ESPRIT Project (P2471), and is supported by the Swedish Board of Technical Development, Televerket, and Ericsson.

References

[BG89] R. Bahgat and S. Gregory. Pandora: Non-deterministic Parallel Logic Programming. In Giorgio Levi and Maurizio Martelli, editors, *Proc. of the Sixth International Conference on Logic Programming*, Series in Logic Programming, pages 471–486, Lisboa, 1989. The MIT Press.

[Fra90] T. Franzén. Formal aspects of Kernel Andorra Prolog. Technical Report R90008, SICS, Sweden, 1990.

[HB88] S. Haridi and P. Brand. Andorra Prolog:, an integration of Prolog and commited choice languages. In *Proc. of the International Conference on Fifth Generation Computer Systems*, pages 745–754, Tokyo, 1988. Institute for New Generation Computer Technology (ICOT).

[HJ90] S. Haridi and S. Janson. Kernel Andorra Prolog and its computation model. In *Proc. of the Seventh International Conference on Logic Programming*, 1990.

[HJ91] S. Haridi and S. Janson. Programming paradigms of the Andorra Kernel Language. Technical report, SICS, Sweden, 1991.

[Kor89] W. Kornfeld. Constraint programming in Andorra Prolog. Presented at the Swedish-Japanese-Italian workshop, 1989.

[Sar87] V.A. Saraswat. The concurrent logic programming language CP: definition and operational semantics. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49–63. ACM, New York, 1987.

[Sar89a] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, january 1989. Published by The MIT Press, U.S.A., 1990.

[Sar89b] V.A. Saraswat. Programming in Andorra Prolog. Technical report, Xerox PARC, 1989.

[War87] D. H. D. Warren. The Andorra principle. Presented at the Gigalips workshop, Stockholm, 1987.

[Yan89] R. Yang. Solving simple substitution ciphers in Andorra-I. In Giorgio Levi and Maurizio Martelli, editors, *Proc. of the Sixth International Conference on Logic Programming*, Lisboa, 1989. The MIT Press.

Customization of First-Class Tuple-Spaces in a Higher-Order Language

Suresh Jagannathan
NEC Research Institute
Princeton, NJ 08540
suresh@research.nec.com

Abstract

A distributed data structure is an object which permits many producers to augment or modify its contents, and many consumers simultaneously to access its component elements. Synchronization is implicit in data structure access: a process that requests an element which has not yet been generated blocks until a producer creates it.

In this paper, we describe a parallel programming language (called $\mathcal{TS}$) whose fundamental communication device is a significant generalization of the tuple-space distributed data structure found in the Linda coordination language[6]. Our sequential base language is a dialect of Scheme[19].

Beyond the fact that $\mathcal{TS}$ is derived by incorporating a tuple-space coordination language into a higher-order computation language (*i.e.*, Scheme), $\mathcal{TS}$ differs from other tuple-space languages in two important ways:

- Tuple-spaces are first-class objects. They may be dynamically created, bound to names, passed as arguments to (or returned as results from) functions, and built into other data structures or tuples.
- The behavior of tuple-spaces may be customized. A tuple-space are manipulated via a *policy closure* that specify its operational characteristics. The representation of policy closures take significant advantage of Scheme's support for higher-order functions; there is no fundamental extension to Scheme needed in order to support them.

We argue that first-class customizable tuple-spaces provide an expressive and flexible medium for building parallel programs in higher-order languages.

1 Introduction

Distributed data structures are widely recognized to be an important device in structuring explicitly parallel programs. A distributed data structure is valuable because it abstracts low-level details about process synchronization and communication to high-level algorithmic design issues involving data structure access and generation. Generally speaking, the semantics of such structures permit many producers to augment or modify its contents, and many consumers simultaneously to access its component elements. Synchronization is implicit in data structure access: a process that requests an element which has not yet been generated blocks until a producer creates it. Some notable examples of distributed data structures are the blackboard object in Shared Prolog[2], stream abstractions in Flat Concurrent Prolog[20], Concurrent Smalltalk's distributed objects[13] and its closely related variant Concurrent Aggregates[7], the I-structure in Id[4] and C.Linda's flat tuple-space[6].

In this paper, we describe a parallel programming language (called $\mathcal{TS}$) whose fundamental communication device is a significant generalization of the tuple-space distributed data structure found in the Linda coordination language. Our sequential base language is a dialect of Scheme[19].

By way of introduction, a $\mathcal{TS}$ tuple-space defines a shared associative memory whose elements are ordered sets of values or processes known as *tuples*. Fields in passive tuples, *i.e.,* tuples containing only values, can be retrieved using a simple pattern matching procedure. In the default case, failure to produce a substitution causes the process that initiated the match operation to block. A blocked process may resume only after a matching tuple is deposited into the specified tuple-space. An active tuple, *i.e.,* a tuple containing concurrently executing processes, turns into a passive one when each process completes and returns a value. Active tuples are not involved in the matching procedure. [5] discusses how distributed data structures found in other programming models may be implemented using tuple-spaces. [15] discusses compile-time analysis techniques for first-class tuple-spaces in higher-order languages.

Beyond the fact that $\mathcal{TS}$ is derived by incorporating a tuple-space coordination language into a higher-order computation language (Scheme), $\mathcal{TS}$ differs from other tuple-space languages in two important ways:

1. Tuple-spaces are first-class objects. They may be dynamically created, bound to names, passed as arguments to (or returned as results from) functions, and built into other data structures or tuples.

2. The behavior of tuple-spaces may be customized. Tuple-spaces are accessed via a set of policy procedures that define its various attributes.

These attributes include (a) access to the tuple-space that is to be manipulated, (b) the matching protocol that governs tuple retrieval, (c) the blocking policy that specifies the conditions under which processes accessing this tuple-space block, and (d) the failure policy that specifies conditions under which a tuple-space operation fails.

There are two classes of tuple-space operations – those that store into a tuple-space, and those that read from it. In either case, tuple-spaces are manipulated via a set of policy definitions encapsulated within a closure that is a mandatory argument to the operation. Thus, an expression of the form, (put P *tuple*), evaluates P to get a policy closure and uses the definitions found in this closure to determine if and where *tuple* is to be deposited.

Policy definitions are defined in terms of ordinary Scheme procedures. Their utility derives from the fact that they can be used to transform a tuple-space into a distributed data structure analogue of a sequential data abstraction – the representation of a tuple-space structure can be manipulated and its behavior may be tailored to conform to (or take advantage of) particular operational requirements. Tuple-spaces are now properly regarded as objects whose behavior is determined by the definition of its policy procedures.

The paper is organized as follows. The next section gives motivation for the design of the language; Section 3 gives an overview of the language, and describes the specification of tuple-space operations. Section 4 describes a number of examples whose formulation is significantly simplified by the presence of first-class customizable tuple-spaces.

2 Motivation

The motivation for the design of $\mathcal{TS}$ is two-fold. First, we are interested in building highly-modular parallel systems for symbolic computation. First-class tuple-spaces offer themselves as an expressive modularity device in this regard.

Second, we wish to apply abstraction, customization and parameterization techniques commonly used in the specification and implementation of sequential data structures to their distributed data structure counterparts. Our contention is that these techniques can reduce complexity, and enhance expressivity. We elaborate on these two points below.

2.1 Modularity

There has been much recent interest on incorporating explicit concurrency into expression-based programming languages. Mul-T[17] and MultiLisp[12] are lan-

guages that augment Scheme with a *future* construct; Concurrent Prolog[21] and Parlog[8] are logic programming languages that are based on an asynchronous process interpretation of clause evaluation. Concurrent Smalltalk[13] and Actors[1] are representative examples of programming languages that extend object-based programming with parallel facilities.

$\mathcal{TS}$ is distinguished from these other efforts in several important respects. By way of comparison, synchronization in concurrent logic programming and other parallel Lisp dialects takes place through low-level primitives (*e.g.*, futures and semaphores in Mul-T) and shared variables (*e.g.*, read-only variables in Concurrent Prolog). Process communication in $\mathcal{TS}$, on the other hand, is decoupled from process instantiation. This attribute makes it possible for processes to initiate requests for the value of objects even if the object itself has not yet been created, and to collectively contribute to the construction of shared objects. These capabilities are absent in many other distributed data structure-based systems (*e.g.*, parallel Lisps[17] or object-based concurrent systems[13]), and have no trivial formulation in a concurrent logic framework[21].

Giving first-class status to tuple-space objects also encourages greater modularity and simplifies implementation[10, 15]. By permitting tuple-spaces to be denoted, we allow the programmer to partition the communication medium as he sees fit. Conventional namespace management techniques available in the base language can be easily applied over tuple-spaces as well. To encapsulate a set of related shared data objects, we deposit them within a single tuple-space; this tuple-space can be made accessible only to those processes that require access to these objects. A set of related processes can also reside within their own tuple-space; data values which they share can be deposited and retrieved within this structure. Other processes need not be aware of these objects, and ad hoc naming conventions need not be applied to ensure that data objects are not mistakenly retrieved by processes which do not need them.

2.2 Customizability

The semantics of other tuple-space languages restrict the ways in which tuple-spaces can be manipulated. In general, the only operations permitted on a tuple-space are those which deposit, read, remove, or test the presence/absence of tuples. Customizing matching protocols, determining where tuples should be deposited based on conditions known only at runtime, specifying the constraints (outside of a match failure) under which processes should block, or specifying constraints under which tuple-space operations may *fail* but not block, are capabilities not available in these languages.

One immediate consequence of preventing user specification and customization

of the operational behavior of tuple-spaces is reduced flexibility: while it is easy to implement data structures whose semantics can be naturally expressed using just `get` (remove tuple), `rd` (read tuple), `put` (deposit tuple) and `spawn` (fork process) operations[1], it becomes problematic to implement structures that don't fit neatly into this framework or which require additional constraints not expressible in terms of these operations. Conceptually, a tuple-space is a data abstraction whose representation is manipulated via these operators; the inability of programmers to customize their behavior limits the ease with which different kinds of distributed data structures can be specified.

For example, given a C-Linda version of a distributed object O, one can trivially deposit a method or instance variable x by writing:

```
put("O","x",x)
```

(`"O"` is a label that tags all operations associated with O, and `put` deposits its tuple argument into a global tuple-space.). To send a message M to O one writes:

```
rd("O","M",?v)
```

"?v" introduces an unbound variable or *formal*; the variable is assigned the value of the third field in the tuple to which this tuple template is matched. If v is bound to a method represented as a procedure, one can subsequently apply this procedure to arguments. Tuple-spaces in this simple framework are distributed analogues of a conventional record containing procedure and scalar-valued fields.

Suppose, however, we wish to have these distributed objects adhere to an inheritance protocol[11]: O is to automatically dispatch messages to another object (its parent) if it does not define a field whose value is requested by the message. In other words, a tuple-space is now to be regarded as the representation of a class instance; and inheritance is realized by directing tuple-space operations to particular tuple-spaces. Since O defines a distributed version of a sequential object, we must also ensure that no `get` operations are permitted on its components; all instance variable and method bindings are intended to be immutable, although these bindings may be bound to mutable locations.

Satisfying these constraints is non-trivial within the standard Linda framework. To delegate operations in the manner suggested requires performing an explicit test to determine if a tuple containing the requested operation exists within the specified tuple-space; if it does not, the same test must be performed on its "parent"; such tests proceed along a tuple-space hierarchy until a matching tuple is found. If O has n ancestors, messages sent to O must explicitly include

[1] The names given to these operations in $\mathcal{TS}$ differ from those used in *e.g.*, C-Linda because the semantics of these operators differ from their counterparts in the Linda framework in several significant ways.

n conditional tests for each of these ancestors; operations on O's siblings and children must be structured similarly. The validity of `get` operations must also be explicitly monitored. Changes in the inheritance structure require global changes to the access operations on affected objects. Modularity is compromised and complexity increased.

$\mathcal{TS}$ provides an alternative solution to this problem. The conditions under which a tuple-space operation is to block or fail may be defined explicitly in terms of a policy procedure. Similarly, the particular semantics of matching or the tuple-space repository in which tuples are to be deposited or removed may all be specified in terms of ordinary Scheme procedures that operate over tuple-spaces.

Since tuple-space operations manipulate policy closures directly, processes that manipulate tuple-spaces need not be aware of their implementation, and no explicit bookkeeping information need be maintained to enforce correctness of its specification. The conceptual and implementation overhead of encapsulating tuple-space operations within user-defined procedures and applying these procedures as needed is absent; all tuple-space operations have a uniform syntax and self-consistent behavior.

We emphasize that policy definitions are *not* orthogonal to the semantics or implementation of tuple-spaces in any significant sense. Any specification of a tuple-space abstraction must contend with issues dealing with tuple access, matching, failure, and blocking. The semantics of tuple-spaces discussed in this paper makes these issues manifest to the $\mathcal{TS}$ programmer.

3 The Language

$\mathcal{TS}$ is a set!- and call/cc-free dialect of Scheme[19] with first-class tuple-spaces. We concentrate in this section on the coordination subset of $\mathcal{TS}$; this subset defines operations for creating and manipulating tuple-spaces. The interaction of tuple-spaces on other Scheme constructs is considered in Section 4.1.

3.1 Creating Tuple-Spaces

The coordination language upon which $\mathcal{TS}$ is based treats tuple-spaces as denotable objects with a distinguished type. To create a tuple-space, we evaluate `(make-ts)`. The value returned is a reference to a newly created tuple-space object. More precisely, this object returns a *representation* structure of a tuple-space as determined by suitable compile-time analysis[15]. In the abstract, a

tuple-space, like any other Scheme structure, is accessed via pointer, not value; like any other Scheme object, a tuple-space may be garbage-collected if there remain no extant references to it.

Operations on a tuple-space T access it indirectly via a policy closure. This closure determines T's operational attributes; in particular, the policy closure encapsulates operations that specify the constraints under which operations block or fail, the protocol used to retrieve tuples are retrieved, and the mechanism by which tuples are to be retrieved.

A policy closure is a procedure of one argument. This argument determines the policy definition to be evaluated. Thus, a policy closure takes the form:

```
(lambda (op)
   (cond ((eq? op policy definition)
          (lambda (args) body)
             ⋮
          (else op))))
```

Tuple-space operations in $\mathcal{TS}$ make use of four such definitions:

1. The *access* policy specifies the tuple-space to be operated upon. Tuple-space operations use this policy to determine the tuple-space of interest. Since tuple-spaces are first-class, the same access policy can yield different tuple-spaces on different calls.

2. The *matching* policy specifies the set of tuples a tuple-template can match against. Match operations are initiated only by processes wishing to read or remove a tuple from a tuple-space.

3. The *blocking* policy specifies the conditions under which a process may not have access to this tuple-space.

4. The *failure* policy specifies the conditions under which a tuple-space operation returns `fail`. Failure does not imply blocking; the failure policy is used to realize non-blocking exceptional conditions on tuple-space access.

Each of these policy definitions have a default implementation defined by the system; these definitions are bound to the names: `:access`, `:match`, `:block` and `:fail` respectively. These default definitions are presumed global.

If T's policy closure is structured thus:

```
(lambda (op)
   (cond ((eq? op :match)
          (lambda (args) match policy))
         (else op)))
```

and is bound to T-`policy`, then evaluating: ((*T*-`policy :match`) *arguments*) applies (`lambda` (*args*) *match policy*) to *arguments*. In this example, T's policy closure does not define definitions for `:access`, `:fail` or `:block`; thus, the expression (*T*-`policy :fail`) returns the default failure policy.

Policy definitions all take three arguments:

1. `PC` – the policy closure of the current tuple-space.
2. `tuple` – the tuple argument being manipulated. We describe the policy operations allowable over tuples below.
3. `kind` – a symbol drawn from the set,

   `{ read, store, remove, fork }`

   that indicates the type of operation being performed. Thus, `put` operations invoke a policy closure with kind `store`; `spawn` operations invoke a policy closure with kind `fork`; `get` operations have kind `remove`; `rd` operations have kind `read`.

3.1.1 Concrete Representations

Compile-time analysis of $\mathcal{TS}$ programs is used to determine efficient representations for tuples and tuple-space objects; this representation is derived by examining the structure and type of tuple components[15]. For example, a tuple-space T closed under a policy closure P_T that contains tuples of length two and whose `rd` and `in` operations are all of the form: (*op* P_T `(i,?v)`) for some integer-yielding expression i can be implemented in terms of a concurrent vector data structure. A tuple denotes an $<$ *index,value* $>$ pair in this vector; attempts to read an "empty" vector elements results in the process blocking; and processes that wish to update a vector element must gain exclusive access to it first. $\mathcal{TS}$ programmers can determine the representation type associated with any tuple-space using operators described below. Insofar as the representation structures generated by the implementation are visible to the program, $\mathcal{TS}$ bears resemblance to "reflective" languages[22, 25]. The underlying behavior of tuple-space operations, and the structures they manipulate can be examined and customized by the programmer.

Let R_T be a representation of a tuple-space T, and let R_t be a representation of a tuple within T. (The representation type of a tuple t is derived by evaluating `:rep_t` t). All concrete representations of tuples provide a set of operations for

accessing various components; we introduce some of these operators during the course of the paper.

The representation object of a tuple-space must provide the following operations; given a tuple-space T, its representation is accessed by evaluating `(:rep` T`)`:

- (`:type` R_T) returns R_T's type; policy definitions dispatch on this type to determine the index and selection operations they should perform.
- (`:size` R_T) returns the number of elements in R_T.
- (`:empty?` R_T) returns true iff R_T contains no tuples.
- (`:write` R_T R_t) stores R_t into R_T.
- (`:with-lock` R_T `kind` *body*) grabs a suitable read or write lock on R_T (depending upon the value of `kind`) and evaluates *body*; the lock is released after *body* completes. Policy definitions manipulate read and write locks to ensure atomicity and serialization constraints.

In general, users needing only standard Linda functionality need not be concerned about manipulating tuple-space representations; the default policy definitions for tuple-spaces implement the necessary constraints. (See figure 1.)

The policies obeyed by a tuple-space created by evaluating `(default-ts)` are the same as, *e.g.*, C-Linda's global tuple-space. The behavior of the default `:match` policy is determined from the representation type of the argument tuple-space. The value returned is a list; the elements in this list are extracted from the fields of a resident tuple such that the i^{th} element in this element is to be bound to the i^{th} formal in the argument tuple. If there are several candidate tuples that satisfy the match requirements, one is chosen non-deterministically. The representation of a tuple-space is responsible for ensuring proper serialization; thus, two `get` operations that occur simultaneously on the same tuple-space must not result in the same tuple being matched for both operations.

The constraint imposed by the default block policy is dictated by the success of `:match`. If a `rd` or `get` operation is performed on a tuple-space such that the `:match` policy is unable to produce a matching tuple, `:block` returns `#block`. In all other instances, the policy procedure succeeds. The default `:access` policy returns a global tuple-space object. The `:fail` policy always succeeds in the default implementation – tuple-space operations either succeed or result in a process becoming blocked; there is no "failure without blocking" semantics in this implementation.

When `default-ts` is applied, it creates a new tuple-space using `make-ts`; the access policy associated with the policy closure returned yields this tuple-space.

```
(define (:match PC tuple kind)
   (let ((TS_rep (:rep ((PC :access) PC tuple kind)))
         (tuple_rep (:rep_t tuple)))
     (case (:type TS_rep)
        ((semaphore)   (read_sem TS_rep kind))
        ((queue)       (head TS_rep kind))
            :
            :)))

(define (:block PC tuple kind)
   (if (memq kind '(read remove))
       (let ((matches ((PC :match) PC tuple kind)))
          (if (null matches) #block matches))
       #succeed))

(define (:fail PC tuple kind)
    #succeed)

(define (:access PC tuple kind)
    (global-TS))

(define (default-ts)
   (let ((newTS (make-TS)))
     (lambda (op)
        (cond ((eq? op :access)
               (lambda (PC tuple kind) newTS))
              (else op)))))
```

Figure 1: One possible implementation of default policy definitions that conform to Linda-style semantics.

3.2 Depositing Tuples

A tuple is deposited into a tuple-space using one of two operators. Let `PC` be an expression that yields a policy closure; then, the expression:

`(put PC (`e_1`, `e_2`, ..., `e_n`))`

is evaluated as follows:

1. Evaluate `PC` to yield a policy closure P.
2. Evaluate each of the e_i in the current evaluation environment to get a value v_i. Let the tuple constructed by replacing each e_i by v_i be called t. Our notion of "value" is consistent with Scheme's[19] – constants and symbols are values as are references to closures, lists, tuple-spaces, and vectors.
3. Evaluate P's blocking policy definition:

   `((`P `:block)` P t `'store)`
4. If the result of this application is `#block`, block the current process. Otherwise, evaluate P's failure policy:

   `((`P `:fail)` P t `'store))`

   If the result of this application is `#fail`, return `#fail`. Otherwise, evaluate

   `(let ((`R_T `((:rep ((`P `:access)` P t `'store)))))`
   `(:with-lock` R_T `'store (:write` R_T `(:rep_t` R_t`)))`

   and resume all processes blocked on this tuple. (`:rep_t` returns the tuple representation of its argument.)

`Spawn` is the *non-strict* counterpart of `put`. If `PC` is a policy closure, the expression

`(spawn PC (`e_1 e_2 ... e_n`))`

concurrently instantiates n *processes*; the i^{th} process in this ensemble is responsible for computing the value of e_i. Since processes cannot communicate via shared variables they may freely access common binding environments. When all processes complete, a passive tuple containing the values they have generated is deposited into the appropriate tuple-space by evaluating a `put` operation using the newly constructed passive tuple as the tuple argument.

3.3 Retrieving Tuples

`Put` and `spawn` generate tuples into tuple-space. `Rd` and `get` read tuples and binding-values from tuple-space. A `rd` expression takes the form:

`(rd PC (`e_1`, `e_2`, ..., `e_n`)` *body*`)`

Unlike tuple-generator expressions, each of the e_i may be either an ordinary $\mathcal{TS}$ expression *or* a formal. The evaluation of the above expression takes place as follows:

1. Evaluate `PC` to yield a policy closure P.
2. Evaluate each e_i that is *not* a formal in the current evaluation environment to get a value v_i. Let the tuple constructed by replacing each e_i by v_i be called t.
3. Bind each formal to `#undefined`.
4. Evaluate P's blocking policy definition:

   `((`P `:block)` P t `'read)`
5. If the result of this application is `#block`, block the current process; the representation type of any tuple-space must provide a queue to hold blocked processes. Otherwise, bind the object returned by the application to V and evaluate P's failure policy:

   `((`P `:fail)` P t `'read)`

   If the result of this application is `#fail`, return `#fail`. Otherwise, bind the i^{th} element in V to the i^{th} formal in t, and return the value yielded by evaluating *body* in this augmented environment.

The `get` operation is semantically identical to `rd` except that the tuple chosen for the match is removed from the given tuple-space.

4 Examples

We consider several simple examples to illustrate the utility of the language. The first is the implementation of a cell abstraction; the second is a specification of a dynamic load balancing system; the third is a sketch of an inheritance system using $\mathcal{TS}$ tuple-spaces as concurrent objects.

While each structure is useful in different domains, they are related to one another insofar as they impose conditions on their use not readily expressible using ordinary tuple-space operations; we argue that first-class tuple-spaces and customization improves the clarity of their definition.

4.1 Cells

$\mathcal{TS}$ contains no assignment operations on variables. Given that the constituent elements of a tuple-space are shared $\mathcal{TS}$ objects, arbitrary assignment operations on $\mathcal{TS}$ objects would permit "backdoor" inter-process communication through shared variables. Such a capability would violate the immutability

constraint on tuples, and make it impossible to enforce mutual exclusion or atomicity constraints.

Nonetheless, despite the absence of explicit `set!` operations, the fact that data objects can be added to and removed from a tuple-space makes it straightforward to implement applications that require object mutation. Consider the implementation of a memory cell shown in figure 2.

A cell will always contain at most one element. A write operation on a cell uses the failure policy to remove an old element before writing a new one. Two processes attempting to write into the cell at the same time are serialized by the definition of the failure policy and `put`. The policy closure returned by `make-cell` is closed over a newly created tuple-space, `newTS`; the closure's access policy returns this tuple-space when applied. Compile-time analysis on the cell abstraction would reveal that the tuple-space holding the current state does not escape its lexical contour and only holds tuples of length one of string type. This tuple-space can be thus represented as an ordinary protected shared variable.

All side-effecting operations in Scheme are reimplemented in $\mathcal{TS}$ to check that the object to be mutated is indeed a cell. For example, to create a mutable vector of n arguments, we write:

```
(make-vector 10 (make-cell))
```

The operation:

```
(vector-set! vector k v)
```

is equivalent to:

```
(let ((obj (vector-ref v k)))
   (if (cell? obj)
       (write-cell obj v)
       error))
```

Cells elevate $\mathcal{TS}$ to a middle ground between purely functional and completely constructive languages. Pure functional languages (*e.g.*, Haskell[14]) eliminate any notion of global state; completely constructive languages (*e.g.*, Ada[24]) use state variables pervasively. Both language designs have significant implications in the context of concurrency. Functional languages prevent users from expressing non-deterministic or explicitly parallel programs; statement-based parallel languages often require bulky modularity devices (*e.g.*, monitors or rendezvous points) to ensure atomicity constraints. $\mathcal{TS}$, on the other hand, purports to serve as a bridge between these two proposals. The functional core of $\mathcal{TS}$ makes it straightforward to program in a mostly side-effect free style; nonetheless, the modularity and serialization properties of tuple-space make it easy to build to local mutable objects when required.

```
(define (make-cell)
  (let* ((newTS (make-ts))
         (state (default-ts)))
    (put state ("empty"))
    (cons 'cell
          (lambda (op)
              (cond ((eq? op :fail)
                     (lambda (PC tuple kind)
                        (let ((ts (:rep ((PC :access)
                                         PC tuple kind))))
                           (get state (?condition)
                                (cond ((and (equal? condition "empty")
                                            (memq kind '(store fork))))
                                      (put state ("full"))
                                      #succeed)
                                     ((memq kind '(read remove))
                                      #succeed)
                                     (else #fail))))))
                    ((eq? op :access)
                     (lambda (PC tuple kind) newTS))
                    (else op))))))


(define (read-cell cell)
   (rd (second cell) (?v) v))


(define (write-cell cell v)
   (let ((contents (second cell)))
     (if (fail? (put contents (v)))
         (get contents (?old-v)
              (write-cell cell v))))


(define (cell? cell)
  (equal? (first cell) 'cell))
```

Figure 2: A $\mathcal{TS}$ implementation of a memory cell.

4.2 Dynamic Load Balancing

Besides allowing the implementation of different kinds of distributed data structures, customizable first-class tuple-space also permit the $\mathcal{TS}$ programmer to manipulate the process state of tuple-spaces. For example, consider a collection of tuple-spaces that represent virtual processors; each tuple-space contains a number of active processes and data elements. These processes communicate with one another via standard tuple-operations. Since processes are periodically run within a virtual processor, it is useful to allow process load on any given processor to be monitored and balanced dynamically, based on runtime conditions.

We present in figure 3 the structure of an abstraction that views virtual processors. The processors coordinate their activities by monitoring their load (*i.e.*, the number of extant processes they contain), and by handing incoming processes to less loaded processors when they become saturated.

In the implementation described here, `dynamic_load` returns a list of policy closures; the i^{th} closure corresponds to the i^{th} virtual processor in the processor ensemble. Each policy closure is closed over its index this ensemble. To permit dynamic load balancing, we specify an access policy that examines the current load of its virtual processor. If the policy is invoked by a `spawn` operation (which is the only device for instantiating new processes), and the current load on this processor has already reached its maximum, we apply the policy closure of its neighbor with the current arguments. In the case where the load has not reached its maximum, we simply increase the load factor by the number of processes being instantiated, and return the current tuple-space. When the processes instantiated by a `spawn` operation complete, the resulting passive tuple is recorded in the specified tuple-space via a `put` operation. Since such an operation signals the completion of a related set of processes, the load value of the virtual processor in which the passive tuple is to be deposited is decremented appropriately. (Of course, this implementation assumes that processes do not generate spurious `put` operations on these virtual processors.) In all other cases, the current tuple-space is returned with no other processing initiated.

Dynamic load balancing in the absence of policy procedures is possible by abstracting access policy decisions into user defined procedures that mediate all tuple-space operations into the processor set. Rather than writing expressions of the form:

`(spawn (nth proc_set i)` *tuple*`)`

we now need to invoke a dedicated procedure to determine the suitable tuple-space. Abstraction is compromised – `spawn` operations over ordinary tuple-spaces are distinguished from operations on the virtual processor ensemble;

```
(define (dynamic_load n)
   (let ((load-vector (make-vector n (make-cell))))
      (vector-fill! load-vector default-load)
      (let iterate ((i 0)
                    (proc_set '()))
         (cond ((= i n) proc_set)
               (else (iterate (+ i 1)
                              (cons proc_set
                                 (load-policy (make-ts) proc_set
                                              load-vector i))))))))


(define (load-policy newTS proc_set load-vector i)
   (lambda (op)
      (cond ((eq? op :access)
             (lambda (PC tuple kind)
                (cond ((and (equal? kind 'fork)
                            (> (vref load-vector i)
                               *max-load*))
                       (let ((next-PC (nth proc_set
                                           (modulo (1+ i) n))))
                         ((next-PC :access)
                          next-PC tuple kind)))
                      ((equal? kind 'fork)
                       (increase! load-vector i
                                  (:tuple_length (:rep_t tuple)))
                       newTS)
                      ((equal? kind 'put)
                       (decrease! load-vector i
                                  (:tuple_length (:rep_t tuple)))
                       newTS)
                      (else newTS))))
            (else op)))))
```

Figure 3: A dynamic load balancing abstraction using a customized access policy definition.

different implementations of a load balancing policy entail different procedure interfaces. Policy closures present a transparent and seamless interface to tuple-spaces despite the fact that the behavior of tuple-space operations may greatly vary across distinct tuple-space objects.

4.3 Inheritance

The technique used to build a dynamic load balancing system using customized policy procedures can be applied in a very different domain with only slight modification. Consider the implementation of a Smalltalk-style[11] inheritance-based system. The basic entities in such system are objects that retain local state, and which communicate via message-passing. Moreover, objects are free to dispatch messages they receive to their parents in the object hierarchy if they determine that they cannot interpret them. A concurrent message-passing system permits objects to process many messages simultaneously; message sends are asynchronous, and objects may respond to many messages concurrently. The approach described here is simple, omitting many important details and disregarding any syntactic sugaring, but it captures the essence of inheritance-based programming relying only on policy definitions and first-class tuple-spaces to do so.

The ability to customize the behavior of tuple-spaces in $\mathcal{TS}$ allows a simple formulation of inheritance within a parallel system. The basic idea is to implement objects as tuple-spaces, and message passing as tuple reads. Methods and instance variables are stored as tuples. If a tuple-read operation is performed on a tuple-space T that contains no matching tuple, the operation is redirected using the `T`'s access policy definition to T parent in the object hierarchy. In other words, a tuple-template for whom no match exists in one tuple-space is *implicitly* sent to another tuple-space to be resolved. Implicit redirection of messages is the operational intuition underlying inheritance. Customization of tuple-spaces gives $\mathcal{TS}$ this capability.

Object methods are implemented as abstractions whose first argument plays the role of Smalltalk's `self` pseudo-variable – `self` yields the object containing the instance variables to be used by a method. Thus, a method definition takes the form:

```
(lambda (self . args)
    definition)
```

Objects are represented as policy closures. To invoke method M in object O with self argument `self` and arguments *args*, we write:

```
(define (make-object parent)
   (let ((newTS (make-ts))
         (objectList (make-cell))
         (initialSet '()))
     (lambda (op)
        (cond ((eq? op :access)
               (lambda (PC tuple kind)
                  (let* ((tuple_rep (:rep_t tuple))
                         (component (:tuple_index tuple_rep 1)))
                    (cond ((equal? kind 'store)
                           (write-cell objectList
                                       (cons component initialSet))
                           newTS)
                          ((equal? kind 'read)
                           (if (memq component (read-cell objectList))
                               newTS
                               ((parent :access) parent tuple kind)))
                          (else newTS)))))
              ((eq? op :fail)
               (lambda (PC tuple kind)
                  (if (memq kind '(remove spawn))
                      #fail
                      #succeed)))))))


(define (send obj name)
  (rd obj (name ?definition)
      definition))
```

Figure 4: Implementation of a simple object in $\mathcal{TS}$, and a `send` operation that returns the definition of its argument, `name` as defined in object `obj`'s hierarchy.

```
(rd O ('M ?v)
    (v self args))
```

Method definitions and instance variables are recorded within a tuple-space as two-tuples: $<$ ısymbol, ıdefinition $>$.

Thus, to augment object O with a new method definition M, we write:

```
(put O ('M definition))
```

An object O is created by applying `make-object`. The policy closure which it returns defines customized access and failure policies. O is closed over a tuple-space T that contains methods and instance variables. These definitions are

```
(define (make-point a b)
   (let ((obj (make-object (global-ts))))
      (put obj ('x a))
      (put obj ('y b))
      (put obj
         ('DistfromOrig (lambda (self)
                          (sqrt (+ (sqr (send self 'x))
                                   (sqr (send self 'y)))))))
      (put obj
         ('ClosertoOrig (lambda (self p)
                          (< ((send self 'DistfromOrig) self)
                             ((send p 'DistfromOrig) self)))))
      obj))
```

Figure 5: A `point` generator.

recorded using `put` operations. When a put operation on O is evaluated, the method or instance variable name is recorded in a list (`objectList`); this list is referenced whenever a process attempts to send a message to this object. When the definition of a method or instance variable is required (via a `rd` operation), the access policy tests whether the symbolic reference to this method is defined within `objectList`; if it is, the tuple-space is returned and the default matching protocol is guaranteed to find a tuple containing a definition for the requested operation. If no such component exists in the current tuple-space, the access policy evaluates the object's parent; the tuple-space yielded by evaluation of the parent is the tuple-space on which the message is to be sent (*i.e.*, the tuple-space on which the `rd` operation is to be performed).

The failure policy prevents object components from being removed or initiating processes within objects. Mutable components must be realized by binding the component name to a cell and performing the necessary changes on the cell.

To illustrate how we might use customized tuple-spaces to build inheritance systems, we consider an example discussed in [9, 16]. A `circle` is a sub-class of a `point`. `Point` defines instance variables `x` and `y` to specify its location. It also defines two methods: `DistfromOrig` computes the distance of a point from its origin and `ClosertoOrig` is a predicate which given another point object as its argument returns true if its point is closer to the origin than its argument, and false otherwise. We use `self` to denote the object whose tuple contents define the environment within which an expression should be evaluated. The code for a `point` generator is given in figure 5.

A circle is defined in terms of points. Because circles have a radius, they have a different meaning of distance from the origin. The notion of distance from

```
(define (make-circle a b r)
   (let ((super (make-point a b))
         (obj (make-object super)))
      (put obj ('radius r))
      (put obj
         ('DistfromOrig (lambda (self)
                          (max (- ((send super 'DistfromOrig) self)
                                  (send self 'radius)) 0))))
       obj))
```

Figure 6: A `circle` generator.

origin for circles is given in terms of the definition of `DistfromOrig` found in point objects: if l is the distance from the origin to the circle's center and r is the circle's radius, then $l - r$ gives the distance from the origin of the circle object. If this difference is negative, the distance is assumed to be 0. Thus, the meaning of `DistfromOrig` in a `circle` instance should refer to its meaning as specified by the `circle` (*not* the `point`) generator. The code for the `circle` generator is given in figure 6.

To create a circle C at coordinates `(3,4)` with radius `10`, we evaluate:

```
(define C (make-circle 3 4 10))
```

To determine C's distance from the origin, we evaluate:

```
((send C 'DistfromOrig) C)
```

Because message sends are implemented via tuple space, many processes may simultaneously send messages to the same object; conversely, many processes may simultaneously respond to received messages. This property distinguishes $\mathcal{TS}$ from several other concurrent object systems[3, 26] that serialize hierarchical abstractions. In these systems, objects can process only one message at a time; in contrast, objects in $\mathcal{TS}$ exhibit totally asynchronous behavior since they are implemented in terms of general tuple-space objects.

Furthermore, new method definitions and instance variables can be added to an object dynamically without requiring reevaluation of the inheritance tree – since object components are tuples, and there is no restriction in an object's specification that limits the addition of new tuples, processes are free to augment an object's component set even after the instance is defined. The ability to add new components to instances of objects is tantamount to a delegation-based system[18, 23]. Different instances of the same class can have different behavior based on how new instances are added and old instances are changed.

Single inheritance is realized as a simple application of tuple-space customization. Generalizing this method to handle multiple inheritance is straightforward, involving only slight modification to the matching policy to specify a total ordering over an object's superclasses.

5 Conclusions

$\mathcal{TS}$ is an attempt to increase the flexibility and expressivity of distributed data structures. First-class tuple-spaces contribute to modularity and abstraction. Permitting the programmer to customize the operational behavior of tuple-spaces makes it possible to build distributed data structures whose contents are devoid of bookkeeping clutter. Although the policy definitions given here subsume a wide range of programming paradigms, an interesting avenue of research is to develop linguistic mechanisms that permit users to generate other kinds of policy definitions dynamically that can be suitably interpreted by tuple-space operations.

First-class tuple-spaces and customization have implications on implementation as well. Because tuple-spaces are denotable objects with a concrete specification in the base language, we can apply standard optimization techniques (*e.g.*, flow- and lifetime analysis) to optimize their representation[15]. We anticipate that such analysis can be used to significantly reduce any implementation penalty incurred because of customization.

In summary, we view the design of $\mathcal{TS}$ as one step towards the realization of a highly flexible, efficient parallel programming system for symbolic computation. Work is currently underway on a parallel implementation.

References

[1] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Mass., 1986.

[2] V. Ambriola, P Ciancarini, and M. Danelutto. Design and Distributed Implementation of the Parallel Logic Language Shared Prolog. In *Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 40–49, March 1990.

[3] Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented System. In *Proceedings of the ECOOP Conf.*, pages 234–242, 1987.

[4] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-Structures: Data Structures for Parallel Computing. *Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[5] Nick Carriero and David Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3), September 1989.

[6] Nick Carriero and David Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444 – 458, April 1989.

[7] Andrew Chien and W.J. Dally. Concurrent Aggreates (CA). In *Second ACM Symposium on Principles and Practice of Parallel Programming*, pages 187–197, March 1990.

[8] K. Clark and S. Gregory. Parlog: Parallel Programming in Logic. *ACM Transactions on Programming Languages and Systems*, 8:1–49, 1986.

[9] William Cook and Jens Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *OOPSLA'89 Conference Proceedings*, pages 433–444, 1989. Published as SIGPLAN Notices 24(10), October, 1989.

[10] David Gelernter. Multiple Tuple Spaces in Linda. In *Proceedings of* PARLE '89, *volume 2*, pages 20–27, 1989.

[11] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Press, Reading, Mass., 1983.

[12] Robert Halstead. Multilisp: A Language for Concurrent Symbolic Computation. *Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[13] Waldemar Horwat, Andrew Chien, and William Dally. Experience with CST: Programming and Implementation. In *ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 101–109, June 1989.

[14] Paul Hudak and Philip Wadler, editors. Report on the Functional Programming Language Haskell. Technical Report YALEU/DCS/RR-666, Yale University, Dept. of Computer Science, December 1989.

[15] Suresh Jagannathan. Optimizing Analysis for First-Class Tuple-Spaces. In *Third Workshop on Parallel Languages and Compilers*, August 1990. Forthcoming from MIT Press.

[16] Samuel Kamin. Inheritance in Smalltalk-80: A Denotational Definition. In 15^{th} *ACM Symposium on Principles of Programming Languages*, pages 80–87, 1988.

[17] David Kranz, Robert Halstead, and Eric Mohr. Mul-T: A High Performance Parallel Lisp. In *Proceedings of the ACM Symposium on Programming Language Design and Implementation*, pages 81–91, June 1989.

[18] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *OOPSLA'86 Conference Proceedings*, pages 214–223, 1986.

[19] Jonathan Rees and editors William Clinger. The Revised^3 Report on the Algorithmic Language Scheme. *ACM Sigplan Notices*, 21(12), 1986.

[20] Ehud Shapiro. Concurrent Prolog: A Progress Report. *IEEE Computer*, 19(8):44–60, August 1986.

[21] Ehud Shapiro, editor. *Concurrent Prolog : Collected Papers.* MIT Press, 1987. Volumes 1 and 2.

[22] Brian Smith and J. des Rivières. The Implementation of Procedurally Reflective Languages. In *Proceedings of the 1984 Conference on Lisp and Functional Programming*, pages 331–347, 1984.

[23] David Ungar and Randall Smith. Self: The Power of Simplicity. In *OOPSLA'87 Conference Proceedings*, pages 227–241, 1987.

[24] United States Dept. of Defense. *Reference Manual for the ADA Programming Language*, 1982.

[25] Mitchell Wand and Daniel Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In *Proceedings of the 1986 Conference on Lisp and Functional Programming*, pages 298–307, 1986.

[26] A. Yonezawa, E Shibayama, T Takada, and Y. Honda. Object-Oriented Concurrent Programming – Modelling and Programming in an Object-Oriented Concurrent Language, ABCL/1. In *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, 1987.

A Formal Specification of the Process Trellis

Michael Factor*
IBM Israel, Science and Technology LTD

Abstract

The *process trellis* is a software architecture for building *parallel real-time monitors*: heterogeneous, large, real-time, continuously executing programs. These programs receive massive quantities of data in domains that are often ill-defined; they filter this data, presenting the user with an *analysis* rather than a simple summary. The process trellis combines heterogeneous processes that communicate among themselves *and* with the external world using a uniform framework.

We begin by motivating and describing the goals of the process trellis. After briefly reviewing the informal description of the trellis architecture, we extend our work by presenting a formal definition of the trellis architecture. We then show how several beneficial properties of the architecture follow from this definition. We briefly describe our experience with the process trellis in building a prototype monitor for a hospital intensive care unit.

1 Introduction

The *process trellis* is a software architecture for building *parallel real-time monitors*: heterogeneous, large, real-time, continuously executing programs. These programs receive massive quantities of data in domains that are often ill-defined; they "intelligently" filter this data, presenting the user with an *analysis* rather than a simple summary. The process trellis combines heterogeneous processes that communicate among themselves *and* with the external world using a uniform framework. We have previously presented an informal description of the trellis architecture [9, 8], demonstrating its usefulness in the construction of a new-generation monitor for a hospital intensive care unit, the Intelligent Cardiovascular Monitor (ICM) [11, 6, 15].

We begin by motivating and describing the goals of the process trellis. After briefly reviewing an informal description of the trellis architecture, we extend our work with a formal definition of the process trellis. Several beneficial properties follow from this

*This work was performed in large part at Yale University, Department of Computer Science, with support from National Library of Medicine grant TI5-LM-07056 and from National Science Foundation grant CCR-8657615.

definition. Using the formal definition, we present a very simple run-time execution scheme. This scheme, which makes it easy to understand a trellis program's run-time behavior, imposes a synchronous behavior on a trellis program, even though the trellis is a collection of independent, asynchronous processes. We also show that, even without this scheme, control-flow and data-flow dependencies impose a (loose) global synchronization on trellis programs. In addition, we show that, given three reasonable assumptions, any process trellis program will become quiescent after receiving new inputs from the "real world." We then briefly describe our experience with the process trellis in building an intensive care unit monitor.

2 Goals

Domains as diverse as medical monitoring, financial analysis and climatological data gathering are characterized by massive quantities of complex data. In all of these domains, the quantity and quality of the available data has tended to grow continuously in recent years. This huge quantity of data can easily lead to *information overload*, where there is more data available than a human can reasonably analyze and use. *Fixating* on a subset of the data and ignoring the remainder, is a natural human response to information overload. Real-time monitors use the power of a computer to minimize information overload and fixation, presenting users with an integrated, high-level analysis of the domain. Unfortunately, such applications have several characteristics which make them difficult to construct.

They are complex and heterogeneous. A real-time monitor will contain a diverse collection of modules since it must take in low-level data from multiple external sources, integrate this data and generate a high-level analysis of the domain as a whole.

They are large. These programs are not needed in trivial domains. It follows that a real-time monitor will be large.

They must run in real time. A monitor must analyze data from the external world as quickly as the data becomes available.

They will often need to take advantage of parallelism. To run a real-time monitor economically in real time, a uniprocessor will often be insufficient.

They must execute continuously. Rather than mapping from a set of inputs to a set of outputs, as in a "conventional program", a real-time monitor generates a continuous, time-varying, stream of outputs based upon its time-varying inputs.

The goal of the process trellis is to simplify the creation and maintenance of a real-time monitor by organizing and structuring the program. The trellis should meet the following specific goals:

Modularity: Modifying a module should have predictable effects.

Flexibility: This modularity cannot, however, come at the price of imposed homogeneity; the architecture must support a diverse collection of modules. In addition, there must be a flexible mechanism for the external world to interact with the program, since monitors execute continuously.

Understandability: A software architecture for large programs must aid in understanding a program's static structure as well as its dynamic behavior.

Predictability: Before real-time guarantees can be possible, one must be able to bound the module invocations engendered by any combination of new inputs.

3 The Process Trellis

The process trellis organizes a collection of heterogeneous decision processes into an acyclic hierarchical network. In this hierarchy, one process dominates another if the lower-level process calculates information that the higher-level process requires. Every process has a set of *inferiors*, a set of *superiors* and a *state*. The value of a process's state, which can be a complex object, depends upon the states of the process's inferiors and upon the state's own prior value, making processes history sensitive; as long as a process obeys the trellis's protocol, it can incorporate any kind of logic to calculate its state. When a process executes, it generates a new state if it has sufficient new information; each process defines "sufficient new information" for itself in an arbitrary, domain-dependent manner. Whenever *any* non-empty subset of its inferiors has new states, a process is *enabled* and can execute. Unlike Petri-net or conventional data-flow models, processes do not require inputs from all of their inferiors to be enabled; thus, processes execute with partial information. When a process generates a new state, its superiors are enabled and attempt to generate new states in turn.

Besides passively waiting for a new state from an inferior, a process can actively query any of its inferiors. A query contains no data nor is it part of a remote procedure call; it simply causes the lower-level process to attempt to generate a new state. A queried process executes and attempts to generate a new state; if it cannot generate a new state — it lacks sufficient information — it may in turn query its inferiors. The trellis, however, provides no guarantee that a queried process will ever generate a new state.

Sometimes higher-level knowledge is necessary to set the parameters of a lower-level process. The *context* is a (possibly empty) subset of a process's superiors whose states are seen by the inferior whenever the inferior executes. The context is intended to influence *how* a process computes; by contrast, its inferiors influence *what* and *when* the process computes. A context process generating a new state does not cause its inferiors to execute; this rule is necessary to avoid feedback loops with cyclic execution.

The process trellis allows two-way communication of both control and data between a process and its neighbors, but it carefully constrains this two-way communication. It distinguishes between the downward flow of control (queries) and the downward flow of data (context). Table 1 summarizes the inter-process communication protocol.

Besides providing for uniform inter-process communication, the trellis provides a uniform mechanism for external-world interactions. *Probes* are an application-independent

An Inferior Generates A New State	$\Longrightarrow$	Control Flows Up
A Superior Executes A Query	$\Longrightarrow$	Control Flows Down
An Inferior Generates A New State	$\Longrightarrow$	Information Flows Up
A Context Generates A New State	$\Longrightarrow$	Information Flows Down

Figure 1: The Trellis Protocol

mechanism for dynamically interacting with *any* process in a running trellis program. Significantly, the user, *not* the program, controls when and with which part of the program this interaction occurs.

Two symmetric "probes" are part of the model. A *write probe* sets the value of any portion of any process's state. A *read probe* reads any process's state. These probes are either *active* or *passive*. When a process receives an active write probe, it immediately executes as if an inferior had generated a new state. It can fully integrate the new data into its state and may generate a new state which its neighbors will see. When a process receives a passive write probe, the probe does not cause it to execute. In either case, the next time this process executes, for whatever reason, its state reflects this write probe. When a process receives an active read probe, if it does not have a currently defined state, it executes as if it were queried. When a process receives a passive read probe, whether or not it has a currently defined state, it does not execute. In either case, when the process next has a state, the read probe will return this state.

Every trellis process should always be as up-to-date as possible, given the trellis protocol. There is a natural, readily parallelized, *iterative execution scheme*, that insures this behavior: repeatedly sweep over the all of the trellis processes in a fixed order, executing any that have received a new state from an inferior, an active probe, or a query. Section 6 elaborates on this execution scheme.

4 Formal Definition

Using the informal, high-level description as motivation, this section formally defines the process trellis. A process trellis program is a set of processes $\mathcal{P} = \{P_1, \ldots P_n\}$; let $P, Q \in \mathcal{P}$ be arbitrary trellis processes. A programmer must supply the following for each process, P:

- I_P — Process P's *inferiors*. $I_P \subset \mathcal{P}$. The inferior relationship forms a directed acyclic graph, $(\mathcal{P}, \mathcal{E})$, where $\mathcal{E} \equiv \{(Q, P) \mid P, Q \in \mathcal{P} \wedge Q \in I_P\}$. Let $\overline{I}_P$ be process P's *superiors*; $\overline{I}_P = \{Q \mid P \in I_Q\}$

- C_P — Process P's *contexts*. The context is a subset of a process's superiors, i.e., $C_P \subseteq \overline{I}_P$.
- f_P — Process P's *state calculation function*. We explain this below.
- σ_P^0 — Process P's initial *private state*. It is a potentially complex object of arbitrary type.
- s_P^0 — Process P's initial *public state*. It is a potentially complex object of arbitrary type.

We distinguish between a process's *private state* and its *public state*. The private state is accessed only by that process; it can change without the process generating a new public state, i.e., the private state changing does not affect the process's superiors. By contrast, the public state, which was the state we described in the prior informal description, is accessed by the process's superiors as well as those processes that have this process for a context. This distinction allows a process to execute and change the data structure representing its (private) state without generating a new (public) state; thus, processes can be completely *history sensitive*, remembering prior invocations even if they do not generate a new public state. In addition, it encourages *information hiding*; a process need not make its entire state visible to its neighbors.

We describe the behavior of an isolated trellis process using recurrence equations. The interactions between processes, and thus the behavior of a trellis program as a whole, are implicit in these equations. In section 7 we describe the conditions under which these equations have a solution. Our initial definition ignores probes, focusing on the trellis as a closed system; section 4.2 extends this definition to include probes. (Throughout this section we present the formal definition first, then define the variables, and finally present a prose explanation of the equations.)

4.1 A Closed System

The behavior of a trellis process ignoring probes is given by:

$$\begin{aligned} \sigma_P^i \times s_P^i \times d_P^i \;\equiv\; & e_P^i \rightarrow \\ & f_P(\sigma_P^{i-1}, \{s_Q^{i-1} \mid Q \in I_P\}, \{s_Q^j \mid \gamma(P,Q,i,j)\}) \square \\ & \sigma_P^{i-1} \times \epsilon \times \{\,\} \qquad (1) \\ e_P^i \;\equiv\; & (\exists Q \,.\, P \in d_Q^{i-1} \wedge P \in I_Q) \vee \\ & (\exists Q \,.\, s_Q^{i-1} \neq \epsilon \wedge Q \in I_P) \qquad (2) \end{aligned}$$

where ϵ is a unique null value defined such that $\{\epsilon\} \equiv \{\,\}$, the notation $b \rightarrow c \square a$ is conditional execution, and

$$\gamma(P,Q,i,j) \;\equiv\; (Q \in C_P) \wedge (j < i) \wedge (s_Q^j \neq \epsilon) \wedge (\nexists k \,.\, j < k < i \wedge s_Q^k \neq \epsilon)$$

The following interpretation applies to the symbols we have not yet described:

- i — iteration count (recurrence index); $i \in 1 \ldots \infty$.
- s_P^i — process P's public state, a fixed-size, potentially complex object of arbitrary type.
- σ_P^i — process P's private state, an arbitrary type, fixed-size, potentially complex object.
- d_P^i — the set of inferiors process P queried during iteration i. $d_P^i \subseteq I_P$, $d_P^0 = \{\ \}$.
- e_P^i — a Boolean that is true *iff* process P is enabled for iteration i.
- γ — a macro to simplify equation 1; used to determine the set of context states passed to P's state calculation function.

A process is said to *generate a new* (public) *state during the i'th iteration* if its i'th public state is non-null, i.e., $s_P^i \neq \epsilon$. Conversely if $s_P^i = \epsilon$, process P did not generate a new state during iteration i. The initial public state of each process, s_P^0, which is supplied by the programmer, must be non-null. This guarantees that there is at least one defined public state (a non-ϵ state) for each process; we motivate the need for this below. The fact that a process's i'th public state is ϵ implies nothing about the value of its i'th private state.

Equation 1 shows the i'th execution of process P. On its i'th execution, process P calculates its i'th private state (σ_P^i), i'th public state (s_P^i) and i'th set of queried inferiors (d_P^i); this is the left side of the equation. If the process is enabled for the i'th iteration, i.e., e_P^i is true, its i'th private state, public state, and set of queried inferiors are determined by invoking process P's state calculation function f_P.

If process P is not enabled for the i'th iteration, i.e., e_P^i is false, the trellis protocol specifies the process's behavior. Its private state does not change (i.e., $\sigma_P^i = \sigma_P^{i-1}$), it does not generate a new public state (i.e., its i'th public state is ϵ), and it does not execute any queries (i.e., $d_P^i = \{\ \}$). The last line of equation 1, the alternate branch of the conditional, shows this behavior.

f_P is a three-argument, black-box function which returns a three tuple. The first argument is process P's prior private state (σ_P^{i-1}). The second argument to f_P is the set of public states of process P's inferiors:

$$\{s_Q^{i-1} \mid Q \in I_P\}$$

Since this set only includes the states of those processes that generated new states during the prior iteration (if a process Q did not generate a new state during iteration $i-1$, $s_Q^{i-1} = \epsilon$ and $\{\epsilon\} = \{\ \}$), its cardinality ranges from zero to $|I_P|$, the size of the set I_P. The final argument to f_P is the set containing the most recently generated public state of each of process P's contexts. Replacing γ with its definition, this third argument is:

$$\{s_Q^j \mid (Q \in C_P) \wedge (j < i) \wedge (s_Q^j \neq \epsilon) \wedge (\nexists k \,.\, j < k < i \wedge s_Q^k \neq \epsilon)\}$$

Since this set contains the newest public state of each context process, its cardinality is always $|C_P|$. If we had not required the initial public state of each process to be non-null,

i.e., $s_P^0 \neq \epsilon$, we could not guarantee the existence of a defined public state for each process Q in P's context.[1] f_P is not permitted to side-effect any of its arguments; however, other side-effects are permissible.

f_P returns a three-element tuple; although f_P's type-signature is fixed, f_P can determine its return values using arbitrary logic. The first element of the returned cross product is process P's i'th private state. The i'th private state may differ from the $i-1$'st, even if P did not generate a new public state on its i'th execution, i.e., even if $s_P^i = \epsilon$. The second element is process P's i'th public state. This element is ϵ if P had "insufficient information to generate a new state" during its i'th execution; as mentioned above, this element is also ϵ if P was not enabled for the i'th iteration. The final element is the subset of its inferiors that P is querying. A query causes an inferior process to be enabled for the next iteration; the trellis protocol includes no presumption that a queried process will ever generate a new state, i.e., the process trellis protocol does not view a query as a remote procedure call. In addition, since a state is always available for each context process — the context is the most recently generated public state, not the state from the prior iteration — the trellis protocol cannot deadlock with a queried inferior waiting for a new context state from the querying process. By contrast the internal logic of the state transformation functions, the black boxes, can incur a "local deadlock," in which individual processes fail to generate new public states, i.e., the state calculation function returns ϵ for the process's new public state. However this local deadlock is uninteresting from the trellis's perspective because *a*) assuming that the f_P's do not diverge, the trellis as a whole is deadlock free (all variables ultimately depend only upon values computed during prior iterations) and *b*) the trellis cannot distinguish between "local deadlock" and failure to generate a new public state for any other lack of sufficient information.

A process is enabled for the i'th iteration in one of two ways, as equation 2 shows. P is enabled if it has been queried by (at least) one of its superiors:

$$\exists Q \;.\; P \in d_Q^{i-1} \wedge P \in I_Q,$$

or if it has received new information from an inferior generating a new state on the prior iteration:

$$\exists Q \;.\; s_Q^{i-1} \neq \epsilon \wedge Q \in I_P.$$

4.2 An Open System

We have defined the process trellis program as a closed system. But the external world must be able to enter data into, and retrieve results from, a real-time monitor. Probes are the mechanism for this interaction. A read-probe returns a process's public state, and a write probe sets any portion of any process's private state. This informal description ignores timing. A process's state evolves over time, with its i'th state (potentially) differing from its $i+1$'st state; which version of the state should a probe access? We address this issue by assuming that a probe for process P occurs instantaneously during some iteration i. Since probes occur instantaneously, we avoid the complications that would

[1] A slightly more complicated constraint that took into account which processes were used as contexts would suffice to make this guarantee.

arise if a probe could begin during one iteration and complete during another. However, even with this simplifying assumption, adding probes to the formal definition requires several extensions.

First, we add two recurrence variables for each process (the meaning of these variables will become clear only after the definition of probes); these variables are both two-element tuples.

$$r_P^0 \equiv \mathbf{F} \times \mathbf{F}$$
$$r_P^i \equiv ((r_P^{i-1} \downarrow 1) \wedge (s_P^{i-1} = \epsilon)) \rightarrow (\mathbf{T} \times \mathbf{F}) \Box (\mathbf{F} \times \mathbf{F}) \quad (3)$$
$$w_P^i \equiv \mathbf{identity} \times \mathbf{F} \quad (4)$$

where $a \downarrow n$ is the n'th element of the cross product a, **identity** is the identity function, and **T** and **F** are Booleans. These variables get a value in two ways: 1) the recurrence equations 3 and 4 define their initial value and 2) as equations 5 and 6 show, probes can asynchronously change the initial value. In equations 7 and 8, we use these variables to refine the behavior of a trellis process to account for interactions with the outside world.

Interpret the variables in equations 3 and 4 as:

- r_P^i — Process P's read probe recurrence variable is the cross product of two Booleans. The first Boolean is true *iff* there is an *unsatisfied read probe* for process P. As equation 3 shows, P has an unsatisfied read probe during iteration i if it had an unsatisfied read probe during the prior iteration $(i-1)$ and it did not generate a new state during that iteration. The second Boolean is true *iff* an active read probe was executed for process P during iteration i; we show this in the definition of read probes below. The second Boolean is initially false.

- w_P^i — Process P's write probe recurrence variable is a cross product: a function whose type is from the type of the process's private state to that same type and a Boolean. The first element, which is initially the identity function, tells how the write probe modifies the process's private state. Like r_P^i, the second Boolean is true *iff* an active write probe was executed for process P during iteration i.

A probe for process P is executed during the i'th iteration if, when the probe is executed, process P's $i-1$'st private state is defined but its i'th private state is not; this is well-defined since probes are executed instantaneously. We do not specify how to determine the iteration during which a probe is executed.

Probes, which enter data or retrieve results from any process in a running trellis program, side-effect the recurrence variables. A write probe alters the variable w_P^i, and a read probe alters r_P^i, where i is the iteration during which the probe was executed.

$$\mathbf{WriteProbe}_P(g,b) \equiv w_P^i \leftarrow (g \circ (w_P^i \downarrow 1)) \times ((w_P^i \downarrow 2) \vee b) \quad (5)$$
$$\mathbf{ReadProbe}_P(b) \equiv r_P^i \leftarrow \mathbf{T} \times ((r_P^i \downarrow 2) \vee b) \quad (6)$$

where $g \circ h$ is function composition and $a \leftarrow b$ assigns b to a.

A write probe for process P has two arguments: g, a function whose type is the type of the process's state to that same type, and b, a Boolean that tells whether the

probe is active. These are the same types as the elements of w_P^i. The function g tells how the private state of process P should be modified. The specification permits a write probe to make arbitrary modifications to P's private state based upon the state's current value. However, an implementation may restrict the complexity of this function due to the difficulty of allowing arbitrary, interactively-specified modifications. When a write probe is executed, g is composed with the first field of w_P^i (P's write probe recurrence variable):

$$g \circ (w_P^i \downarrow 1).$$

Though this permits multiple write probes to be executed during the same iteration for the same process, a later probe can hide the effects of an earlier probe. After a write probe is executed for process P during iteration i, the second element of w_P^i is true if this probe was an active probe or if an active write probe had already been executed during iteration i:

$$(w_P^i \downarrow 2) \vee b.$$

A read probe for P has one argument, a Boolean indicating whether the probe is active. A read probe for process P executed during iteration i modifies the recurrence variable r_P^i. The first element of the tuple r_P^i is set to true since there is an unsatisfied read probe for process P, namely, the probe being executed. The number of unsatisfied read probes is irrelevant. The second field is true either if this is an active probe or an active probe was already executed during iteration i; this is analogous to the definition for write probes.

We now show how the addition of probes modifies the definition of the trellis as a closed system:

$$\begin{aligned} \sigma_P^i \times s_P^i \times d_P^i \quad &\equiv \quad e_P^i \rightarrow \\ &\qquad f_P(\hat{\sigma}_P^{i-1}, \{s_Q^{i-1} \mid Q \in I_P\}, \{s_Q^j \mid \gamma(P,Q,i,j)\})\square \\ &\qquad \hat{\sigma}_P^{i-1} \times \epsilon \times \{\,\} \qquad (7) \\ e_P^i \quad &\equiv \quad ((\exists Q \,.\, P \in d_Q^{i-1} \wedge P \in I_Q) \vee (r_P^{i-1} \downarrow 2)) \vee \\ &\qquad ((\exists Q \,.\, s_Q^{i-1} \neq \epsilon \wedge Q \in I_P) \vee (w_P^{i-1} \downarrow 2)), \qquad (8) \end{aligned}$$

where

$$\hat{\sigma}_P^i \quad \equiv \quad (w_P^i \downarrow 1)(\sigma_P^i)$$

Equation 7 refines equation 1 to account for the modifying effects of a write probe. We have replaced occurrences of the process's private state (σ_P^{i-1}) on the right-hand side with $\hat{\sigma}_P^{i-1}$: the $i-1$'st private state, updated to reflect the changes specified by any write probes executed during the i'th iteration. If no write probe was executed during the i'th iteration, the updated version of the process's private state is the same as the original version, i.e., $\hat{\sigma}_P^{i-1} = \sigma_P^{i-1}$, since the first field of w_P^i is initially the identity function. Even if P is not enabled for the i'th iteration, P's private state is updated to reflect the effects of any write probes.

Equation 8 modifies equation 2 to account for the enabling effects of active probes. A process is enabled for the i'th iteration if during the prior iteration it was queried, by a

P is enabled if it:

1. received new data from:
   - an inferior.
   - an active write probe.
2. received a query from:
   - a superior.
   - an active read probe.

Figure 2: Process Enablement

superior or by an active read probe:

$$(\exists Q \,.\, P \in d_Q^{i-1} \wedge P \in I_Q) \vee (r_P^{i-1} \downarrow 2),$$

or if it received new information either from an inferior or an active write probe:

$$(\exists Q \,.\, s_Q^{i-1} \neq \epsilon \wedge Q \in I_P) \vee (w_P^{i-1} \downarrow 2).$$

Figure 2 summarizes the rules for process enablement.

While we have presented the definition of a read probe, we have not shown how a probe returns the process's state, which, after all, is the point. $\mathbf{Display}_P^i$ is true *iff* process P's i'th public state should be returned to the user as the result of a read probe.

$$\mathbf{Display}_P^i \equiv ((r_P^i \downarrow 1) \wedge (s_P^i \neq \epsilon)).$$

This returns the first new state generated by process P after the read probe is executed. If the process generates a new state during the same iteration during which the probe is executed, this state is returned. We do not specify the mechanism by which the state is actually displayed.

The recurrence equations given in this section do not necessarily have a solution. Intuitively, there are two reasons for this: 1) the equations contain a black-box component, f_P, and 2) the equations do not define a closed system, i.e., probes can asynchronously change the value of a recurrence variable. However, given certain reasonable assumptions which address these issues, these equations do have a solution. Section 7 describes these assumptions from an operational point of view.

5 Global Behavior

Our formal definition left implicit the global behavior of trellis programs, considering only individual processes in isolation. This makes it simple to think about individual processes, but difficult to reason about a program as a whole. However, we can view the formal definition of a trellis program as an operational description. Processes compute

```
For i = 1...∞
    For every process P
        If P is enabled then execute P.
```

Figure 3: Iterative Execution Scheme

successive values of their recurrence variables. For a process to compute the values of its i'th recurrence variables (e.g., compute its i'th private state), all of its inferiors and superiors must have computed their $i-1$'st variables. As follows from equation 8, determining if a process is enabled for iteration i requires knowing the queries executed by all of its superiors for iteration $i-1$ and knowing whether its inferiors generated new states during iteration $i-1$.[2]

A process's *current iteration* is the greatest iteration for which the value of the process's private state has been calculated. Assuming that the trellis graph is connected, the trellis protocol bounds the difference between any two processes' current iterations.

Theorem 1 *The difference between the current iteration of any two processes can be no greater than the length of the shortest path connecting the two processes in the undirected graph* $(\mathcal{P}, \mathcal{E})$ *where* $\mathcal{E} \equiv \{(Q,P) \mid \exists P \in \mathcal{P} \,.\, Q \in I_P\}$.

Proof: It follows from equation 8 that determining if a process is enabled for the i'th iteration requires that all of its neighbors (inferiors and superiors) have executed their $i-1$'st iterations. Given this observation, a simple induction on the length of the shortest path connecting two processes enables us to conclude the claim. ∎

Thus, while the trellis processes are defined as asynchronous and independent entities, the behavior of a trellis program as a whole has only limited asynchronism. Control-flow and data-flow dependencies impose a (loose) global synchronization on trellis programs.

6 The Iterative Execution Scheme

The trellis structure inherently constrains how out-of-synchrony any two processes can be. However, we wish to make a stronger guarantee; we want all processes to always be as up-to-date as possible, i.e., we want to guarantee that the greatest difference between any two processes' current iteration is one. The *iterative execution scheme*, which we mentioned in our informal description of the process trellis, makes this guarantee; it repeatedly sweeps over all of the processes in a trellis, executing any that are enabled. This scheme (see figure 3) guarantees that no process executes more than once while another process, which has been enabled at least as long, does not execute at all. The iterative execution scheme constrains the order in which the recurrence variables are calculated. The i'th private state must be computed for all processes before the $i+1$'st is computed for any process.

[2] We assume applicative-order evaluation.

A parallel version of the iterative execution scheme replaces the inner loop with (what is effectively) a PARDO. Except for the communications that are part of the trellis protocol, the processes are independent. All of the data a process needs during the i'th iteration is calculated during the $i-1$'st iteration. Thus, during each iteration all of the processes can execute in parallel.

(LUSTRE [5] is a data-flow language for synchronous real-time systems. Ignoring interactions with the external world, the dynamic behavior of a LUSTRE program is similar to the iterative execution scheme. A LUSTRE program is a data-flow operator net with the additional assumption that operators respond instantly to their inputs. A clock is associated with each operator; all of the clocks are multiples of a base clock. An operator is only enabled when its clock has advanced. A LUSTRE program has cyclic behavior; on the n'th cycle, all operators whose clocks have advanced at time n calculate a new value. While the trellis implementation, which [9] describes, is based on this scheme, LUSTRE seems to use this observation only as a pedagogical tool in giving language semantics.)

7 Stability

Under certain reasonable assumptions, we can show that arbitrary trellis programs will become quiescent. We call this quiescence property *stability*; a running process trellis program is *stable* if no process is enabled or executing. Once a trellis program is stable, it remains stable until acted upon by an external agent executing a probe. A stable program generates no states and executes no queries.

Any trellis program that meets the following three conditions possesses the stability property.

1. No process generates a new state without receiving a new state from an inferior or new data from an active write probe. This corresponds to processes not being permitted to create information. Note that a new context state or a passive write probe is insufficient.

2. No active probes are executed during the interval under consideration.

3. Every process terminates whenever it executes.

The first assumption restricts processes to generating new states only when they have received new information. A process's public state may not always reflect the most recent values of all of its context states, but it can always reflect the newest values of all of the process's inferior states. This does not significantly restrict the behavior of trellis processes since the context is intended to influence how, but not what, a process computes, and since a context process generating a new state does not enable its inferiors. Since processes are black boxes, this assumption is impossible to guarantee at compile time. However, it is easy to verify compliance at run time. Second, since "stability" means that a program will become quiescent in a predictable time after receiving new inputs, it is reasonable for stability to depend on the absence of new inputs. Finally, while real-time monitors are

themselves non-terminating, their components should terminate; no individual calculation may diverge. Thus, the this assumption is also reasonable, even though it is impossible to guarantee (for black-box code) at either compile or run time.

Theorem 2 *Assuming:*

1. *No process generates a new public state without receiving new information:*

$$s_P^i \neq \epsilon \Longrightarrow (\exists Q \;.\; s_Q^{i-1} \neq \epsilon \wedge Q \in I_P) \vee (w_P^{i-1} \downarrow 2).$$

2. *No active probes are executed:*

$$\forall P, \forall i \in j \ldots 2\mathcal{H} + j \;.\; (r_P^i \downarrow 2 = \mathbf{F}) \wedge (w_P^i \downarrow 2 = \mathbf{F}).$$

3. *f_P terminates for all P.*

4. *No process calculates its i'th public state before all processes have calculated their $i-1$'st public state.*[3]

Given any initial configuration of a trellis at iteration $j-1$, there will be no enabled processes during iteration $j + 2\mathcal{H} + 1$.

Before proving the stability property, we present some definitions. Define $\mathcal{L}(P)$, the *level number* of process P, as follows:

$$\mathcal{L}(P) \equiv (I_P = \{\,\}) \rightarrow 0 \square 1 + \max_{Q \in I_P} \mathcal{L}(Q).$$

The level numbers give a topological sort. The height, $\mathcal{H}$, of a trellis program, $\mathcal{P}$, is the maximum level number of any process in the program:

$$\mathcal{H} \equiv \max_{\mathcal{P}} \mathcal{L}(P).$$

Process P is *up-enabled* during iteration i if its generating a new state during iteration i would not violate assumption 1. Formally, P is up-enabled during iteration i if

$$(\exists Q \;.\; s_Q^{i-1} \neq \epsilon \wedge Q \in I_P) \vee (w_P^{i-1} \downarrow 2). \tag{9}$$

In other words, a process is up-enabled if, during the prior iteration, either an inferior generated a new state (the first disjunct) or it received an active write probe (the second disjunct). In the absence of active probes a process can be up-enabled during an iteration only if (at least) one of its inferiors was up-enabled during the prior iteration.

[3]This assumption is not necessary to show stability, *per se*; however, it simplifies the proof by allowing a global view of an iteration.

Proof: The proof of theorem 2 has two parts.

During iteration $j+i$ (for positive integers i) there are no up-enabled processes with $\mathcal{L}(P) < i$. By induction on the iteration number:

The base case, $i = 1$. Since processes with $\mathcal{L}(P) < 1$ have no inferiors, the first disjunct in equation 9 is false, and since by assumption no active probes are executed during iteration j, the second disjunct is also false.

Inductive Hypothesis: Assume that during iteration $j+i$, no process P with $\mathcal{L}(P) < i$ is up-enabled. By the inductive hypothesis, no inferiors of processes at level i or lower were up-enabled during iteration $j+i$. Therefore, during iteration $j+i+1$ there are no up-enabled processes P with $\mathcal{L}(P) < i+1$; in particular, during iteration $j+\mathcal{H}+1$ there will be *no* up-enabled processes.

In the absence of active probes and new states from inferiors, the only way for a process to be enabled during iteration i is if a superior queried the process during iteration $i-1$.

During iteration $j+\mathcal{H}+i$ there are no enabled processes with $\mathcal{L}(P) > \mathcal{H}-i$. By induction:

The base case, $i = 1$. Since during iteration $j+\mathcal{H}$ there are no up-enabled process with $\mathcal{L}(P) < \mathcal{H}$ and since processes with $\mathcal{L}(P) = \mathcal{H}$ have no superiors, no process with $\mathcal{L}(P) > \mathcal{H}-1$ will be enabled during iteration $j+\mathcal{H}+1$.

Inductive Hypothesis: Assume that during iteration $j+\mathcal{H}+i$ no process P with $\mathcal{L}(P) > \mathcal{H}-i$ will be enabled. By the inductive hypothesis, no superior of a process at level $\mathcal{H}-i$ or greater will be enabled during iteration $j+i$. Since in the absence of probes and new states from inferiors the only way for a process to be enabled is to be queried by a superior and since no superior of a process with $\mathcal{L}(P) > \mathcal{H}-i$ is enabled during iteration $j+\mathcal{H}+i$, no process with $\mathcal{L}(P) > \mathcal{H}-i-1$ will be enabled during iteration $j+\mathcal{H}+i+1$; in particular, during iteration $j+2\mathcal{H}+1$ there will be no enabled processes. ∎

The ability to show stability is derived from the decoupling of control flow from information flow. This decoupling enables the trellis to possess the stability property, while still permitting bidirectional communication. Without this decoupling, it would be impossible to show quiescence (stability) for a generic trellis program. While we need to make some assumptions about the behavior of individual trellis processes to show stability, it is not necessary to understand the internal details of a particular trellis program to show quiescence.

Saying that a process trellis possesses the stability property is the operational equivalent of saying that the recurrence equations describing a trellis program have a unique solution. The stability assumptions are the conditions necessary for the recurrence equations given in section 4 to have a solution. The stability property says that after a trellis is externally modified by probes, it will, within a known number of iterations, reach a global state in which no processes are enabled, and thus no processes generate new states or execute queries. Taking a non-operational view of this statement, if assumptions 1, 2, and 3 hold, then starting from any values of the recurrence variables at iteration $j-1$, a "fixed point"[4] will be reached by iteration $j+2\mathcal{H}+1$ in which $\forall P$:

[4]For the trellis, the standard definition of "fixed point" is inappropriate since the recurrence variables can be externally modified by probes, thus moving a trellis program away from a fixed point. Instead,

1. $s_P^k = \epsilon$, i.e., no process generates a new public state.
2. $d_P^k = \{\ \}$, i.e., no process is queried.
3. $e_P^k = \mathbf{F}$, i.e., no process is enabled.

This fixed point will hold for all iterations k, $l \geq k \geq j + 2\mathcal{H} + 1$, where l is the first iteration $\geq j + 2\mathcal{H} + 1$ during which an active probe is executed.

8 Experience

To aid in writing process trellis programs, we have designed and implemented the process trellis shell. To use this shell, which is for the most part faithful to the specification, a programmer supplies for each process its state calculation function, its inferiors, its contexts, its initial state and several other attributes. The shell includes a run-time kernel that implements a parallel version of the iterative execution scheme; this kernel handles all of the parallelism and all of the inter-process communication.

We have used this shell, to implement the Intelligent Cardiovascular Monitor (ICM), a prototype ICU (intensive care unit) monitor [11, 6, 15, 10]. The ICM seeks to provide early, systematic detection of evolving trends, taking advantage of asynchronous inputs when available but using primarily on-line data.[5] Currently, the ICM contains over one hundred processes. The prototype ICM runs off-line, but at real-time rates, at a frequency up to one hundred times a second.

9 Related Work

PSDL [14, 13] is a description language for rapidly prototyping large, real-time systems. PSDL combines "operators" using a modified, idiosyncratic data-flow firing rule. These operators are either hierarchically composed of other operators, or they are arbitrary code implemented in some base language. Thus, PSDL permits heterogeneous modules. Since PSDL programs can contain cycles, one cannot bound the operator firings that will result when new data enters the program. Nor does PSDL provide a uniform mechanism for the external world to interact with any operator; although it does provide mechanisms that can model input from the external world.

The process trellis has been compared to object-based and object-oriented systems (e.g., Actors [12, 1]) and to concurrent object-based systems (e.g., [3, 4]). Since object connections are all internal to the objects — an object explicitly communicates with other objects from within its code — one cannot easily determine the complete effects of one object on the rest of the system. In addition, although one can implement a uniform external interface for objects by creating methods for this interface at the most general

we view a trellis as being at a "fixed point" when no recurrence variables can change except due to an external event, namely the execution of a probe.

[5]The ICM is joint work with Drs. Dean Sittig, Aaron Cohn, Stanley Rosenbaum and Perry Miller of the Yale School of Medicine, Department of Anesthesiology.

object in the hierarchy and having these methods be inherited by all other objects, no uniform interface with the external world is inherent in objects. Ada [2, 7] is probably the best known language for real-time programming. While Ada is a real-time language — or more properly, a language for embedded systems which by their nature have real-time constraints — it has been long realized that Ada's facilities for dealing with time (e.g., the delay statement, static priority assignments, etc.) are inadequate for real-time applications [16].

10 Conclusions

We have presented a formal definition of the process trellis software architecture and have shown how several benefits follow from this structure. The iterative execution scheme is a simple description of a trellis program's global, run-time behavior; it cleanly guarantees that all processes are always as up-to-date as possible. Even without the iterative execution scheme and despite the fact that trellis processes are defined as independent asynchronous entities, the definition of the trellis limits the asynchronism of the program as a whole, making the program easier to understand. In addition, though permitting bidirectional information and control flow, the trellis structure is such that, under a reasonable set of assumptions, a trellis program will predictably become quiescent after new data enters.

We conclude by reviewing how the process trellis addresses the goals presented in section 2. Process trellis programs are clearly modular; they are collections of independent processes. They are also flexible insofar as processes are heterogeneous; the state calculation function, f_P, can use any logic as long as it conforms to the appropriate type signature.

Since a process is enabled whenever any inferior generates a new state, processes execute with partial information. Trellis processes are history sensitive, an essential feature for a continuously executing program. In addition, probes provide a simple, uniform mechanism for dynamic, non-pre-planned interactions between the trellis and its environment — an especially important attribute for a continuously executing program.

Though defined as a collection of asynchronous processes, trellis programs as a whole have (loosely) synchronous behavior. This greatly enhances global understandability. Even without the iterative execution scheme, which is very clean and simple to understand, the definition of the structure prevents some processes from running all of the time while other processes don't run at all.

Finally, trellis programs are explicitly parallel. The iterative execution scheme is naturally parallelized; [9] shows that this can be very efficient.

References

[1] G. A. Agha. *ACTORS: A model of Concurrent Computation in Distributed Systems.* The MIT Press, Cambridge, MA, 1986.

[2] J. G. P. Barnes. An overview of Ada. *Software Practice and Experience*, 10:851–887, 1980.

[3] J. K. Bennet. The design and implementation of distributed Smalltalk. In *Proceedings 1987 ACM Conference on Object Oriented Programming*, pages 318–330, Dec. 1987.

[4] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, Jan. 1987.

[5] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *14th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 178–188. ACM Sigplan, Jan. 1987. Munich, West Germany.

[6] A. I. Cohn, S. Rosenbaum, M. Factor, and P. L. Miller. DYNASCENE: An approach to computer-based intelligent cardiovascular monitoring using sequential clinical 'Scenes'. *Methods of Information in Medicine*, 29:122–131, Apr. 1990. Revised version of paper in *SCAMC-89.*

[7] Department of Defense, Washington, DC. *Military Standard Ada Programming Language*, MIL-STD-1815 edition, Apr. 1982.

[8] M. Factor. *The Process Trellis Software Architecture for Parallel, Real-Time Monitors.* PhD thesis, Yale University, Department of Computer Science, Dec. 1990. New Haven, CT.

[9] M. Factor. The process trellis software architecture for real-time monitors. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 147–155. ACM, SIGPLAN, Mar. 1990. Seattle, WA. (*SIGPLAN Notices*, 25(3)).

[10] M. Factor, D. H. Gelernter, C. Kolb, P. L. Miller, and D. F. Sittig. Real-time performance, parallelism and program visualization in medical monitoring. Research Report YALEU/DCS/RR-808, Yale University, Department of Computer Science, June 1990.

[11] M. Factor, D. F. Sittig, A. I. Cohn, D. H. Gelernter, P. L. Miller, and S. Rosenbaum. A parallel software architecture for building intelligent medical monitors. *International Journal of Clinical Monitoring and Computing*, 7:117–128, 1990. Revised version of paper in *SCAMC-89.*

[12] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, June 1977.

[13] Luqi and V. Berzins. Rapidly prototyping real-time systems. *IEEE Software*, pages 25–36, Sept. 1988.

[14] Luqi, V. Berzins, and R. T. Yeh. A prototyping language for real-time software. *IEEE Transactions on Software Engineering*, 14(10):1409–1423, Oct. 1988.

[15] D. F. Sittig and M. Factor. Physiologic trend detection and artifact rejection: A parallel implementation of a multi-state Kalman filtering algorithm. *Computer Methods in Programs in Biomedicine*, 31:1–10, 1990. Revised version of paper in *SCAMC-89*.

[16] J. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19, Oct. 1988.

Strong Bisimilarity on Nets Revisited
(Extended Abstract)

C. Autant, Z. Belmesk and Ph. Schnoebelen *

Laboratoire d'Informatique Fondamentale
et d'Intelligence Artificielle,
Institut Imag - CNRS,
Grenoble - FRANCE

Abstract

In [Old89b], Olderog proposed a new notion of bisimulation between Petri nets. His proposal considers bisimulations *between places of nets* rather than between markings. Unfortunately, his definition leads to several problems that have not been noticed. It turns out that the situation is more complicated than in classical bisimulation theory over transition systems.

We propose a new definition which solves the problems and which is much more general. We investigate the consequences of the new definition: many results of classical bisimulation theory can be recovered modulo some adaptation. This indicates that our definition is "correct" and productive.

1 Introduction

Bisimulation, introduced in [Par81], is a fundamental concept in the theory of concurrency (see [BK89] for a review). Informally, two systems are bisimilar if their possible behaviors have the same branching structure, i.e. any behavior of one system can be reproduced by the other system, in a way that preserves the places where the non-deterministic choices have been made.

Classical bisimulation theory deals with transition systems (non-deterministic automata where transitions are labeled by actions.) It is possible to adapt the basic notion to systems with a richer structure, e.g. (Petri) nets. A simple way to do this is to view a net as a transition system, by considering the graph of its global states [Pom85, vGV87]. But this does not take the richer structure into account. Several recent proposals (e.g. [Pom85, vGV87, RT88, BDKP90, Dev90]) took (part of) the structure of nets into account and defined bisimulations between markings (the global states) that preserve (part of) the structure of transition sequences.

Recently, Olderog proposed a new notion of bisimulation between nets [Old89b, Old89a]. He considered bisimulations *between places of nets* rather than between markings. The relation is then lifted from places to markings. One advantage is that a relation between places is easier to visualize (it can easily be drawn on the graphical representation of a net) and to understand.

*LIFIA-IMAG, 46 Av. Félix Viallet, 38031 GRENOBLE Cedex, FRANCE. E-mail:{cyril,zoubir,phs}@lifia.imag.fr

This new semantic equivalence preserves more information. Bisimilar markings must be consistent at the lower structural level of the net: e.g. they must carry the same number of tokens. This yields a stronger and richer equivalence. It really is a new concept in the semantics of nets and fully deserves further exploration.

Unfortunately, the definition in [Old89b] leads to several subtle problems that have not been noticed (as far as we know). The most striking one is that it does not give an equivalence relation ! In fact, the situation is more complicated than it appears at first sight and the whole subject is rather treacherous.

In this paper, we propose a new definition that starts from Olderog's seminal idea of a bisimulation between places of nets. This new definition solves the problems of the previous one, and is more general. We compare its discriminating power with classical bisimulations on nets.

In order to gain some confidence that it is a "correct definition", we consider a few properties, inspired from the classical case, that any bisimulation should satisfy. Specifically, these *requirements for a bisimulation* are:[1]

1. a bisimulation must be an equivalence relation,
2. there should exist largest bisimulations,
3. there should exist canonical representatives of equivalence classes of bisimilar nets,
4. nets having isomorphic unfoldings into acyclic nets should be bisimilar.

As an example, [Old89b] proves that his definition of bisimulation identifies isomorphic nets and preserves concurrent computations. These are subsumed by point 4 in our list. Nevertheless, the definition in [Old89b] fails on all four points.

In this paper, we prove that the first three properties of our list are indeed satisfied (possibly with some adaptation) by our definition. Section 5 explains why unfolding is not consistent with the idea of bisimulation between places. These results give a somewhat deeper understanding of the new notion and we show precisely where and when some restrictive hypothesis is required. Furthermore, useless restrictions can cripple a theory: for example we show (see Remark 2 in section 7 and developments in [ABS91]) that it is unwise to restrict oneself to safe nets.

The paper is organized as follows: We recall the general theoretical background in section 2. Then section 3 presents Olderog's seminal idea and our proposal, before we discuss Olderog's original definition in section 4. Then section 5 gives several examples aiming at explaining our definition. We investigate largest bisimulations in section 6 and canonical representatives in section 7. Section 8 mentions a few variants of our definition and explains why we discarded them. We conclude in section 9 by assessing the potential applications of the theory, and by listing several research directions that should be investigated in future work.

Some easy proofs have been omited and some long proofs have been sketched. As a rule, complete proofs appear in the full version of this paper [ABS91].

[1] We did not include congruence properties as this paper is not concerned with compositions of nets.

2 Basic definitions on nets

We give here the basic definitions and notations about nets we shall use in the following. (See also e.g. [Rei85].)

2.1 Multisets

Given any set X, a *multiset* M over X is a mapping from X to ω, the set of natural numbers. For $x \in X$, $M(x)$ is the *multiplicity* of x in M and we write $x \in M$ when $M(x) > 0$. When $M(x) \leq 1$ for all $x \in X$, M is a proper set.

M is *finite* if $M(x) = 0$ for all $x \in X$ except maybe a finite number of them. From now on we only consider finite multisets and write $\mathcal{M}(X)$ the set of all finite multisets over X. The usual notation will be used to denote the elements of a finite multiset. For example, if $M(a) = 3$, $M(b) = 2$ and $M(x) = 0$ for $x \neq a$, $x \neq b$, we may write $M = \{a, a, a, b, b\}$.

Set-theoretic notions are extended to finite multisets in the standard way: given $M, M' \in \mathcal{M}(X)$, we define $M + M'$ by $(M + M')(x) = M(x) + M'(x)$. We write $M \subseteq M'$ when $M(x) \leq M'(x)$ for all $x \in X$. When $M' \subseteq M$, we define $M - M'$ by $(M - M')(x) = M(x) - M'(x)$. Finally, we write $\emptyset$ for the empty multiset.

2.2 Labeled nets

We assume a set $Act = \{a, b, \ldots\}$ of *action names* or *labels.*

Definition 1 *A* labeled Petri net, *or simply a net, is a tuple* $N = \langle P, T, pre, post, l, M_{init} \rangle$ *where*

- P *is a set of* places, *with typical elements* $p, p', p_1, q, q', \ldots$,
- T *is a set of* transitions, *with typical elements* $t, t', t_1, \ldots$,
- *pre (resp. post) is a mapping from* T *to* $\mathcal{M}(P)$. *Given* $t \in T$, $pre(t)$ *and* $post(t)$ *are the* precondition *and the* postcondition *of* t,
- $l : T \to Act$ *is a* labeling *of transitions with action names,*
- $M_{init} \in \mathcal{M}(P)$ *in an* initial marking.

Note that we allow P empty, M_{init} empty, and $pre(t)$ or $post(t)$ empty. Given a net N, we shall denote its components by P_N, T_N, pre_N, $post_N$, l_N and $init(N)$. A net N is *finite* (resp. *countable*) if $P_N \cup T_N$ is.

In following sections, we use the standard graphical representation of nets, with circles for places and boxes for transitions. The *pre* (resp. *post*) relation is drawn through directed edges leading from places to transitions (resp. from transitions to places) with e.g. 3 edges from a same place p to a same t if p appears 3 times in $pre(t)$.

Given a net $N = \langle P, T, pre, post, l, M_{init} \rangle$, a *marking* of N is any $M \in \mathcal{M}(P)$. In a graphical representation a marking M is displayed by putting $M(p)$ tokens in each place p.

Given a marking M, we say that $t \in T$ is *firable in* M, written $M \xrightarrow{t}$, if $pre(t) \subseteq M$. If t is firable in M, firing it yields a new marking $M' \stackrel{\text{def}}{=} M - pre(t) + post(t)$, written $M \xrightarrow{t} M'$.

We write $M \xrightarrow{a} M'$ when $M \xrightarrow{t} M'$ for some t s.t. $l(t) = a$, and $M \rightarrow M'$ when $M \xrightarrow{t} M'$ for some t. $\rightarrow^*$ denotes the reflexive and transitive closure of this last relation.

Two nets N_1, N_2 are *isomorphic*, written $N_1 \equiv N_2$, when they only differ by the names of their places and transitions.

Two transitions $t, t' \in T_N$ are *equivalent in* N, written $t \equiv_N t'$, if they have same *pre*, *post* and labeling. Then $M \xrightarrow{t} M'$ iff $M \xrightarrow{t'} M'$. A *simple* net is a net where no two transitions are equivalent. By identifying equivalent transitions, it is possible to transform any net N into a simple net, denoted $simple(N)$.

We say that N_1 and N_2 are *simply isomorphic*, written $N_1 \equiv_s N_2$, when $simple(N_1) \equiv simple(N_2)$.

Given $N = \langle P, T, pre, post, l, M_{init} \rangle$, $mark(N) \stackrel{\text{def}}{=} \{M \in \mathcal{M}(P) \mid M_{init} \rightarrow^* M\}$ is the set of *reachable markings* of N. We define by

$$places(N) \stackrel{\text{def}}{=} \{p \in P \mid p \in M \text{ for some } M \in mark(N)\}$$
$$trans(N) \stackrel{\text{def}}{=} \{t \in T \mid M_{init} \rightarrow^* M \xrightarrow{t} \text{ for some } M\}$$

the set of *reachable places* and of *reachable transitions*.

A *reachable* net is a net where all places are reachable. It is possible to prune a net N into a reachable net:

$$\| N \| \stackrel{\text{def}}{=} \langle places(N), trans(N), pre', post', l', M_{init} \rangle$$

where pre', $post'$ and l' are pre, $post$ and l restricted to $trans(N)$.

A *safe* net is a net N where, for any $t \in T_N$, $pre_N(t)$ and $post_N(t)$ are proper sets, and where all reachable markings are proper sets: in a safe net, at any time, there can only be 0 or 1 token in any place.

Finally, it is possible to see classical transition systems as nets. An *S-graph* is a net N where $init(N)$, all $pre_N(t)$ and all $post_N(t)$ are singletons [NT84]. S-graphs are safe.

It is possible to give a polished theory by only considering reachable simple nets. We preferred to consider the general case and only introduce restrictions when they are required. This gives a better understanding of the situation.

3 Place bisimulation

Given N_1, N_2 two nets with $N_i = \langle P_i, T_i, pre_i, post_i, l_i, M_{init,i}\rangle$, a relation $B \subseteq P_1 \times P_2$ may be lifted to yield a relation $\overline{B} \subseteq \mathcal{M}(P_1) \times \mathcal{M}(P_2)$ between markings, defined by

$$M_1 \overline{B} M_2 \stackrel{\text{def}}{\Leftrightarrow} \begin{cases} \text{there exists } \{(p_1,q_1),\ldots,(p_n,q_n)\} \in \mathcal{M}(B) \\ \text{such that } M_1 = \{p_1,\ldots,p_n\} \text{ and } M_2 = \{q_1,\ldots,q_n\} \end{cases} \quad (1)$$

Note that this ensures $\emptyset \overline{B} \emptyset$ and that $\sum_i M_i \ \overline{B} \ \sum_i M'_i$ when $M_i \overline{B} M'_i$ for any i. We also have

$$\overline{Id_P} = Id_{\mathcal{M}(P)} \qquad \overline{B \circ B'} = \overline{B} \circ \overline{B'} \qquad \overline{B^{-1}} = \overline{B}^{-1} \qquad \overline{B} \subseteq \overline{B'} \text{ iff } B \subseteq B' \quad (2)$$

$\overline{B}$ is extended to transitions with the following

$$t_1 \overline{B} t_2 \stackrel{\text{def}}{\Leftrightarrow} \begin{cases} pre_1(t_1) \overline{B} pre_2(t_2) \\ \wedge\ post_1(t_1) \overline{B} post_2(t_2) \\ \wedge\ l_1(t_1) = l_2(t_2) \end{cases}$$

which also satisfies (2).

We now define

Definition 2 *Given two labeled nets N_1, N_2 with $N_i = \langle P_i, T_i, pre_i, post_i, l_i, M_{init,i}\rangle$, a relation $B \subseteq P_1 \times P_2$ is a (place) bisimulation between N_1 and N_2 iff*

1. *$M_{init,1} \overline{B} M_{init,2}$, and*
2. *for all $M_1 \in \mathcal{M}(P_1)$, $M_2 \in \mathcal{M}(P_2)$ s.t. $M_1 \overline{B} M_2$*
   - *for all steps $M_1 \stackrel{t_1}{\rightarrow} M'_1$ in N_1, there exists a $M_2 \stackrel{t_2}{\rightarrow} M'_2$ in N_2 s.t. $t_1 \overline{B} t_2$ and $M'_1 \overline{B} M'_2$,*
   - *and, reciprocally, for all steps $M_2 \stackrel{t_2}{\rightarrow} M'_2$ in N_2, there exists a $M_1 \stackrel{t_1}{\rightarrow} M'_1$ in N_1 s.t. $t_1 \overline{B} t_2$ and $M'_1 \overline{B} M'_2$.*

(Point 2 above is called the *transfer property*.) We write $N_1 \approx_B N_2$ when B is a place bisimulation between N_1 and N_2, and $N_1 \approx N_2$ when $N_1 \approx_B N_2$ for some B. In a graphical representation, the pairs (p,q) of a bisimulation are represented by dashed lines between places.

With this notion of bisimulation, a marking M_2 simulates another marking M_1 if it has the same number of tokens in equivalent places and if any step of M_1 can be reproduced in an equivalent way. Here, reproducing a step $M_1 \stackrel{t}{\rightarrow} M'_1$ in an equivalent way means moving tokens from places equivalent to $pre(t)$ to places equivalent to $post(t)$, reaching a marking equivalent to M'_1.

Proposition 1 (a) *$N \approx_{Id_{P_N}} N$,*

(b) *$N \approx_B N'$ implies $N' \approx_{B^{-1}} N$,*

(c) *$N_1 \approx_B N_2$ and $N_2 \approx_{B'} N_3$ imply $N_1 \approx_{B' \circ B} N_3$,*

(d) *$N \approx_{Id_{P_N}} simple(N)$,*

(e) $\approx$ *is an equivalence relation on labeled nets which contains* $\equiv_s$.

(a), (b), (c) are direct consequences of (2). With (d), they entail (e).

We give now a lemma we shall need in the following

Lemma 1 $N_1 \equiv_s N_2$ *iff* $N_1 \approx_B N_2$ *for some bijective* B.

Proof Use B to build an isomorphism. □

4 An analysis of Olderog's definition

In [Old89b], Olderog uses another way of lifting B from places to markings. Instead of our (1), he only considers safe nets (i.e. nets where all markings are proper sets) and defines:

$$M_1 \overline{B} M_2 \stackrel{\text{def}}{\Leftrightarrow} B \cap (M_1 \times M_2) \text{ is a bijection}$$

First note that this is only meaningful for safe nets, and that when two safe nets are bisimilar in Olderog's sense (written $N_1 \approx^o N_2$), they are bisimilar in our sense.

The problem with this definition is that it ensures only

$$M \overline{B' \circ B} M' \;\Rightarrow\; M(\overline{B'} \circ \overline{B})M'$$

and not the opposite implication. Compare this with (2). The consequence is that $\approx^o$, as defined in [Old89b], is not transitive and thus *is not an equivalence relation* on nets. Figure 1 contains an example of a possible situation.

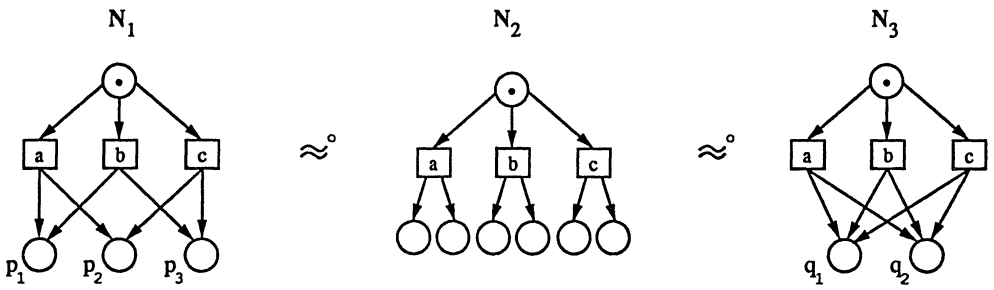


Figure 1: Transitivity fails.

Here $N_1 \approx^o N_2$ and $N_2 \approx^o N_3$. We do not have $N_1 \approx^o N_3$: indeed, assume $N_1 \approx^o_B N_3$ for some B. Then, given any two places $p \neq p'$ in $\{p_1, p_2, p_3\}$, we must have $\{p, p'\}\overline{B}\{q_1, q_2\}$. Then B must relate different p_i's to a same q_j, which forbids the bijective relation required in $\{p, p'\}\overline{B}\{q_1, q_2\}$.

Note that N_1 and N_3 have isomorphic unfoldings, so that Olderog's definition is not compatible with unfoldings.

Finally, observe that with our definition $N_1 \approx N_3$ does hold.

5 Examples and comments

We consider a few examples of place-bisimilar nets. This gives some understanding of which nets are equivalent according to our definition. In these examples, we concentrate on specific features of place bisimulation and will not illustrate the basic behavior of branching-time equivalences as opposed to linear-time equivalences (for which we refer to [vG90]). However, we compare place bisimulation with $\leftrightarrows$, the classical bisimulation (between global markings of safe nets) of [vGV87] to which we refer for a definition.

First consider Figure 2. Place bisimulation respects the number of tokens in markings, so that $N_1 \not\approx N_2$

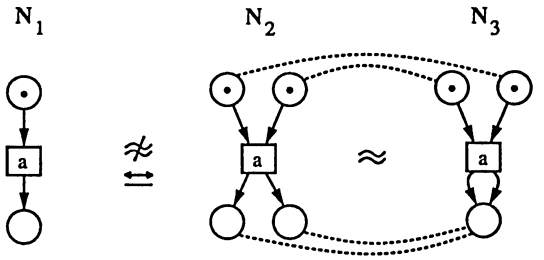


Figure 2: $\approx$ preserves the number of tokens.

while $N_2 \approx N_3$. Note that $N_1 \leftrightarrows N_2$. Note that the number of places needs not be respected as several tokens may appear in a same place. Of course, for safe nets, the number of places is preserved across bisimilar markings.

The examples in Figure 3 show that not only the number of tokens is preserved, but also the way they move. In N_1, one token moves, while two tokens move in N_2. So that $N_1 \not\approx N_2 \approx N_3$.

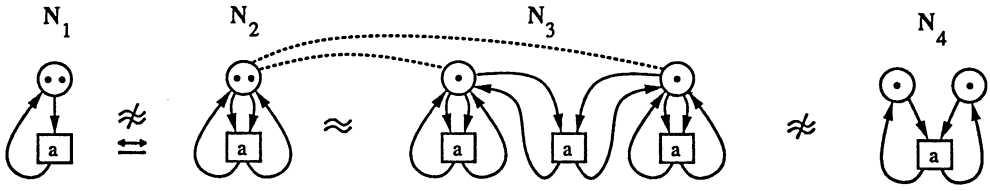


Figure 3: $\approx$ preserves the way tokens move.

Remark 1 *$N_3 \not\approx N_4$ in Figure 3 because if we consider two markings M_3 and M_4 of (resp.) N_3 and N_4 having the form $\{p,p\}$, then $M_3 \xrightarrow{a} M_3$ in N_3 while in N_4 no transition is allowed from M_4. Of course, M_3 and M_4* are not reachable markings *but Definition 2 requires that the transfer property holds for any $M_1 \in \mathcal{M}(P_1)$ and $M_2 \in \mathcal{M}(P_2)$. This is a crucial point of our definition which cannot be modified without introducing inconsistencies. See section 8.1 for details.*

Figure 4 contains simple examples of place bisimulations. Observe how unwinding loops is possible.

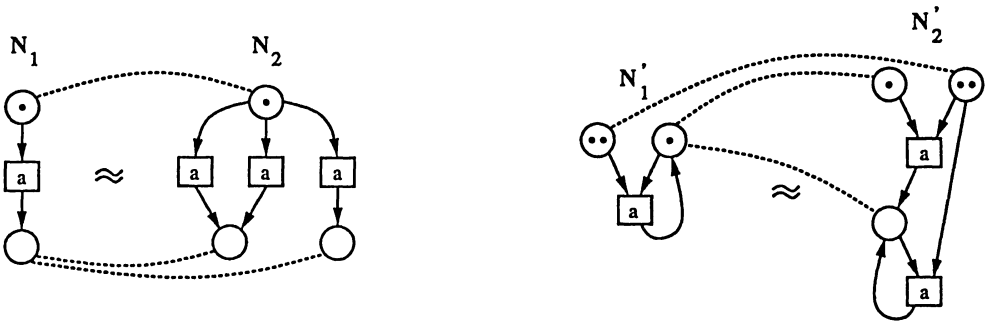


Figure 4: $\approx$ allows sharing and unwindings of loops.

These examples suggest that $\approx$ is stronger than, but not completely unrelated to, $\underline{\leftrightarrow}$. Indeed, we have:

Proposition 2 *For any nets N_1 and N_2:*
1. $N_1 \approx N_2$ implies $N_1 \underline{\leftrightarrow} N_2$.
2. The reciprocal holds for S-graphs.

The result holds for the different concurrent bisimulations $\underline{\leftrightarrow}_c$ and $\underline{\leftrightarrow}_{pom}$ that are considered in [vGV87]. The proof is rather technical (at least for $\underline{\leftrightarrow}_{pom}$). As it requires that the definition of $\underline{\leftrightarrow}_{pom}$ be recalled, we did not include it here. It appears in [ABS91].

Figure 5 exhibits one peculiarity of place bisimulation. N_1 and N_2 are not place-bisimilar. Any

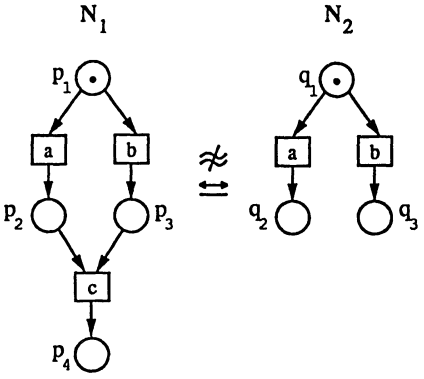


Figure 5: $N \not\approx ||N||$.

bisimulation should relate p_2 to q_2 and p_3 to q_3. But the marking $\{q_2, q_3\}$ for N_2 cannot reproduce the step allowed in the marking $\{p_2, p_3\}$ of N_1. This shows that N and $||N||$ need not be bisimilar.

The reason is that p_2 and q_2 (or p_3 and q_3) *are not equivalent places.* p_2 can be involved in a c step. Of course, in N_1, this c step is never allowed, but this does not depends on p_2 itself. The marking $\{p_2\}$ in N_1 is deadlocked, while $\{q_2\}$ in N_2 is terminated.

We tried to develop a variant of our definition that would identify N and $\| N \|$. This appeared impossible. The obvious tentative is to require that a bisimulation only preserve moves *from reachable markings*, writing

$$\textit{For all } M_1 \in mark(N_1) \textit{ and } M_2 \in mark(N_2) \textit{ s.t. } M_1 \overline{B} M_2 \ \ldots$$

in Definition 2. This certainly ensures $N \approx \| N \|$ but transitivity of $\approx$ is lost again ! See section 8.1 for details.

A place in a net can be reached by different behaviors. A bisimulation between places cannot take into account global information that depends on the particular history that allowed a place to be reached. With classical transition systems this situation does not happen, for a state uniquely determines its own possible futures.

It appears that this distinction of terminated vs. deadlocked place is the reason why nets having isomorphic unfoldings ($N_1 \equiv_{occ} N_2$ in the notation of [vGV87]) needs not be place-bisimilar. We already showed (through examples) that unwinding loops is compatible with bisimulations of places and unfolding is just unwinding + pruning. Another indication is given by point 2 of Proposition 2: in S-graphs, a place does not have to synchronize with other places.

Finally, consider Figure 6 which shows how *statically unreachable places* (places that cannot be connected by directed edges from one place of M_{init}) can be abstracted from.

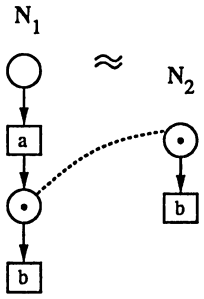


Figure 6: $\approx$ and statically unreachable places.

6 Largest bisimulations

In classical bisimulation theory for transition systems (e.g. [BK89]), the union of arbitrary bisimulations is still a bisimulation. This ensures that if two automata are bisimilar, there exists a largest bisimulation between them.

The situation is more complicated with nets: assume $N_1 \approx_{B_1} N_2$ and $N_1 \approx_{B_2} N_2$. Then $N_1 \approx_{B_1 \cup B_2} N_2$ needs not be true. Even worse, we need not have $N_1 \approx_B N_2$ for some bisimulation containing B_1 and

B_2. Figure 7 shows an example. Here $B = \{(p_1, p_4), (p_2, p_3), (p_3, p_2), (p_4, p_1), (p_5, p_5)\}$ is a bisimulation

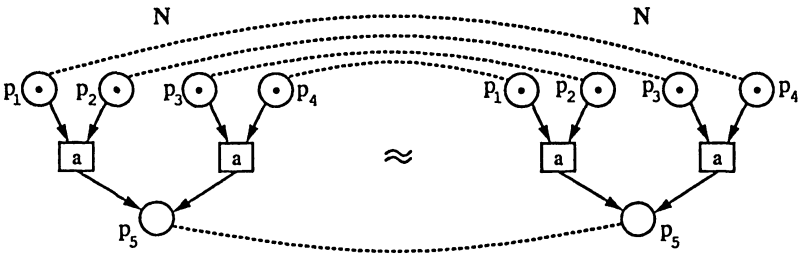


Figure 7: Bisimulations need not have upper bounds.

between N and itself. Id_P is another bisimulation, but no bisimulation includes $Id_P \cup B$.

To investigate these problems, we mainly work with bisimulations between a net and itself. Given a net N, an *autobisimulation* of N is a bisimulation between N and itself. An *equibisimulation* of N is an autobisimulation which is also an equivalence relation on P_N. See the previous example and notice that autobisimulations need not be equibisimulations.

The key idea in this section is to consider *reflexive* autobisimulations, of which equibisimulations are a special case. Reflexivity allows to abstract from initial markings:

Proposition 3 *Two nets differing only in their initial marking have the same reflexive autobisimulations.*

Proof See [ABS91]. □

Proposition 4 *If B_1 and B_2 are two reflexive autobisimulations of a net N, then the symmetric and transitive closure of $B_1 \cup B_2$ is an equibisimulation of N.*

The proof needs a lemma:

Lemma 2 *When B_1 and B_2 are reflexive relations*

$$\overline{(B_1 \cup B_2 \cup B_1^{-1} \cup B_2^{-1})^*} = (\overline{B_1} \cup \overline{B_2} \cup \overline{B_1}^{-1} \cup \overline{B_2}^{-1})^*$$

Proof The "$\supseteq$" direction is obvious. We only prove the other inclusion.

Write B for $(B_1 \cup B_2 \cup B_1^{-1} \cup B_2^{-1})^*$. If $M_1 \overline{B} M_2$, we know that M_1 is some $\{p_1, \ldots, p_n\}$ and M_2 some $\{q_1, \ldots, q_n\}$ with $p_i B q_i$ for $i = 1, \ldots, n$. Given the definition of B, $p_i B q_i$ means that there exists a sequence $\sigma_i = (p_i^0, p_i^1, \ldots, p_i^{k_i})$ such that $p_i = p_i^0$, $q_i = p_i^{k_i}$ and, for $j = 1, \ldots, k_i$, $p_i^{j-1} R_i^j p_i^j$ where R_i^j is one of B_1, B_2, B_1^{-1} or B_2^{-1}. These sequences need not have the same length, but as B_1 and B_2 are reflexive, it is always possible to insert duplications in a sequence such that all σ_i's have the same

length. Similarly, it is possible with such insertions to ensure that, for any j, $R_1^j, \ldots, R_n^j$ are the same relation R^j (this may further lengthen the σ_i's to some common length k). When this is done, the markings $M^0, \ldots, M^k$ defined by $M^j = \{p_1^j, \ldots, p_n^j\}$ satisfy $M_1 = M^0$, $M_2 = M^k$ and $M^{j-1}\overline{R^j}M^j$ for $j = 1, \ldots, k$. This ensures $M_1(\overline{R^k} \circ \cdots \circ \overline{R^1})M_2$, and then $M_1(\overline{B_1} \cup \overline{B_2} \cup \overline{B_1}^{-1} \cup \overline{B_2}^{-1})^* M_2$ as each R^j is one of B_1, B_2, B_1^{-1} or B_2^{-1}. □

The same lemma holds for $\overline{B}$ considered between transitions.

Proof (of Proposition 4) Assume $M_1\overline{B}M_2$ (with B as defined in the previous proof.) The lemma tells us that $M_1\overline{R}M_2$ for some R of the form $R^k \circ \ldots \circ R^1$ (which generally depends on M_1 and M_2). R is a bisimulation as it is a composition of B_1's, B_2^{-1}'s, ... which are bisimulations. Then any $M_1 \xrightarrow{t_1} M_1'$, can be simulated by some $M_2 \xrightarrow{t_2} M_2'$ s.t. $M_1'\overline{R}M_2'$and $t_1\overline{R}t_2$, which entails $M_1'\overline{B}M_2'$ and $t_1\overline{B}t_2$ (as $R \subseteq B$). The same reasoning holds for steps $M_2 \xrightarrow{t_2} M_2'$.

Finally B is an autobisimulation, and then an equibisimulation by construction. □

The result applies to arbitrary (not necessarily finite) families of reflexive autobisimulations. The example in Figure 7 shows that reflexivity is required.

Proposition 5 *Given a net N, the* largest equibisimulation *over N is*

$$B(N) \stackrel{\text{def}}{=} \bigcup\{B \mid N \approx_B N \text{ and } B \text{ reflexive}\}$$

Proof Standard from the previous results. □

$B(N)$ is the canonical equibisimulation of N.

We saw that unions of bisimulations need not be bisimulations. We shall need to know that they exist in special cases:

Lemma 3 *If $N_1 \approx_{B_j} N_2$ for a directed family $(B_j)_{j\in J}$ of bisimulations, then $N_1 \approx_B N_2$ with $B = \bigcup_{j\in J} B_j$.*

Proof Assume $N_i = \langle P_i, T_i, pre_i, post_i, l_i, M_{init,i}\rangle$ for $i = 1, 2$. First, we clearly have $M_{init,1}\overline{B}M_{init,2}$. Consider now $M_1\overline{B}M_2$. This means that M_1 is some $\{p_1, \ldots, p_n\}$ and M_2 some $\{q_1, \ldots, q_n\}$ with p_iBq_i for $i = 1, \ldots, n$. Then, for any i, there exists some index $j_i \in J$ s.t. $p_iB_{j_i}q_i$. The B_j's being a directed family, there is some B_k containing all B_{j_i}'s, so that $M_1\overline{B_k}M_2$. If now $M_1 \xrightarrow{t_1} M_1'$, then we can find some $M_2 \xrightarrow{t_2} M_2'$ with $t_1\overline{B_k}t_2$ and $M_1'\overline{B_k}M_2'$. This implies $t_1\overline{B}t_2$ and $M_1'\overline{B}M_2'$, and a symmetric argument concludes the proof. □

We need one more lemma about reachable places:

Lemma 4 *If $N \approx_B N'$, then $places(N) \subseteq dom(B)$ and $places(N') \subseteq codom(B)$.*

Proof See [ABS91]. □

The useful corollary is

Lemma 5 *If N_1 and N_2 are bisimilar reachable nets, then $N_1 \approx_B N_2$ for some B such that $B = B \circ B^{-1} \circ B$.*

Proof Assume $N_1 \approx_B N_2$ and write P_i for $places(N_i)$. First note that, for any n, $B_n \stackrel{\text{def}}{=} B \circ (B^{-1} \circ B)^n$ is a bisimulation between N_1 and N_2 (Proposition 1). As N_1 and N_2 are reachable, $dom(B) = P_1$ and $codom(B) = P_2$ (Lemma 4). This entails $Id_{P_1} \subseteq B^{-1} \circ B$, so that $B_n \subseteq B_{n+1}$. Thus we have an increasing family of bisimulations, with l.u.b. $B_\omega \stackrel{\text{def}}{=} \bigcup_{n=0,1,\ldots} B_n$.

B_ω is the required bisimulation: it is $B \circ (B^{-1} \circ B)^*$, entailing $B_\omega = B_\omega \circ B_\omega^{-1} \circ B_\omega$. Note that $B_\omega^{-1} \circ B_\omega$ is an equivalence relation on P_1. □

7 Quotient nets

We investigate quotient nets and how they are related to bisimulations, looking for canonical representatives of equivalence classes of bisimilar nets.

Consider an equibisimulation B of some net N. As B is an equivalence relation, it is possible to define a quotient net $N/_B$.

If $N = \langle P, T, pre, post, l, M_{init} \rangle$, then $P/_B$ is $\{[p]_B \mid p \in P\}$ where $[p]_B$ denotes the equivalence class of p. This extends to markings by $\{p_1, \ldots, p_n\}/_B \stackrel{\text{def}}{=} \{[p_1]_B, \ldots, [p_n]_B\}$, verifying $M/_B \in \mathcal{M}(P/_B)$. Now, $N/_B$ is given by the following:

$$N/_B \stackrel{\text{def}}{=} \langle P/_B, T, pre/_B, post/_B, l, M_{init}/_B \rangle$$

where $(pre/_B)(t) \stackrel{\text{def}}{=} (pre(t))/_B$ and similarly for $post$. We write $[\,]_B$ to denote the function that maps any place p to its equivalence class $[p]_B$.

We already saw several examples of such quotients. In Figure 1, N_1 and N_3 are quotients of N_2. In Figure 2, N_3 is a quotient of N_2, while N_2 is (simply isomorphic to) a quotient of N_3 in Figure 3.

Lemma 6 *If B is an equibisimulation of N, then $N \approx_{[\,]_B} N/_B$.*

Proof See [ABS91]. □

In some sense $N/_B$ is smaller than N. This suggests the following definition:

Definition 3 $N_1 \preceq N_2 \stackrel{\text{def}}{\Leftrightarrow} N_1 \equiv_s N_2/_B$ *for some equibisimulation B.*

Note that $N_1 \equiv_s N_2$ implies $N_1 \preceq N_2$.

A useful characterization is given by the following

Lemma 7 *$N_2 \preceq N_1$ iff $N_1 \approx_B N_2$ for some surjective mapping $B : P_1 \to P_2$.*

Proof "$\Leftarrow$": If $N_1 \approx_B N_2$, then $B^{-1} \circ B$ is an autobisimulation of N_1. If B is a surjective mapping, $B^{-1} \circ B$ is an equivalence relation, so that we can quotient N_1. We write N_1' for $N_1/_{B^{-1} \circ B}$. We have

$$N_2 \approx_{B^{-1}} N_1 \approx_{[\,]_{B^{-1} \circ B}} N_1'$$

Then, writing h for $[\,]_{B^{-1} \circ B} \circ B^{-1}$, we have $N_2 \approx_h N_1'$. It is easy to check that h is a bijection, so that $N_1' \equiv_s N_2$ thanks to Lemma 1.

"$\Rightarrow$": If $N_2 \preceq N_1$, then by definition $N_1/_B \equiv_s N_2$ for some equibisimulation B. Then Lemma 1 implies that $N_1/_B \approx_{B'} N_2$ for some bijective B', so that we have

$$N_1 \approx_{[\,]_B} N_1/_B \approx_{B'} N_2$$

We only have to remark that $B' \circ [\,]_B$ is a surjective mapping to conclude the proof. □

The composition of surjective mappings is a surjective mapping, so that an immediate corollary is

Proposition 6 *$\preceq$ is a partial ordering on nets.*

We also have

Lemma 8 *If B_1 and B_2 are two equibisimulations on N, then $B_1 \subseteq B_2$ implies $N/_{B_2} \preceq N/_{B_1}$.*

Proof We have $N/_{B_1} \approx_{[\,]_{B_1}{}^{-1}} N \approx_{[\,]_{B_2}} N/_{B_2}$ by Lemma 6. Then $N/_{B_1} \approx_B N/_{B_2}$ with $B = [\,]_{B_2} \circ [\,]_{B_1}{}^{-1}$. Now, $B_1 \subseteq B_2$ implies that B is a surjective mapping. □

with the important corollary:

Proposition 7 *If $N_1 \preceq N$ and $N_2 \preceq N$, there exists some net N' such that $N' \preceq N_1$ and $N' \preceq N_2$.*

Proof $N_i \preceq N$ implies that $N_i \equiv_s N/_{B_i}$ for some B_i. Writing B for $(B_1 \cup B_2)^*$, we know from Proposition 4 that B is also an equibisimulation of N, and as $B_i \subseteq B$, $N' \stackrel{\text{def}}{=} N/_B \preceq N/_{B_i} \equiv_s N_i$. □

With largest equibisimulations come "smallest" quotients:

Definition 4 *For any net N, the* canonical quotient *of N is $N/_{\approx} \stackrel{\text{def}}{=} N/_{B(N)}$*

These quotients are canonical representatives in the following sense:

Proposition 8 *For two reachable nets N_1 and N_2, $N_1 \approx N_2$ iff $N_1/_{\approx} \equiv_s N_2/_{\approx}$*

which gives us canonical representatives of reachable nets, unique up to $\equiv_s$.

Proof The "$\Leftarrow$" direction is obvious. The other direction is done in two steps.

Consider two bisimilar reachable nets N_1 and N_2 and write P_i for $places(N_i)$. Lemma 5 allows us to assume $N_1 \approx_B N_2$ with $B = B \circ B^{-1} \circ B$ and, writing B_1 for $B^{-1} \circ B$ and B_2 for $B \circ B^{-1}$, each B_i an equivalence relation on P_i. The B_i's can be used to quotient the N_i's, yielding

$$N_1/_{B_1} \approx_{[\,]_{B_1}{}^{-1}} N_1 \approx_B N_2 \approx_{[\,]_{B_2}} N_2/_{B_2}$$

Remark that $[\ ]_{B_2} \circ B \circ [\,]_{B_1}{}^{-1}$ is a surjective mapping from $P_1/_{B_1}$ to $P_2/_{B_2}$. As it is clearly a bisimulation from $N_1/_{B_1}$ to $N_2/_{B_2}$, Lemma 7 implies that $N_1/_{B_1} \preceq N_2/_{B_2}$. A similar reasoning shows that $N_2/_{B_2} \preceq N_1/_{B_1}$, so that $N_1/_{B_1}$ and $N_2/_{B_2}$ have a common quotient, thanks to Proposition 7. Finally N_1 and N_2 also have a common quotient.

But if $N_1 \approx N_2$, then also $N_1/_{\approx} \approx N_2/_{\approx}$ and, the quotient of a reachable net being reachable, the same reasoning applies, allowing to conclude that $N_1/_{\approx}$ and $N_2/_{\approx}$ have a common quotient N. But, by construction, $N_i/_{\approx}$ cannot be quotiented any further, therefore N must be $N_1/_{\approx}$ itself (or some $\equiv_s$-variant). Finally $N_1/_{\approx} \equiv_s N_2/_{\approx}$. □

There remains one natural question about quotients. [vGV87, Old89b] use safe nets while we allowed arbitrary Petri nets. The canonical quotient of a safe net needs not be safe. Is it possible to develop a notion of a *safe quotient* of a safe net ? This turns out to be impossible:

Remark 2 *If $N_1 \preceq N$ and $N_2 \preceq N$ are three safe nets, there is in general no safe common quotient of N_1 and N_2.*

Figure 8 shows an example. In Figure 8, N can be "safely" quotiented by identifying p_1 and p_2, or p_2

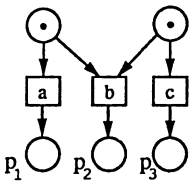


Figure 8: Safe quotients need not exist

and p_3. Note that these two quotients are not isomorphic. No common quotient is safe.

8 Variants of place bisimulations

We investigate other possible attempts at defining bisimulations between places. It appears that Definition 2 is the only viable one.

8.1 Reachable markings

We already mentioned that the main problem with our definition is that a net N needs not be bisimilar to the pruned $||N||$. One possible solution is to modify Definition 2 so that it only considers reachable markings.

Definition 5 *Given two nets N_1, N_2 with $N_i = \langle P_i, T_i, pre_i, post_i, l_i, M_{init,i}\rangle$, a relation $B \subseteq P_1 \times P_2$ is a bisimulation between N_1 and N_2 iff*

1. *$M_{init,1} \overline{B} M_{init,2}$, and*
2. *for all (reachable) $M_1 \in mark(N_1)$, $M_2 \in mark(N_2)$ s.t. $M_1 \overline{B} M_2$*
   - *for all steps $M_1 \xrightarrow{t_1} M_1'$ in N_1, there exists a $M_2 \xrightarrow{t_2} M_2'$ in N_2 s.t. $t_1 \overline{B} t_2$ and $M_1' \overline{B} M_2'$,*
   - *and, reciprocally, for all steps $M_2 \xrightarrow{t_2} M_2'$ in N_2, there exists a $M_1 \xrightarrow{t_1} M_1'$ in N_1 s.t. $t_1 \overline{B} t_2$ and $M_1' \overline{B} M_2'$.*

Denote $\approx^r$ the relation defined this way: we clearly have $N \approx^r || N ||$ for all N. Unfortunately, Definition 5 does not yield a transitive relation. Figure 9 shows an example where $N_1 \approx^r N_2 \approx^r N_3$ while $N_1 \not\approx^r N_3$.

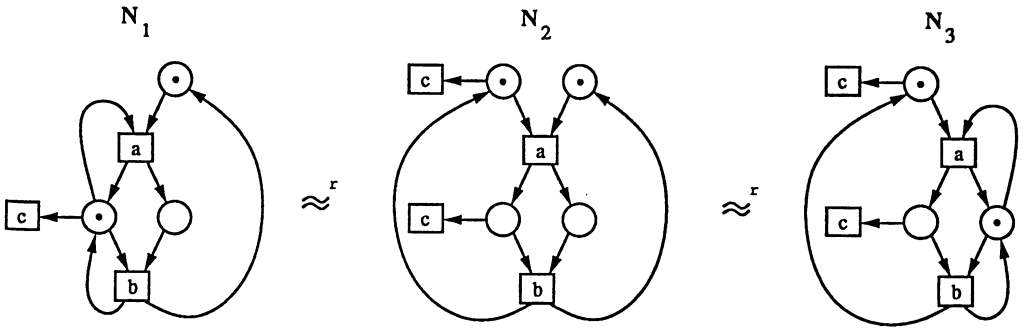


Figure 9: Transitivity lost again.

8.2 Bisimilar transitions

The next possible variant of Definition 2 is not aimed at solving the "$N \not\approx ||N||$" problem.

One strong requirement of Definition 2 is that, when $M_1 \overline{B} M_2$, a step $M_1 \xrightarrow{t_1} M_1'$ can only be imitated by a step $M_2 \xrightarrow{t_2} M_2'$ with $t_1 \overline{B} t_2$. As we already mentioned, this entails that *the same number of tokens move from bisimilar places to bisimilar places.* It is possible to relax this requirement and only ask for $l_1(t_1) = l_2(t_2)$, giving:

Definition 6 *Given two nets N_1, N_2 with $N_i = \langle P_i, T_i, pre_i, post_i, l_i, M_{init,i}\rangle$, a relation $B \subseteq P_1 \times P_2$ is a bisimulation between N_1 and N_2 iff*

1. $M_{init,1} \overline{B} M_{init,2}$, *and*
2. *for all $M_1 \in \mathcal{M}(P_1)$, $M_2 \in \mathcal{M}(P_2)$ s.t. $M_1 \overline{B} M_2$*
   - *for all steps $M_1 \xrightarrow{t_1} M_1'$ in N_1, there exists a $M_2 \xrightarrow{t_2} M_2'$ in N_2 s.t. $l_1(t_1) = l_2(t_2)$ and $M_1' \overline{B} M_2'$,*
   - *and, reciprocally, for all steps $M_2 \xrightarrow{t_2} M_2'$ in N_2, there exists a $M_1 \xrightarrow{t_1} M_1'$ in N_1 s.t. $l_1(t_1) = l_2(t_2)$ and $M_1' \overline{B} M_2'$.*

Remark 3 *Observe that, in this definition, the requirement that $M_1' \overline{B} M_2'$ is no longer superfluous.*

Denote $\approx^t$ the relation defined this way. $\approx^t$ is an equivalence relation. It is slightly weaker than our $\approx$:

Proposition 9 [ABS91]. *For any nets N_1 and N_2:*
1. $N_1 \approx N_2$ implies $N_1 \approx^t N_2$.
2. $N_1 \approx^t N_2$ implies $N_1 \leftrightarrow_c N_2$.

Cursory exploration indicates that $\approx^t$ deserves to be studied in its own right. We did not consider it in this paper because it is not compatible with pomset bisimulation. Figure 10 shows a counter-

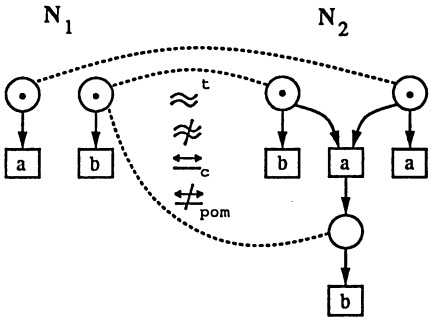


Figure 10: $\approx^t$ and $\leftrightarrow_{pom}$.

example[2]. Here N_2 can perform the pomset "a followed by b" while N_1 can only perform "a and b in parallel".

9 Conclusion

The major contributions of this paper are twofold. We first uncovered subtle problems with previous definitions of bisimulation between places of Petri nets. Then we proposed a corrected definition and investigated it.

[2] used in [vGV87] to show that $\leftrightarrow_{pom}$ is stronger than $\leftrightarrow_c$.

These investigations clearly show that the idea of bisimulation between places is very interesting and original. We proved that there exists a notion of "equivalent places" where equivalent places can be identified in a very real sense.

Let us mention how we envision practical applications of the theory. As indicated in the introduction, we tried to make as few restrictions as possible in order to get a better understanding of which property depends on which requirement. As a consequence, the equivalence we studied is rather crude and would probably not be the best suited as e.g. a semantic equivalence for a parallel language. However, it is possible to define derived notions, e.g. "two nets N_1 and N_2 are equivalent if $||N_1|| \approx ||N_2||$". We can already state several properties of such an equivalence.

We also believe that place bisimulation and the possibility of collapsing equivalent places (Lemma 6) has real applications for languages implemented with some kind of finite interpreted Petri nets. This is because collapsing places allows to share code. There, it is important to reason at the level of places (which are static objects and correspond to code) and not at the level of markings (which are dynamic objects). Of course, this requires our theory to be adapted to interpreted nets, but we believe that this is feasible.

Obviously, even at the theoretical level, this paper does not close the whole subject. Much work remains to be done. Let us list a few important problems we did not consider in this paper:

- Variants and/or restrictions of our definition must be investigated more deeply. This is partly done in [ABS91]. There are several possible notions of exactly when two places are bisimilar.
- The theory must be extended to nets with silent actions. We are working on this.
- The comparison between bisimulation of places and bisimulations of markings should go beyond Proposition 2. There already exists many more bisimulations on markings [RT88, BDKP90, Dev90], etc.
- Congruence properties should be investigated, as in [Old89a]. Unless silent actions are dealt with, this is probably more interesting for operators like e.g. refinements [Dev90] than for the *CCS*-like nets combinators investigated in [vGV87].
- A semantic equivalence on nets can be used to give a semantics to *CCS*-like languages [Gol88, DDM88, vGV87, Old87]. "Bisimulation on places" should be compared with other distributed semantics.

Many other questions could be mentioned but it is more difficult to assess their interest. For example, the following may well be specific to place bisimulation:

Proposition 10 [ABS91] *It is decidable whether two finite nets N_1 and N_2 are place bisimilar.*

At the moment, we do not know if $\leftrightarrows$ is a decidable relation between finite non-necessarily safe nets.

10 Acknowledgements

We thank Sophie Pinchinat for her useful comments and suggestions.

References

[ABS91] C. Autant, Z. Belmesk, and Ph. Schnoebelen. *Strong Bisimilarity on Nets Revisited.* Research Report, LIFIA-IMAG, Grenoble, 1991.

[BDKP90] E. Best, R. Devillers, A. Kiehn, and L. Pomello. Concurrent bisimulations in Petri nets. September 1990. To appear in Acta Informatica.

[BK89] J. A. Bergstra and J. W. Klop. Process theory based on bisimulation semantics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout, LNCS 354*, pages 50–122, Springer-Verlag, 1989.

[DDM88] P. Degano, R. De Nicola, and U. Montanari. A distributed operational semantics for CCS based on condition/event systems. *Acta Informatica*, 26:59–91, 1988.

[Dev90] R. Devillers. *Maximality Preserving Bisimulation.* Tech. Report LIT-214, Lab. Informatique Théorique, Université Libre de Bruxelles, March 1990.

[Gol88] U. Goltz. On representing CCS programs by finite Petri nets. In *Proc. Math. Found. Computer Science, LNCS 324*, pages 339–350, Springer-Verlag, 1988.

[NT84] M. Nielsen and P. S. Thiagarajan. Degrees of non-determinism and concurrency: a Petri net view. In *Proc. 4th Conf. on Foundations of Software Technology and Theoretical Computer Science, Bangalore, India, LNCS 181*, pages 89–117, Springer-Verlag, December 1984.

[Old87] E.-R. Olderog. Operational Petri net semantics for CCSP. In *Advances in Petri Nets 1987, LNCS 266*, pages 196–223, Springer-Verlag, 1987.

[Old89a] E.-R. Olderog. Nets, terms and formulas: three views of concurrent processes and their relationship. Habilitationsschrift, Christian-Albrechts-Univ. Kiel, July 1989.

[Old89b] E.-R. Olderog. Strong bisimilarity on nets: a new concept for comparing net semantics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout, LNCS 354*, pages 549–573, Springer-Verlag, 1989.

[Par81] D. Park. Concurrency and automata on infinite sequences. In *Proc. 5th GI Conf. on Th. Comp. Sci., LNCS 104*, pages 167–183, Springer-Verlag, March 1981.

[Pom85] L. Pomello. Some equivalence notions for concurrent systems. An overview. In *Advances in Petri Nets 1985, LNCS 222*, pages 381–400, Springer-Verlag, 1985.

[Rei85] W. Reisig. *Petri Nets. An Introduction.* Volume 4 of *EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, 1985.

[RT88] A. Rabinovich and B. A. Trakhtenbrot. Behavior structures and nets. *Fundamenta Informaticae*, 11(4):357–404, 1988.

[vG90] R. J. van Glabbeek. *The Linear Time - Branching Time Spectrum.* Research Report CS-R9029, CWI, July 1990.

[vGV87] R. J. van Glabbeek and F. Vaandrager. Petri net models for algebraic theories of concurrency. In *Proc. PARLE 87, vol. II: Parallel Languages, Eindhoven, LNCS 259*, pages 224–242, Springer-Verlag, June 1987.

A Configuration Approach to Parallel Programming

Jeff Magee, Naranker Dulay

Department of Computing,
Imperial College of Science, Technology and Medicine,
180 Queen's Gate, London SW7 2BZ, UK.

Abstract

This paper advocates a configuration approach to parallel programming for distributed memory multicomputers, in particular, arrays of transputers. The configuration approach prescribes the rigorous separation of the logical structure of a program from its component parts. In the context of parallel programs, components are processes which communicate by exchanging messages. The configuration defines the instances of these processes which exist in the program and the paths by which they are interconnected.

The approach is demonstrated by a toolset (Tonic) which embodies the configuration paradigm. A separate configuration language is used to describe both the logical structure of the parallel program and the physical structure of the target multicomputer. Different logical to physical mappings can be obtained by applying different physical configurations to the same logical configuration. The toolset has been developed from the Conic system for distributed programming. The use of the toolset is illustrated through its application to the development of a parallel program to compute Mandelbrot sets.

1 Introduction

The work described in this paper arose from our interest in applying the principles embodied in Conic [KRA85, MAG89] to a programming environment for multicomputers [ATH88]. The shortcomings we perceived in existing programming environments for multicomputers based on transputer arrays provided additional motivation. The modifications necessary to Conic to enable its efficient use in the transputer environment led to naming the variant Tonic, for obvious reasons.

Conic is a toolkit for constructing distributed systems. It provides two languages: the first, a declarative configuration language used to describe the structure of a logical node in terms of its constituent process types, process instances and process interconnections and the second, a programming language used to program individual process types. The programming language is Pascal augmented with message passing primitives. Distributed systems are constructed in Conic by dynamically assigning instances of logical nodes to physical nodes and interconnecting these instances. Conic embodies the configuration approach in rigorously separating the logical structure of a distributed program from the components which implement its computational function. The differences between Tonic and Conic arise from the characteristic differences between parallel and distributed programs. We see these as being:

Objective - Distributed programs can be considered as consisting of a number of logically distinct entities which intercommunicate to achieve some overall goal - typically access to geographically distributed resources. Parallel programs are logically one entity where the constituents co-operate to achieve some computational goal - the overall objective of performing the computation in parallel being speedup.

Failure - Failure of one component of a distributed program generally requires continued operation albeit in degraded mode whereas failure of one component of a parallel computation can generally be allowed to cause termination of the overall computation. In distributed environments, the longevity of execution, together with the probability of communication and node failure means that software development toolkits must provide programming abstractions to deal with such failures. This is not the case in multicomputers where we can assume reliable communication and low probability of node failure during the execution of programs which have as their primary objective speedup rather than continuous execution.

Evolution - A large class of critical distributed programs execute perpetually and for economic or safety reasons require the facility to be modified and updated on-line. The Conic toolkit supports this requirement through its ability to dynamically configure running systems. On-line evolution is not a requirement for parallel programs which run for relatively short periods, completing when a result has been computed.

Heterogeneity - Distributed programs are generally designed such that components of the program may run on computers with different processor types. Programming environments for distributed systems deal with this hardware heterogeneity by providing multiple code generators and message datatype conversion facilities. Distributed memory multicomputers, typified by transputer networks, provide hardware homogeneity, although they are usually hosted by a computer with a different processor type. However, programming environments for multicomputers should optimise for hardware homogeneity.

In summary, Tonic is optimised to support the development of parallel programs for distributed memory multicomputers where the primary objective is speedup. Tonic does not support dynamic configuration and assumes reliable processors and reliable inter-processor communication. It inherits from Conic the configuration approach in providing a separate language to define logical structure and extends the Conic configuration facilities by also applying this language to describing the physical structure of the target multicomputer. This physical configuration description is used to drive the logical to physical mapping process.

Currently, the most commonly used toolset for developing parallel programs for transputer based multicomputers is the Occam language [INM88a] and the Transputer Development System (TDS) [INM88b]. While these are efficient tools for developing embedded applications, they have drawbacks when used for developing application programs for the current generation of transputer based multicomputers such as the Meiko Computing Surface and the Supernode. The drawbacks are primarily concerned with the flexibility permitted in mapping an arbitrary network of communicating Occam processes to the hardware topology of interconnected transputers. The developer must take into account the limit of four links per transputer and laboriously place logical channels onto physical channels. Explicit multiplexor and demultiplexor processes must be provided where it is necessary to map more than one logical communication channel onto a physical link. Changing the number of processors on which an application runs requires recompilation.

More recent toolsets such as CStools [MEI89] address the problem of making the logical structure independent from the underlying hardware structure through the use of configuration descriptions (termed "par files"), however, these descriptions do not permit runtime parameterisation of the number of processors or flexible logical to physical mapping unless the user resorts to the underlying library routines. The Helios operating system [DSL90] allows a user to describe the hardware configuration (Resource Map) separately from the logical configuration (CDL). These descriptions use different notations and are limited to compile time parameterisation. Further, the application programmer has little control over the logical to physical mapping. Helios programmers are limited to Unix style I/O for inter-process communication.

In the following section, the Tonic facilities for developing parallel programs are illustrated through the development of a program to compute the Mandelbrot Set. The facilities provided for mapping this program to different hardware topologies are described in section 3. Section 4 overviews the implementation of Tonic and provides some performance data. Finally, section 5 evaluates the approach.

2 Program Construction in Tonic (Logical Structure)

The following overviews the programming features offered by Tonic for parallel programming through the example of a program to generate Mandelbrot sets. The program generates a 512 by 512 pixel image where the colour of each pixel is represented by an 8 bit quantity. This quantity is computed as the number of iterations (up to a maximum of 255) of the calculation **z= z*z+c** before **|z|>2** where **z** is a complex variable and **c** a complex constant. If the maximum is reached **c** is assumed to be in the Mandelbrot Set, otherwise the number of iterations indicates how "close" **c** is to the set. The simplistic approach to parallelising this is to divide the image into the same number of chunks as there are processors and hand each chunk to a processor for computation. Since some image areas, far outside the set, require much less computation than others this approach leads to poor load balancing and thus poor performance. A more sophisticated approach employs a work allocator or supervisor process to hand out smaller chunks to worker or slave processes [MAG91]. A slave process computes a chunk and hands it back to the supervisor for display and then gets another chunk to compute until none are left. In the following, chunks are the size of one horizontal line of pixels. The logical structure of the program is shown in Figure 1.

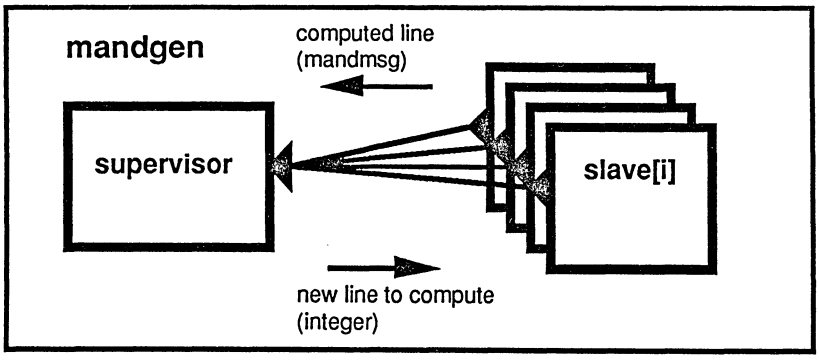


Figure 1 - Logical structure of the Mandelbrot Generator Program

The types of message exchanged between components of the program together with program wide constants are defined in a definitions unit (c.f. Modula-2 modules) as shown below.

```
1  define mandbrot:Xmax,Ymax,mandmsg,mandp;
2  const  Xmax=512;  Ymax=512;
3  type     mandp =  ^mandmsg;
4           mandmsg =  record
5                         lineno:integer;
6                         linebuf:packed array[1..Xmax] of char;
7                      end;
8  end.
```

The program unit shown below is the definition of the slave process type - in Tonic process types are **task modules**. Tasks[1] communicate with the outside world by sending messages to exitports and receiving messages from entryports. A task has no direct knowledge of which other tasks it will be connected to. This configuration independence greatly facilitates reuse. In this case, the slave task sends a computed line of pixel colour values (type *mandmsg*) to its exitport *result* (line 4). The communication primitive used is a send-wait-fail (line 23) which sends a request message to the exitport and then suspends the task waiting for a reply or an abort. In the program below, an abort (indicated by a non-zero value of *rcode*) indicates that the supervisor has no more lines to compute. The first message from a slave to the supervisor has a zero linenumber indicating that the message does not include a computed line - it is merely a request for the first line to compute. Subsequent messages overload a request for a new line with the results of computing the last line.

```
1  task module slave(x0,y0,d0:real);
2  use
3      mandbrot:Xmax,Ymax,mandmsg;
4  exitport
5      result:mandmsg reply integer;
6  var
7      M:mandmsg; x1,y1,delta:real; i,rcode:integer;
8
9  function mandcalc(cx,cy:real):integer;
10 var i:integer; zx,zy,xx,xy,yy,t:real;
11 begin
12     i:=0; zx:=cx; zy:=cy;
13     repeat
14         xy:=zx*zy; xx:=zx*zx; yy:=zy*zy;  zy:=xy+xy+cy;  zx:=xx-yy+cx;
15         t:=xx+yy; i:=i+1;
16     until (t>4.0) or (i=256);
17     mandcalc:=i;
18 end;
19
20 begin
21     M.lineno:=0; delta:=d0/Xmax;
22     loop
23         send M to result wait M.lineno fail rcode;
24         if rcode<>0 then exit;
25         x1:=x0; y1:=y0-(M.lineno-1)*delta;
26         for i:=1 to Xmax do begin
27             M.linebuf[i]:=chr(mandcalc(x1,y1)); x1:=x1+delta;
28         end;
29     end;
30     delay(maxint); {supends task indefinitely}
31 end.
```

The *supervisor* component shown in Figure 1 is implemented by two tasks as shown in Figure 2.

[1] The terms **task** and **process** are used interchangeably throughout the paper.

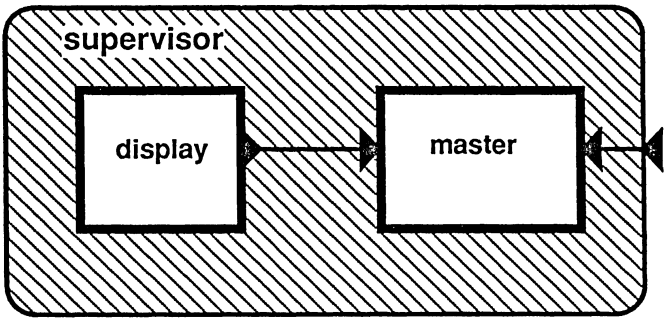


Figure 2 - Supervisor Composite Component

The *supervisor* composite component is described in the Tonic configuration language by the group module below. Note that the interface to a group module is defined in an identical way to task module interfaces. Thus tasks can be replaced by groups and vice-versa at any point during program development without affecting the rest of the program. The configuration description consists of four parts: i) the use clause (line 2) specifies the message and component types used to construct the group, ii) the interface to the component in terms of entry and exitports (line 6), iii) the instance of component types from which the component is constructed (line 8) and iv) the interconnections between instances of these components (line 11). Since the *master* task is critical to the overall performance of the program, its associated pragma (line 10) indicates that this task instance should be run as a high priority transputer process with its workspace in on-chip memory if possible.

```
1  groupmodule supervisor;
2  use
3       mandbrot:mandmsg;
4       display;
5       master;
6  entryport
7       result:mandmsg reply integer;
8  create
9       display;
10      master <PRI=0, MEM=ONCHIP>;
11 link
12      display.out to master.out;
13      result to master.result;
14 end.
```

The program for task module *master* is given below. This task allocates lines to be computed to *slave* tasks through replies to the entryport *result* (line 18) and stores computed lines in the array *bufs*. When there are no more lines to be allocated, the task sending to *result*

is aborted (line 19). As noted previously, this stops the *slave* task requesting lines. Since lines will be computed in different times by slave processes, computed lines will not be received by *master* in line order.

```
1  task module master;
2  use
3       mandbrot:Xmax,Ymax,mandmsg,mandp;
4  entryport
5       result:mandmsg reply integer;
6       out:signaltype reply mandp;
7  var
8       written,allocated:integer; current:mandp;
9       bufs:array[0..Ymax] of mandp;
10 begin
11      for written:=0 to Ymax do bufs[written]:=nil;
12      written:=0; allocated:=0; new(current);
13      loop
14           select
15                receive current^ from result
16                => allocated:=allocated+1;
17                     if allocated<=Ymax then
18                          reply allocated to result
19                     else abort(result);
20                     with current^ do if lineno<>0 then begin
21                          bufs[lineno-1]:=current; new(current);
22                     end;
23           or
24                when bufs[written]<>nil
25                receive signal from out reply bufs[written]=>written:=written+1;
26           end;
27      end;
28 end.
```

The guard on the receive from the entryport *out* (line 25) ensures that the display task receives lines in the correct order. The semantics of the select statement are identical to the Conic select statement. It should be noted that the *master* task does not have or need information on the number of *slave* tasks that are connected to it. In the following, this will allow us to simply parametrise the overall program with the number of *slave* tasks.

The remaining task *display*, listed below, exists to decouple I/O latencies to the display from the response time of *master* to requests for lines. The send-wait (line 10) has no fail clause indicating that if this send-wait was aborted the task would terminate in an error state. Note that this is the only task in the program that terminates. The Tonic termination model simply states that when any one task terminates (correctly or erroneously) the entire program is terminated. In this Tonic differs considerably from its predecessor which allowed continued operation in the presence of failures. This decision is consistent with the different characteristics of parallel and distributed programs identified in the introduction.

```
1  task module display;
2  use
3      mandbrot:Xmax,Ymax,mandmsg,mandp;
4  exitport
5      out:signaltype reply mandp;
6  var
7      current:mandp; i,j:integer; output:text;
8  begin
9      for i:=1 to Ymax do begin
10         send signal to out wait current;
11         for j:=1 to Xmax do
12             write(output,current^.linebuf[j]);
13     end;
14 end.
```

The final step in developing the Mandelbrot program is to describe the overall configuration structure of slave and supervisor components together with an abstract description of how we wish these to be executed on the multicomputer. At this stage, we merely indicate a mapping of components to an abstract machine which consists of **maxprocessor** identical processors. We are not concerned with the physical details of how these processors are interconnected. In fact, we assume that they are fully interconnected or, as termed in the following - **globally interconnected**. The configuration description for the program *mandgen* is given below.

```
1  group module mandgen(x:real=-2.0;y:real=2.0;d:real=4.0); {default parameter values}
2  use
3      execpar;
4      supervisor;
5      slave;
6  create
7      execpar;
8  create
9      supervisor;
10 create forall k:[1..maxprocessor] at (k)
11     slave[k](x,y,d)  <MEM=ONCHIP> ;
12 link forall k:[1..maxprocessor]
13     slave[k].result to supervisor.result;
14 end.
```

The replicator **forall** is used to declare vectors of components (line 10) or links (line 12). The **at** clause (line 10) specifies the processor at which the component instance is to be located. Any integer expression may follow the keyword **at** to denote the processor. Components with no **at** clause are by default allocated to the processor to which their parent group is allocated (in this case 1). More precisely, the rules governing allocation to processor numbers are:

(1) Processors are numbered from 1 to **maxprocessor.**
(2) Any component instance can be allocated to a processor by **at.**
(3) The default allocation is to the parent group (ie **at** not used) .
(4) The top-level group is conceptually allocated to processor 1.

These rules mean that an **at** clause can appear at any level of a configuration description. For example, the configuration description for the parallel executive component *execpar* (below), allocates an instance of the component *exec800* to each processor. *Execpar* provides I/O to the host, error reporting and inter-processor communication It should be noted that **maxprocessor** is not a compile-time constant - it is initialised at run-time as described in the next section.

```
1  groupmodule execpar(buffers:integer=4);
2  use
3      exec800;
4  create forall k:[1..maxprocessor] at(k)
5      exec800[k](buffers);
6  end.
```

This section has demonstrated how parallel programs are constructed in Tonic. The *mandgen* program includes examples of request-reply communication and one-to-one and many-to-one communication (many *slave* exitports to one *supervisor* entryport). Tonic also includes unidirectional synchronous communication primitives, a mechanism for responding to requests in a different order to which they were received and a forwarding facility. Description of these is beyond the scope of the present paper. The next section describes how the Mandelbrot program can be executed on many different hardware configurations.

3 Logical to Physical Mapping

The previous section has described the logical structure of the Mandelbrot program *mandgen.* This logical structure is annotated with a mapping of components to an abstract machine consisting of **maxprocessor** identical globally interconnected processors. In this section, we outline how this abstract machine is realised on the actual hardware. In our case the target hardware is a Meiko Computing Surface consisting of a SPARC based host (running Unix) and 32 T800 transputers each with 4 Megabytes of memory (figure 3). Via a utility provided by Meiko (*svcsd*), a user can reserve a variable number of transputers and set up inter-transputer links before downloading an application program. *Svcsd* provides a bidirectional message passing interface from the host to link 0 of one of the reserved transputers

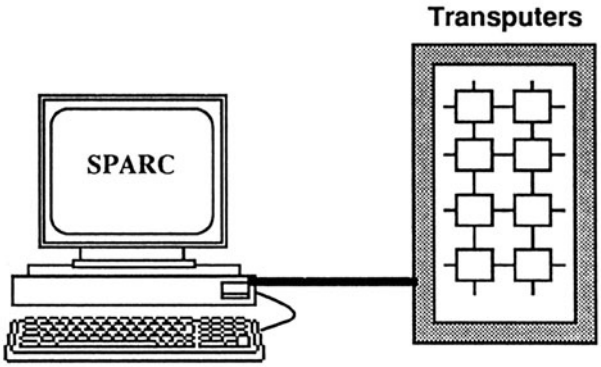
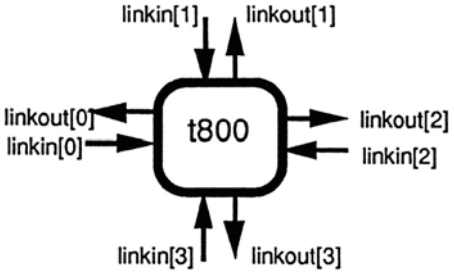


Figure 3 - Meiko Computing Surface

The Tonic Configuration language is used to describe the desired physical topology of interconnected transputers. This physical configuration description is used to drive the logical to physical mapping process. Figure 4. depicts the configuration view of an individual T800 transputer.




```
1  task module t800;
2  exitport
3      linkout[0..3]:byte;
4  entryport
5      linkin[0..3]:byte;
6  begin
7      {never executed}
8  end.
```

Figure 4 - Configuration view of T800 transputer.

The T800 transputer type is represented by a Tonic task module. The task has no code since it is never executed. It serves only to provide an interface specification to configuration descriptions. Using this definition, we can now describe physical topologies of transputers. The following group module describes a pipeline where link 1 of each transputer is connected to link 0 of its successor in the pipeline. The pragma (line 7) associates an integer processor identifier with each *t800* instance. These processor identifiers are used during the mapping process. A component in the logical configuration will execute at the processor whose identity corresponds to that specified by the component's **at** clause. In the interest of conciseness, it is only necessary to define either *linkout[m] to linkin[n]* or *linkout[n] to linkin[m]* to specify a hardware connection between two transputers.

```
1  groupmodule pipeline(length:integer);
2  entryport
3      linkin:byte;
4  use
5      t800;
6  create forall k:[1..length]
7      t800[k] <PID=k>;
8  link forall k:[1..length-1]
9      t800[k].linkout[1] to t800[k+1].linkin[0];
10 link
11     linkin to t800[1].linkin[0];
12 end.
```

The following description of a ternary tree of transputers illustrates some of the more powerful features of the Tonic configuration language - namely guards and recursion. In this example we have omitted pragmas to explicitly associate identities to processors but relied on the default assignment of identities supplied by the underlying system. This definition of a ternary tree is of limited usefulness since it only generates configurations of 1 processor (depth=0), 4 processors (depth=1), 13 processors (depth=2), etc. In practice, we use a definition of *ternarytree* which generates balanced ternary trees for any number of processors.

```
1  groupmodule ternarytree(depth:integer);
2  entryport
3      linkin:byte;
4  use
5      t800;
6  create
7      root:t800;
8  link
9      linkin to root.linkin[0];
10 when depth>0
11 create forall k:[1..3]
12     child[k]:ternarytree(depth-1);
13 when depth>0
14 link forall k:[1..3]
15     root.linkout[k] to child[k].linkin;
16 end.
```

To complete the hardware configuration description the connection between the host processor and target transputer system must be specified as shown below for both the pipeline and ternary tree topologies. Line 9 specifies the connection between the host, represented by the component *gin,* and the first transputer in the pipeline (or ternarytree). *Gin* is the engine which performs most of the work of providing the Globally INterconnected abstract machine required to execute the logical configuration. Its implementation is described in outline in the next section. The name was irresistible.

```
1  groupmodule pipe(length:integer);
2  use
3      gin;
4      pipeline;
5  create
6      gin;
7      pipeline(length);
8  link
9      gin.linkout to pipeline.linkin;
10 end.
```

```
1  groupmodule ttree(depth:integer);
2  use
3      gin;
4      ternarytree;
5  create
6      gin;
7      ternarytree(depth);
8  link
9      gin.linkout to ternarytree.linkin;
10 end.
```

The above group modules *pipe* and *ttree* compile into the host executable files **pipe** and **ttree.** The program *mandgen* described in the previous section compiles into the target executable file **mandgen.800.** To execute the Mandelbrot program on a pipeline of four processors the user types the following command on the host. The logical to physical mapping is depicted in figure 5.

pipe 4 mandgen.800 2.0 2.0 4.0 | pixdisp[2]

Similarly, to execute the Mandelbrot program on a ternary tree of depth 2 (13 processors), the user types the command:

ttree 2 mandgen.800 2.0 2.0 4.0 | pixdisp

Note that **ttree** and **pipe** can be applied to any application program, they are not specific to the Mandelbrot example.

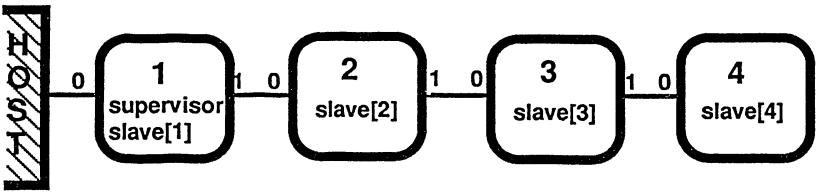


Figure 5 - Mandgen mapped to a pipeline of 4 transputers

[2] Pixdisp is a program executing on the Unix host which reads from its standard input and displays the bytes read as coloured pixels on an Xwindow.

4 Implementation & Performance

In this section, we give an overview of how Tonic programs are executed on the Meiko Computing Surface. The latter part of the section discusses performance.

Tonic Configuration Language

Each group module in a configuration description compiles into a procedure to elaborate the structure of that group at run-time. The set of these elaboration procedures when executed at run-time generate a directed graph in which the nodes are task instances and the arcs are intertask links. Group modules are not represented in this graph, it is a flat representation of the hierarchical configuration structure [DUL90]. Consequently, no penalty is paid at run-time for using hierarchically structured configuration descriptions.

Bootstrapping the transputer network.

When a physical configuration description is executed on the host (e.g. **ttree**) the graph structure generated is passed to **Gin.** This graph represents the desired configuration of transputers required to execute the logical configuration. **Gin** performs the following sequence of actions:

1) The graph is checked to ensure that it represents a legal transputer configuration. That is, all transputer connections are one-to-one, processor identifiers are in a contiguous range, and the graph is fully connected (so that there is a path to boot every processor).

2) **Gin** then computes a minimum depth spanning tree for the graph. This identifies which transputer links will be used for bootstrapping. The complete graph is recorded in an adjacency matrix *AJ[0..maxprocessor, 0..maxlink]* where *maxlink=3* and *AJ[i,j]* is the identity of the processor to which link *j* of processor *i* is connected.Processor 0 represents the host. The matrix is marked with those links which will be booted.

3) Using the *svcsd* utility, **gin** grabs the required number of processors and interconnects them to conform to the graph generated by the physical configuration description.

4) In the next stage, **gin** bootstraps the first transputer by sending the application program (e.g. mandgen.800) to its link 0. Once the program starts executing, **gin** sends it four further pieces of information:

a) its processor identifier (in the range *1..maxprocessor*).

b) the maximum number of processors **maxprocessor**.
c) the adjacency matrix **AJ**.
d) the command arguments represented as strings (Unix argc & argv).
For the example, these would be *mandgen.800 2.0 2.0 4.0*.

5) At this stage, **gin** is finished with the bootstrapping process and becomes a server which services I/O requests from the application program. It runs until either the program running on the network reports an error or terminates.

The application program loaded into each transputer continues the bootstrap process. When started it receives the items a) to d) listed in 4) above. The program then examines the entry in the adjacency matrix corresponding to its processor identifier. If an entry *AJ[self,j]* is marked to be bootstrapped, the program sends its code to outgoing link *j* to bootstrap the processor to which link *j* is connected. It then sends the processor identifier *AJ[self,j]*, **maxprocessor**, *AJ* and the command arguments to complete the bootstrap. Note that exactly the same code is loaded into each processor. The only value which a processor receives to distinguish it from others is its processor identifier. After the first level of the boot spanning tree has been bootstrapped, booting continues in parallel until the leaves of the tree have been bootstrapped.

Initialisation

After completing bootstrapping, each transputer executes the same initialisation code. This code first calculates a minimum distance routing table from the adjacency matrix. The routing table is used by *execpar* at execution time. Initialisation proceeds by invoking the group elaboration procedures. Each transputer node thus has a copy of the complete logical configuration graph. However, the kernel (which is part of *execpar*) only instantiates tasks which correspond to its processor identifier ie. an **at** clause in the logical configuration specified "this" processor. The kernel in addition to instantiating tasks creates datastructures to implement both local and remote inter-task communication. These communication datastructures contain initialised transputer channel words. The Tonic communication primitives are implemented using the transputer communication instructions , ***in***, ***out***, ***alt*** etc.

Loading the entire code for the application at each processor has the disadvantage of wasting storage when task types are loaded but not instantiated. However, this scheme has the following major advantages:

1) It permits the bootstrap to proceed in parallel. This considerably reduces startup time for large numbers of processors.

2) Since each node has a complete copy of the logical configuration graph, the setup of inter-task communication associations at initialisation time does not require remote communication. The initialisation of each transputer proceeds in parallel. Again this reduces application startup time. Tonic applications typically take less than a second to startup.

Performance

Figure 6 shows the times required for a request-reply message exchange. The time is measured from the time the sending task initiates the exchange by a **send-wait** to the time the reply message completes the exchange. The receiving task executes a **receive** followed by a **reply.**

	4 byte Request 4 byte Reply	100 byte Request 4 byte Reply
Intra-Processor	17uS	19uS
Inter- Processor	174uS	241uS
+ time per additional intermediate processor	144uS	210uS

Figure 6 - Request - Reply times.

The times above are for a one to one request-reply communication i.e. one exitport connected to one entryport. Where the communication is **n** to **1** (**n** exitports connected to **1** entryport as in the Mandelbrot example) the time for an individual intra-processor request-reply is **T + 5*n** uS (for **n>1**) where **T** is the time for a one to one communication. This is because many to one communication is implemented using the transputer **alt** instruction. The receiver task waits on a set of transputer channels representing the set of exitports connected to it. Consequently, the time to receive from an entryport is proportional to the number of exitports connected to it . This represents a considerable performance penalty for large fan-in configurations. For example, with 32 processors, the *mandgen* program has a 32 to 1 connection to the *supervisor's result* entryport. Consequently, the performance penalty is 160uS per communication. We are currently re-implementing intra-processor communication using critical regions rather than transputer communication channels to make the receive time independent of the fan-in factor.

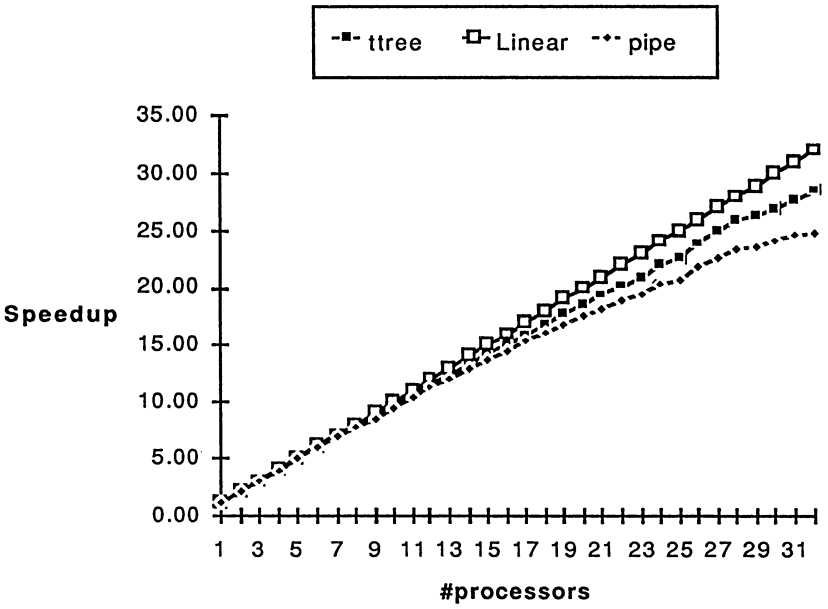


Figure 7 - Mandgen program speedup

Figure 7 represents the speedup for the Mandelbrot program example plotted against the number of processors using a balanced ternary tree hardware configuration (*ttree*) and a pipeline hardware configuration (*pipe*). Speedup(n) is measured as the time for 1 processor (i.e. 1 *slave* & 1 *supervisor*) divided by the time for n processors (i.e. n *slaves* & 1 *supervisor*). Not surprisingly, the ternary tree mapping outperforms the pipeline mapping. The average processing rate for the 32 processor ternary tree mapping is 26Mflop/S or 0.8Mflop/S per transputer. The overall time with 32 processors to complete the computation with the parameters (2.0,2.0,4.0) was 3.3 seconds.

5 Discussion & Conclusions

The paper has presented a configuration approach to the construction of parallel programs in which the functional behaviour of individual components is specified by a programming language and the overall parallel program structure is specified by a configuration language. The configuration specification declares the instances of component types and their interconnections. Component instances execute in parallel. This logical configuration is annotated with a mapping to an abstract machine which consists of **maxprocessor** identical, globally interconnected processors. The physical configuration of the real machine is specified

using the same configuration language. This physical description drives the logical to physical mapping process. Both the logical and physical configuration specifications are considerably more flexible than those provided by existing systems [INM88b, DSL90, MEI89]. Running a Tonic program on different physical configurations with different numbers of processors and different inter-processor connections requires no re-compilation. This facilitates both portability and experimentation with different logical to physical mappings. The configuration approach is similar to that of the MUPPET [MUH88] system which uses a graphical notation to express configurations. However, MUPPET does not clearly separate logical from physical configurations and is limited in the mappings which can be expressed. Tonic also has a graphical notation for expressing configurations and an associated display tool [KRA89]. However, for describing regular structures, we find the power of the textual language to be more useful.

In concurrent programming terms, Tonic falls into the class of languages which support the process/message passing paradigm [BAL89]. The best known of these languages are Occam [INM88b] and Ada[DOD83]. Unlike these languages, Tonic incorporates a separate language to describe the structure of concurrent programs in terms of task instances and inter-task message paths (links). These languages also differ in the time at which the program's process structure is fixed. Occam defines the process structure statically at compile time, Tonic at instantiation/initialisation time and Ada dynamically at run-time. In fairness, it should be noted that Ada semantics do not permit an efficient distributed memory implementation. We are currently experimenting with an approach which would allow changes to the process structure at run-time while retaining a strict separation between programming and configuration [MAG90]. This will permit a limited, but efficient, form of process migration to facilitate dynamic load balancing.

We regard Tonic as a prototype implementation to validate the configuration approach to parallel programming. Its application is restricted by the reliance on one specific programming language - Pascal + message passing. Currently, we are engaged in the development of a configuration language (Darwin) [MAG90] and associated tools which will permit the configuration approach to be applied to parallel programs composed of components written in commonly available languages such as C & Fortran.

Despite the limitations expressed above, Tonic is a practical and efficient tool for developing parallel programs. It hides many of the irrelevant details about the underlying hardware which currently harass parallel programmers. Typically, application programmers select a physical configuration from a library rather than programming their own. The library currently includes pipeline, ring, mesh, torus, binary tree, ternary tree, cube connected cycles and WK-Recursive physical topologies. The toolset is used by both research and undergraduate students.

Acknowledgements

The authors would like to acknowledge discussions with our colleagues in the Parallel and Distributed Systems Group during the formulation of these ideas. We gratefully acknowledge the SERC under grants GE/E/62394 (ACME) & GR/G31079, and the CEC in the REX Project (2080) for their financial support.

References

[ATH88] W.C. Athas, C.L. Seitz, "Multicomputers:Message-Passing Concurrent Computers", IEEE Computer, Vol. 21, No.8, August 1988, pp 9-24.

[BAL89] H.Bal, J. Steiner, A Tanenbaum, "Programming Languages for Distributed Computing Systems, ACM Computing Surveys,Vol. 21, No. 3, September 1989, pp 261-322.

[DOD83} Department of Defense, U.S.A., "Reference Manual for the Ada Programming language", ANSI/MIL-STD-1815A, DoD, Washington D.C. Jan 1983.

[DUL90] N. Dulay, "A Configuration Language for Distributed Programming", Ph.D. Thesis, Dept. of Computing, Imperial College, February 1990.

[DSL90] Distributed Software Ltd, " The Helios Parallel Programming Tutorial", 670 Aztec West, Bristol, January 1990.

[INM88a] Inmos Ltd, "OCCAM 2 reference manual", Prentice Hall, 1988.

[INM88b] Inmos Ltd, "Transputer Development System", Prentice Hall, 1988.

[KRA85] J.Kramer, J.Magee, "Dynamic Configuration for Distributed Systems", IEEE Transactions on Software Engineering, SE-11 (4), April 1985, pp. 424-436.

[KRA89] J. Kramer, J. Magee, K. Ng, "Graphical Configuration Programming", IEEE Computer, Vol 22, No 10, Pages 53-65.

[MAG89] J.Magee, J.Kramer, and M.Sloman, "Constructing Distributed Systems in Conic" IEEE Transactions on Software Engineering, SE-15 (6), June 1989.

[MAG90] J.Magee, J.Kramer, M. Sloman, and N.Dulay, " An Overview of the REX Software Architecture", Proceedings of the 2nd IEEE Workshop on Future Trends of Distributed Computing Systems" Cairo, Egypt, Sept. 1990, pp 396-402.

[MAG91] J.N. Magee and S.C. Cheung, " Parallel Algorithm Design for Workstation Clusters", Software-Practice and Experience, Vol. 21. March 1991, pp 235-250.

[MEI89] Meiko Ltd, "CS Tools Documentation Guide", 650 Aztec West, Bristol, 1989.

[MUH88] H. Muhlenbein, Th. Scheider, and S. Streitz, "Network Programming with MUPPET", Journal of Parallel and Distributed Computing, Vol 5, 1988, Pages 641-653.

CHAOTIC LINEAR SYSTEM SOLVERS IN A VARIABLE-GRAIN DATA-DRIVEN MULTIPROCESSOR SYSTEM*

Jean-Luc Gaudiot and Chih-Ming Lin

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-0781

Abstract

Linear systems are important problems in many scientific applications. While asynchronous methods are effective solutions to linear systems, they are difficult to realize due to the chaotic behavior of the algorithms. In this paper, we investigate the implementation as well as the performance of an asynchronous method, namely chaotic relaxation, in our Variable-grain Tagged-Token Data-flow (VTD) System. We compare asynchronous methods with synchronous methods executed on both the fine-grain and the coarse-grain execution models. New high-level data-flow language constructs are introduced in order to express asynchronous operations. A new *firing* rule that deviates from the *single assignment* rule of functional languages is proposed to support the implementation of asynchronous computations in the VTD system. In addition to the conventional *speedup* measure, we then define new performance measurements, called *Growth Factor*, *Scalability Factor*, and *Robustness* to characterize the system performance from the machine and application viewpoints. Simulation results indicate that asynchronous methods are more efficient than synchronous methods and that the coarse-grain execution mode is more efficient that the fine-grain execution mode in our VTD system.

1 Introduction

Linear systems play an important role in many applications such as PDE solvers. Generally, linear systems can be solved by direct or iterative methods. Iterative methods can further be classified as synchronous [9] or asynchronous [3]. While synchronous methods are easy to implement, they do not yield acceptable levels of performance for complex problems, mainly because of the synchronization necessary among the various processes. On the other hand, asynchronous approaches have been found by many researchers [3, 5] to efficiently exploit runtime parallelism. In an asynchronous approach, communication between processes is achieved by reading the dynamically updated variables while each

* This material is based upon work supported in part by the U.S. Department of Energy, Department of Energy Research, under Grant No. DE-FG03-87ER25043.

process continues its execution to update shared variables. Therefore, the chaotic behavior of data in an asynchronous algorithm is very complex. However, while an asynchronous method can be effective in parallel machines and can deliver high performance, it is difficult to implement due to the chaotic behavior of the method itself. From the software perspective, language constructs must be defined to specify the asynchronous method, thereby parallelizing the algorithm. From the hardware point of view, special architecture schemes dedicated to the algorithm need to be developed.

The data-flow principles of execution [2] offer the programmability needed to synchronize at runtime the many parallel processes on a large scale multiprocessor. Instead of relying on the conventional central program counter, the availability of data renders an instruction executable. Asynchronous algorithms have been implemented in data-driven systems, more precisely in *micro-actor-based* data-driven systems [5]. Although the micro approach to asynchronous methods correspond well to the simplicity of data-driven principles, it yields much overhead to respect the functionality of execution.

In this paper, we will first introduce special high-level data-flow language constructs (*Async-Repeat* and *Async-For*) to describe the chaotic behavior in asynchronous algorithms. The scheme to form coarse-grain (macro-actor) data-flow graphs and a specific firing rule in the *Matching Store with Locks* of processors will also be introduced in order to correctly execute the computations of the asynchronous algorithms. In this paper, we are also interested in measuring and comparing the performance of algorithms as well as our VTD system: First, to evaluate the performance of the architecture, the conventional *"Speedup"* measurement will be taken to depict the trend of the performance with larger machine configurations. Second, to estimate the growth of parallelism within an algorithm when the algorithm's complexity has been increased, a new measurement, called *"Growth Factor"*, will be defined to show how suitable an algorithm is for multiprocessor systems. Third, to measure the efficiency of parallel systems in the execution of parallel algorithms, we will introduce another new measurement, called *"Scalability Factor"*, to demonstrate the scalability property of the systems. Finally, we will define *"Robustness"* to indicate the potential performance of the systems.

We shall start our discussion in section 2 by giving a brief introduction to the data-flow principles of execution as well as to asynchronous methods for solving linear systems. In section 3, the Jacobi method and the chaotic relaxation method are described in a High-level data-flow language along with the new languages constructs. The VTD System and the new firing rule for chaotic relaxation and the new performance measurements will be described in section 4. Section 5 will present the results of a deterministic simulation on the system and concluding remarks will be made in section 6.

2 Data-flow Principles and Iterative Solutions for Linear Systems

In this section, we first introduce the data-flow principles of execution and review the essentials of the synchronous and asynchronous linear system solvers which will be evaluated on our VTD system.

2.1 Data-flow Principles

Programmability has been identified as the major issue in the design of large-scale multiprocessor systems [1, 2]. Indeed, programmers cannot be expected to be able to schedule and synchronize the hundreds or thousands of tasks that are required to fully utilize the resources of such machines. Therefore, the data-flow model of computation has been introduced to alleviate this problem [1]. Data-flow principles allow runtime synchronization of operations based on their data dependencies. This allows a very large number of different tasks to be scheduled efficiently and transparently.

Data-flow computing is an alternative to the control-flow model. It is inherently parallel, as the execution of an instruction is based upon the availability of its arguments. Data-flow principles can be characterized by two statements: First, operations execute only when all required operands are available. Second, actors are purely functional and execution produces no side-effects. Data-flow programs are represented by directed graphs which consist of actors connected together with arcs. Arcs represent the data dependencies between actors and carry tokens which are the data values passed between actors [2].

2.2 Jacobi Method

The Jacobi iterative method can be derived from a liner system $A \times x = b$ as:

$$x_i^{(k+1)} = \frac{-\sum_{j \neq i, j=1}^{n} a_{ij} x_j^{(k)} + b_i}{a_{ii}} \quad for\ i = 1, \ldots, n\ and\ k \geq 0 \qquad (1)$$

where the $x_i^{(0)}$'s are initial estimates of the components of the solution x. By examining the Jacobi iterative method shown above, it can be seen that all the components of the previous (*old*) vector $x^{(k)}$ must be saved before the components of the next (new) vector $x^{(k+1)}$ are computed. Therefore, in this algorithm, an iterative sequence of approximations $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$ will be sequentially computed.

2.3 Chaotic Relaxation

In the asynchronous approach, each process continues execution to update the elements in $x_{(i)}$ and communication between processes has been achieved by reading the dynamically updated variables. A subset of asynchronous methods, called *chaotic relaxation schemes*, was introduced by Chazan and Miranker [3] to solve linear systems. In a chaotic relaxation scheme, practical constraints on the asynchronous behavior are imposed. While an asynchronous algorithm imposes no restriction on how "old" a value may be (*i.e.*, how many iterations ago it was produced), chaotic relaxation requires that the updated value of a point be received within a fixed amount of time.

3 From Algorithms to Data-flow Graphs

We have now established two categories of algorithms for linear system solvers that we will implement and evaluate on our VTD system. These algorithms will now be expressed in a high-level data-flow language translated into data-flow graphs.

3.1 The Jacobi Method and Synchronous Constructs

The algorithm is shown in Fig. 1 in SISAL [8]. Line 4 contains the decision to proceed or not with the relaxation at each iteration. The *Relaxation* procedure (lines 5 through 15) performs the relaxation for all of the elements in $X[i]$ and generate new values of vector $X[i]$. The *Convergence Check* procedure (lines 16 through 22) checks all the elements and generates a termination signal back to line 4. Here, we use a stopping criterion evaluated by an L_∞ norm.

In the SISAL program, one should note that the relaxation on each element $X[i]$ under the *for* constructs (lines 5 and 6) can be executed in parallel mainly due to the definition of the language constructs. In the same way, the convergence check on each elements of $X[i]$ and *old* $X[i]$ can be executed in parallel under the *for* constructs in line 16. However, the algorithm itself will be executed in a synchronous manner. In other words, a *step-by-step* iteration process will take place.

3.2 Chaotic Relaxation and Asynchronous Constructs

The chaotic relaxation is an approach which is particularly successful in parallel environments. However, new language constructs must be introduced to describe the algorithm.

3.2.1 The Asynchronous Constructs

Asynchronous computations cannot be easily implemented by traditional high-level programming constructs. Therefore, we designed the *async-repeat* and the *async-for* operators to represent an asynchronous behavior. The main idea of the new *async-repeat* and *async-for* constructs is to release the synchronization constraints from the *repeat* and *for* constructs in SISAL since then inherently create synchronization points in the body of the loops.

1. **Async-repeat** : This construct allows the procedures inside it to be concurrently evaluated without any synchronization between one another. For example, in the following program, statement (1) and statement (2) can be executed simultaneously and repeatedly as long as the condition $c \leq 100$ remains true:

```
for initial
while c ≤ 100 async-repeat
        a:= old a + 1 ; - - - - (1)
        c:= a + b ; - - - - - - - (2)
return value of c;
end for
```

In the above program, under the asynchronous construct, statement (2) may be executed and its result generated before the completion of the execution of statement (1). In other words, if statements (1) and (2) were executed independently, c may be already larger than 100 (assume $b = 101$ and $a = 0$). This would force termination of the process before a is updated by statement (1). Note that the execution model described above will not be allowed in the conventional *repeat* construct which only executes the two statements one after the other due to the synchronization point imposed by the language construct.

```
define main, jacobi
type OneDim = array[real];
type TwoDim = array[OneDim] ;
function jacobi ( A : TwoDim ; B : OneDim ; N : integer ;
                  returns OneDim )
(1) for initial
        Err := 0 ;
        X := array [1: 0.0, 0.0, 0.0, 0.0] ;
(4)     while Err < N repeat          % convergence check
(5)         X := for i in 0, N        % relaxation on X
               temp1 := for j in 1, N
                  temp2 :=
                  if i ≠ j
                     then A[i,j] * old X[j]
(10)                 else 0.0
                  end if ;
                  returns value of sum temp2
               end for;
               returns array of ( B[i] -temp1 ) / A[i,i]
(15)        end for;                  % generate new X
            Err := for i in 1, N      % generate error norm
               temp3:= if abs(X[i] - old X[i]) < ϵ
                  then 0
                  else 1
(20)           end if ;
               returns value of sum temp3
            end for ;
            returns value of X
        end for
    end function
```

Figure 1: A SISAL program for Jacobi methods.

2. **Async-for** : While the conventional *for* construct in SISAL allows every index value to be synchronously executed in parallel, the *async-for* construct releases the synchronization between each index value and allows independent execution of index values in parallel. For example, in the following program, the return values of array X does not need to wait until all the new values of each index i become available. Instead, each new value of index i can be updated asynchronously as soon as the value is available and the next computation can be started.

```
for initial
X := async-for i in 0, N
        temp := Y[i] × old X[i] ;
        returns value of temp × temp
end for;
```

While the *async-for* construct allows each index i to be evaluated asynchronously, it should be noted that the operation within the construct corresponds to an infinite loop. It ensures that the computation will proceed until another process (outer loop) terminates the whole execution. The following program is an example which shows that the process under the *async-for* construct will be terminated by the process that is under the *async-repeat* construct.

```
for initial
while condition async-repeat
    X := async-for i in 0, N
        temp := A[i] × old X[i] ;
        returns value of temp × temp
    end for;
    Procedure_two ;
    Procedure_three ;
end for;
```

Overall, under the bodies of the new *async-repeat* and *async-for* constructs, synchronization constraints can be released while the *repeat* and *for* constructs create synchronization points inside the bodies of the constructs. However, in general, the *async-for* constructs will require the *async-repeat* to co-exist in a program. This is because only the *async-repeat* construct can terminate the process under the *async-for* constructs.

3.2.2 The SISAL Programs

Chaotic relaxation can be expressed in SISAL by using the new constructs. First, in order to allow several procedures to be executed asynchronously in parallel, the *repeat* must be replaced by *async-repeat* at the outer loop of these procedures. In Fig. 1, in line 4, the *repeat* should be replaced by *async-repeat.* Therefore, under the *async-repeat* construct, both the *Relaxation* procedure (from line 5 to 15) and the *Termination Check* (from line 16 to 23) can be executed concurrently without any dependency between each other. Second, in order to allow each index value, which is under the *for* construct, to be executed asynchronously in parallel, the *for* must be replaced by *async-repeat* at the beginning of the procedure. In line 5, we replace *for* by *async-for.* Therefore, inside

the *async-for* construct, each index value can concurrently proceed the execution of the computation without waiting for other values which are executing the same function.

4 VTD System and Performance Measurements

While the macro-actor concept is a solution which reduces overhead in fine-gain computations, the architecture must be able to execute actors of varying sizes. Our Variable-grain Tagged-token Data-flow (VTD) system has therefore been designed for this purpose. A new firing rule in the VTD system is also proposed to guarantee the proper behavior of chaotic relaxation and to achieve efficient computations. To characterize the performance in the VTD system, new performance measurements are defined along with the conventional performance measurements.

4.1 The VTD System

The VTD system consists of a set of identical Processing Elements (PEs) connected by a hypercube (message-passing) communication network. A single PE consists of 4 units : Matching Store Unit, Instruction Fetch Unit, ALU, Token Formatting Unit [5].

4.2 The Matching Store with Locks

In chaotic relaxation, due to the asynchronous iterations at each grid point, the value of each grid point must be saved for the relaxation of other grid points. In order to guarantee the proper behavior of the chaotic actors, we introduce the notion of **locks** at the inputs of the actors. In other words, we create *locks* inside the matching store for the firing of an actor. Note that the implementation of *locks* in actors corresponds to the *Async-repeat* and *Async-for* constructs of the high-level language. The *locks* will be attached to the input actors of a subgraph. These actors represent the processes that can be executed asynchronously under the *Async-repeat* and *Async-for* constructs.

Under the new firing rule, when an actor is fired, the input tokens remain in the input lock until the next input token is received. In this fashion, the incoming token will replace the stored value and will activate once more the actor. Fig. 2 shows the step-by-step the operation of the new firing rule of an actor along with the *matching store with locks*:
1. Initially, when either token A or token B (A and B have the same tags) comes into the actor F, it will be *locked* inside the actor.
2. When the partner token arrives, actor F will be fired and will produce an output token.
3. After firing actor F, both input tokens remain *locked* inside the actor.
4. When another token C is later received by the actor, the actor is fired with the *locked* token on the other port and the new value on the first port. The incoming token will remain locked in the actor. Note that it overwrites the previous token value.

4.3 Performance Measurements

Many measurements of system performance, such as speedup and system utilization, have been used to evaluate multiprocessor systems in the past. However, these measurements do not clearly indicate the effectiveness of architectures as well as application programs

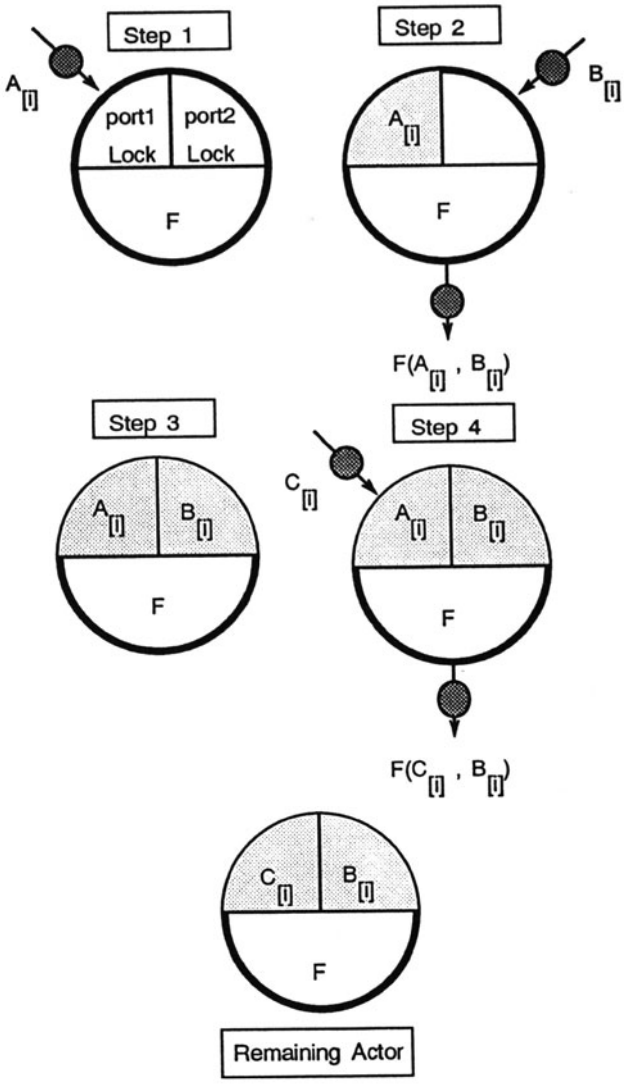


Figure 2: A New Firing Rule with locks in Matching Store.

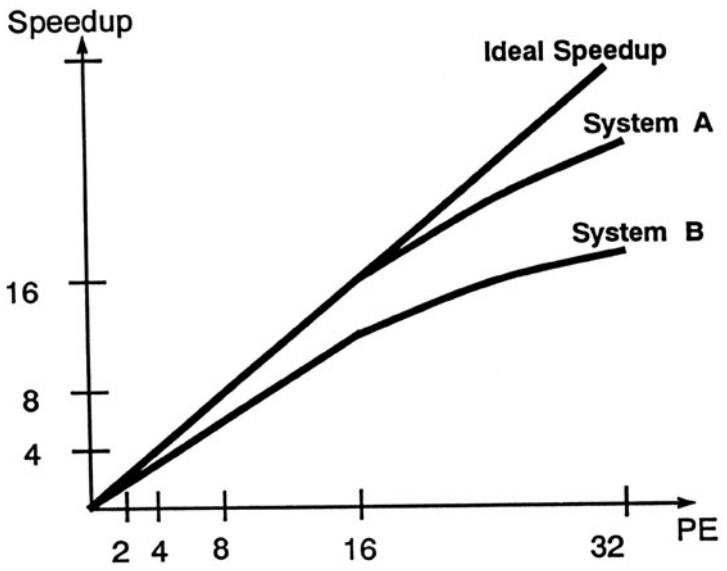


Figure 3: Speedups with Various PEs.

because there is no indication of how much of the performance is due to the architectures and how much of it is due to the applications. Indeed, the speedup should be measured by scaling the problem to the number of processors, not by fixing problem size. An explanation of misuses of Amdahl's speedup formula has been demonstrated in [7]. Therefore, when multiprocessor systems are evaluated, both parallel algorithms and parallel architectures are required to achieve high performance. For instance, a parallel machine cannot deliver high efficiency in executing a sequential program due to the lack of parallelism within the program. On the other hand, a parallel algorithm cannot guarantee high performance in a multiprocessor system if the system cannot exploit the parallelism involved in the program. Clearly, what we need is a better performance measurement to reflect the degree of exploited parallelism resulting from algorithms as well as the ability of the architectures to utilize such parallelism.

In this paper, we are interested in measuring and comparing the performance of algorithms as well as our VTD system: First, to evaluate the performance of the architecture, the conventional *Speedup* measurement is taken to depict the trend of the performance with larger machine configurations. Second, to estimate the amount of the growing parallelism within an algorithm when the algorithm's complexity has been increased, a new measurement, called *Growth Factor*, is defined to show how suitable of an algorithm is for multiprocessor systems. Third, to measure how efficient of parallel systems in executing parallel algorithms, we introduce a new measurement, called *Scalability Factor*, to demonstrate the scalability property of the systems. Finally, we define the *Robustness* to indicate the potential performance of the systems.

1. **Speedup :** Speedup has been conventionally defined as the ratio of the execution time of an Application (AP) on N Processing Elements (PEs) to the execution time of the same application on a single PE :

$$Speedup\ (AP,\ PE(N)) = \frac{Exe.\ time\ of\ AP\ on\ one\ PE}{Exe.\ time\ of\ AP\ on\ N\ PEs}$$

Under this definition, the *ideal* speedup of an architecture is N when there are N PEs in the system. In other words, if a speedup cure is closer to the line of ideal speedup, then the architecture is considered a better parallel system. For instance, Fig. 3 shows that system "A" performs better than system "B" in term of system "A" having a speedup curve closer to the line of ideal speedup. However, by this definition, it is only shown how the execution time can be reduced in various system configurations while the amount of complexity in the application remains unchanged. However, this does not show the suitability of an algorithm for multiprocessor systems. In other words, the speedup curves only demonstrate the machine domain performance without considering the application aspect.

2. **Growth Factor :** Before the speedup in a system is measured, how well an application can perform in parallel systems must be studied. The growth factor shows how much parallelism changes when the complexity of an algorithm is changed. Here, the complexity of an algorithm refers to the number of operations needed to execute the algorithm. For example, the inner product of vectors $V(a_1, a_2, a_3, ..., a_m)$ and $U(b_1, b_2, b_3, ..., b_m)$ has a complexity of $O(m)$. The growth factor therefore is defined as the ratio of the execution time of an Application (AP) with a complexity $(M \times m)$ on a fixed number of n PEs to the execution time of the same application with a complexity of m on the n PEs.

$$Growth\ Factor\ (AP(Mm),\ PE(n)) = \frac{Exe.\ time\ of\ Mm\ AP\ on\ n\ PEs}{Exe.\ time\ of\ m\ AP\ on\ n\ PEs}$$

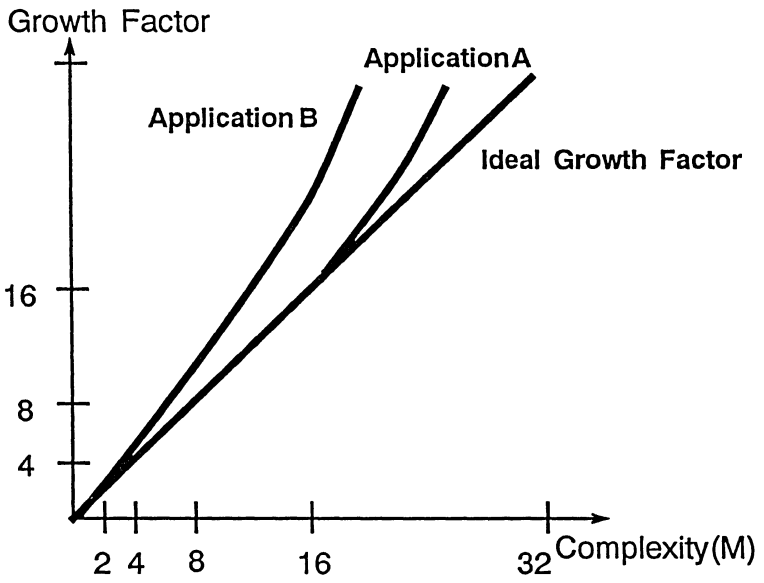


Figure 4: Growth Factors with Various Complexity (M).

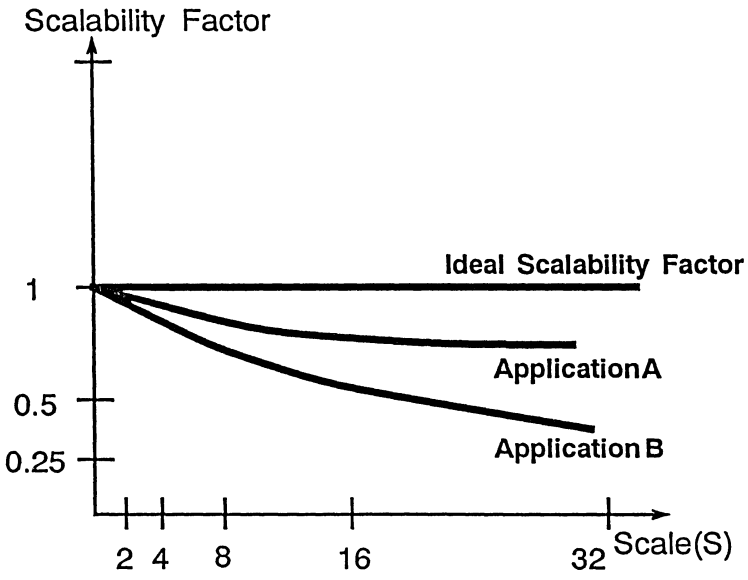


Figure 5: Scalability Factors with Various Scales (S).

Ideally, a perfectly parallel algorithm should have a growth factor proportional to the increasing rate of its complexity (M). For example, a vector to vector multiplication is a perfectly parallel statement that the amount of parallelism increases at the same rate as the vector length (complexity). Therefore, if an application has a curve of growth factor close to the line of ideal growth factor, it is considered a better parallel application. For example, in Fig. 4, application "A" is a better parallel application than application "B" because the curve of growth factor in "A" is closer to the line of ideal growth factor.

3. **Scalability Factor :** The performance of multiprocessor systems should also be measured by comparing the execution time of large problems with that of small problems on single processor systems. In other words, the complexity in the applications should be increased while the size of the machine configuration is increased. The scalability factor is defined as the ratio of the execution time of an Application (AP) with complexity (m) on n PEs to the execution time of the same application with complexity $S \times m$ on $S \times n$ PEs.

$$Scalability\ Factor\ (AP(Sm),\ PE(Sn)) = \frac{Exe.\ time\ of\ m\ AP\ on\ n\ PEs}{Exe.\ time\ of\ Sm\ AP\ on\ Sn\ PEs}$$

If an algorithm has an ideal growth factor and a system has an ideal speedup, then the scalability factor should remain a constant for various values of N. In other words, a perfectly parallel algorithm with a large complexity on a large perfectly parallel system configuration should require the same execution time as it would with a small complexity

Problem Size = 16 × 16				
System Size	Chaotic(Macro)		Chaotic(Micro)	
number of PEs	exe. time	speedup	exe. time	speedup
1 PE	108291	1	108990	1
2 PEs	56690	1.91	54430	2.002
4 PEs	26999	4.01	27174	4.01
8 PEs	13548	7.99	14840	7.34
16 PEs	8050	13.45	10538	10.34
32 PEs	6867	15.76	9708	11.22

TABLE 1 : Execution Time and Speedup in Chaotic Relaxation.

on a small system configuration. However, due to the fact that most algorithms and systems are not perfectly parallelized, the actual scalability factors will fall below the line of ideal scalability factor. Fig. 5 shows that the closer the curve is to the ideal line, the easier it will be to scale up the application/system configuration combination.

4. **Robustness:** The robustness property of a system can actually indicate its potential performance [6]. The robustness is defined as the ratio of the execution time of an Application (AP) with a complexity $(R \times m)$ on one PE to the execution time of the same application with a complexity of $R \times m$ on the $R \times n$ PEs.

$$Robustness\ (AP(Rm),\ PE(Rn)) = \frac{Exe.\ time\ of\ Rm\ AP\ on\ one\ PE}{Exe.\ time\ of\ Rm\ AP\ on\ Rn\ PEs}$$

Essentially, robustness is an indication of how well the architecture/execution model will scale up when machine sizes and problem sizes are increased. In fact, one of the most important parameters in evaluating a multiprocessor system is to observe the system performance with various problem sizes. We thus express the performance of an architecture by showing the robustness in a large number of PEs.

5 Simulation Results

Once the Jacobi method and chaotic relaxation have been programmed and compiled into data-flow graphs. The execution of the graphs in the VTD system can be verified by a deterministic simulation in both micro-actor (fine-grain) and macro-actor (coarse-grain) execution models.

5.1 Simulation Results

The execution of the Jacobi method and chaotic relaxation to solve various sizes of linear systems with the termination criterion $||x^{(k)} - x^{(k-1)}||_\infty < 10^{-3}$ have been simulated. From the simulation results, several statistics and observations have been obtained:

1. **Speedup** : The speedup measure has been defined in the previous section. The reports of the speedups in various system sizes for both chaotic relaxation and the Jacobi method are attached in Tables 1 and 2, while Fig. 6 shows the trend of the speedups with

Problem Size = 16 × 16				
System Size	Jacobi(Macro)		Jacobi(Micro)	
number of PEs	exe. time	speedup	exe. time	speedup
1 PE	79924	1	92203	1
2 PEs	42112	1.89	49399	1.86
4 PEs	23109	3.45	27901	3.30
8 PEs	13640	5.86	18219	5.06
16 PEs	9759	8.18	14470	6.37
32 PEs	9244	8.64	13971	6.59

TABLE 2 : Execution Time and Speedup in the Jacobi Method.

increasing PEs for the two different relaxation methods.

Observation: The results indicate that the speedup in chaotic relaxation is better than the speedup of the Jacobi method in both macro and micro execution modes. In chaotic relaxation, a superlinear speedup can be sometimes observed due to the nondeterministic property of the algorithm itself. Indeed, the random sequence of relaxations may lead to a faster convergence in multiprocessor systems. This feature is confirmed in Table 1: the speedups in a 4 PE system for both macro and micro execution of chaotic relaxation can be as high as 4.01

2. **Scalability Factor**: The scalability factor of a system was defined in the previous section. We exploit the trend of scalability factors in different problem sizes with various system configurations. We start with the matrix size equal to 8×8 and the machine size equal 8 PE, then 16×16 in 16 PEs, 32×32 in 32 PEs, and 64×64 in 64 PEs. The report is shown in Table 3 and the curves are shown in Fig. 7.

Observation: The results show that the chaotic relaxation in the macro execution mode of the VTD system has the best scalability factor while the Jacobi methods in the micro execution mode has the worst scalability factor. However, one should note that the increasing rate of the machine size from 8 PEs to 16 PEs does not equal the increasing rate of the complexity of the algorithms with a matrix size from 8×8 to 16×16. Therefore, we only compare the relative performance of different algorithms in various execution modes, instead of comparing it with the ideal scalability factor.

3. **Robustness**: The robustness of a system was defined in the previous section. We exploit the trend of "speedups" in many different problem sizes with various system configurations. We start with the matrix problem size from 8×8 up to 64×64 and the machine size from 1 PE to 64 PEs. The report is shown in Table 4 and the curves are shown in Fig. 8.

Observation: In the results, we know that there are almost linear increasing speedup curves for the two methods in each operation mode. This is a very promising feature for data-driven multiprocessor systems. Indeed, the robustness property of data-flow architectures can guarantee the performance in multiprocessor systems for various problem sizes. For example, from Table 4, the speedup of chaotic relaxation for 64×64 problem size can reach up to 52 in a 64 PEs system with the macro execution mode.

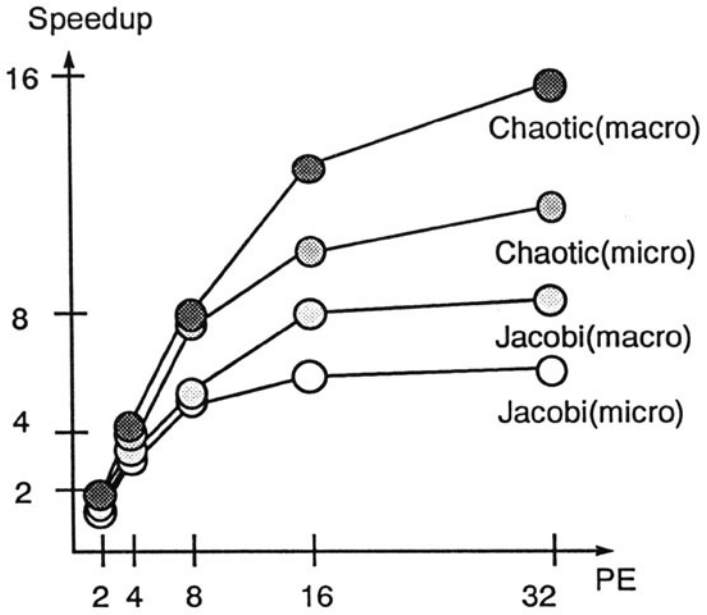


Figure 6: Speedup with Problem Size : 16 × 16.

Scalability Factors					
Number of PEs	Problem Size	Chaotic (Macro)	Chaotic (Micro)	Jacobi (Macro)	Jacobi (Micro)
8 PE	8 × 8	1	1	1	1
16 PEs	16 × 16	0.416	0.409	0.383	0.372
32 PEs	32 × 32	0.278	0.262	0.238	0.226
64 PEs	64 × 64	0.153	0.132	0.121	0.114

TABLE 3 : Scalability Factors in the VTD System with Differents Algorithms.

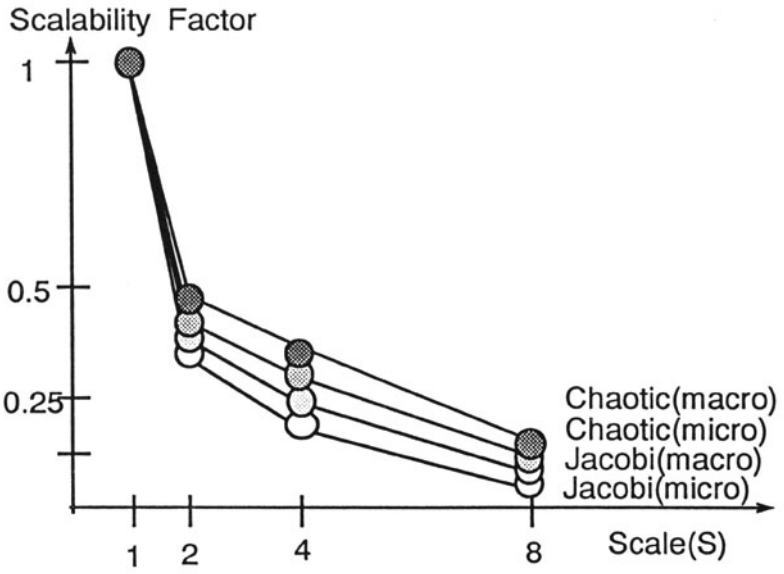


Figure 7: Scalability Factors in the VTD System.

Speedups of Various Problem Sizes					
Number of PEs	Problem Size	Chaotic (Macro)	Chaotic (Micro)	Jacobi (Macro)	Jacobi (Micro)
8 PE	8 × 8	7.15	5.54	4.28	3.45
16 PEs	16 × 16	13.45	10.34	8.18	6.37
32 PEs	32 × 32	26.26	20.30	16.05	12.21
64 PEs	64 × 64	52.15	39.77	31.49	23.70

TABLE 4 : Robustness in Data-flow Architectures.

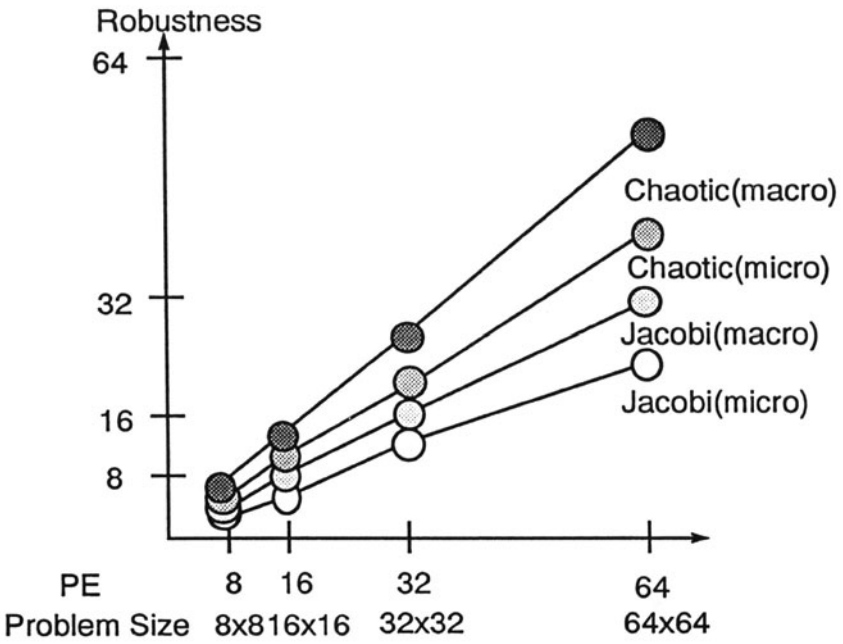


Figure 8: Robustness curves in Data-flow Architectures.

6 Conclusions

In this paper, we have demonstrated how synchronous and asynchronous linear systems solvers could be described in a high level data-flow language (SISAL) and implemented on the Variable-grain Tagged-token Data-flow (VTD) multiprocessor system in both micro and macro execution models. The conventional Jacobi method and chaotic relaxation were chosen for their known inherent parallelism of execution. While the "*conventional*" principles of the U-interpreter were used in the graph construction of the Jacobi method, chaotic behavior could not be easily realized in this model of interpretation. We therefore proposed a new scheme for the implementation of chaotic relaxation: the "*Matching Store with Locks*" scheme proceeds with the execution to detect any change on the input arcs, instead of allowing execution upon arrival of a matched token set. The new defined performance measurements *Growth Factor*, *Scalability Factor*, and *Robustness* have also characterized the system performance more precisely, besides the traditional *speedup* performance measurement in multiprocessor systems.

References

[1] Advanced Topics in Data-flow Computing. Edited by J.L. Gaudiot and L. Bic, Prentice Hall, 1990.

[2] Arvind and R.A. Iannucci. Two fundamental issues in multiprocessors: the data-flow solution. *Technical Report* LCS/TM-241, Laboratory for Computer Science, MIT,September 1983.

[3] D. Chazan and W. Miranker. Chaotic relaxation. *Linear Algebra and Application*, pages:199–222, 1969.

[4] J-L. Gaudiot and M.D. Ercegovac. Performance evaluation of a simulated data-flow computer with low resolution actors. In *Journal of Parallel and Distributed Computing*, November 1985.

[5] J-L. Gaudiot, C.M. Lin, and M. Hosseiniyar. Solving partial differential equations in a data-driven multiprocessor environment. In *Proceedings of the 15th International Symposium on Computer Architecture*, Honolulu,Hawaii, May 1988.

[6] J-L. Gaudiot and Y.H. Wei. Token relabeling in a tagged token data-flow architecture. *IEEE Transactions on Computers*, September, 1989.

[7] J. Gustafson. Reevaluating Amdahl's law. *Communication of the ACM*, May 1988.

[8] J.R. McGraw and S.K. Skedzielewski. SISAL: *Streams and iterations in a single assignment language*, language reference manual, version 1.2. *Technical Report* M-146, Lawrence Livermore National Laboratory, March 1985.

[9] R. S. Varga. *Matrix iterative analysis.* Prentice Hall, 1962.

Parallel Associative Combinator Evaluation

Martin Waite, Bret Giddings and Simon Lavington

Department of Computer Science, University of Essex,
Wivenhoe Park, Colchester,
Essex, England. CO4 3SQ.

[waitm, bret, lavis]@ uk.ac.sx

Abstract

A new evaluation model for SK combinator expressions is presented and used as a basis for the design of a novel processor. The resulting machine architecture resembles a dataflow ring, but executions are constrained to be fully lazy. When used in a multiprocessor context, different grains of parallelism are exploited at different architectural levels. A dynamic load sharing mechanism based on the current physical state of the machine is suggested. Initial simulation results are presented, and the cost-effectiveness of the proposed architecture is discussed.

1 Introduction

Several architectures have been proposed for the efficient execution of functional languages. Some researchers, inspired by the simplicity and elegance of Turner's SK graph reduction model [Tur79], have constructed hardware that yields orders of magnitude performance improvement over the original software implementation [Cla80], [Sch86]. To a large extent these efforts have been overtaken by the development of sophisticated supercombinator-based compilation techniques [Hug83] [Joh84], which give similar performance improvements on conventional microprocessors.

Attention is now focused on architectures that can exploit the inherent parallelism of the functional paradigm. These are usually divided into two broad categories - DataFlow machines, as exemplified by the MIT Tagged-Token DataFlow project [ArN87], and Parallel Graph Reduction machines, as exemplified by GRIP [Pey87]. This division tends to obscure the fact that they share the same underlying concept of reduction [ArN87]. Each approach possesses properties that assist in the design of multiprocessor architectures. Dataflow models are able to hide network communication latency [ArI87]. Graph reduction models provide a natural synchronisation and problems like cache coherence are easily handled [Pey89].

When considering computational models for the Flagship project [Wat85], the designers recognised the importance of trying to combine these properties of the two models. At the same time, however, they rejected the dataflow ring hardware design in favour of conventional microprocessors, mainly for pragmatic reasons. This view is shared by many other researchers [Pey89][Bev89], who wish to exploit recent advances in both compilation and microprocessor technology. Whilst not denying the practicality of such an approach, we tend to agree with the authors of [ArN87] with regard to the suitability of the von Neumann processor as the node in multi-node reduction architectures.

In this paper we explore another approach to the synthesis of dataflow and graph reduction principles. Starting from a new evaluation model for SK graph reduction, we develop the design of a processor which is very similar to dataflow architectures, and shares their ability to exploit fine-grained parallelism. We then show how this processor can be used as the basic node in an extensible architecture

2 The PACE Evaluation Model

The majority of SK reduction schemes use a graph of binary application nodes to represent the SK expression. This graph is alternately traversed, using either an ancestor stack or pointer reversal scheme, to assemble the next redex, and then transformed according to the relevant rewrite rule. The many memory accesses required in this process are a major source of inefficiency, and to a large extent this inefficiency is carried over into the specialised SK machines. The evaluation models employed in these machines are essentially no different from that of the abstract machine originally described by Turner, and consequently retain the influence of the sequential von Neumann processors which were the vehicles for the original software implementations.

The PACE model uses a different form of representation, which has the effect of reducing the number of memory accesses required during evaluation. It also allows the evaluation process to be described in terms of independent operations on components of the graph, rather than as a series of whole-graph transformations, thereby revealing the inherently parallel nature of the reduction process.

2.1 Representation

SK combinator expressions are represented by a collection of labelled sequences having the form *#Lm:< x ... xn >* where *#Lm* is the unique label of the sequence and *x ... xn* is the sequence itself ($0 < n < 6$).

Each item *xn* is either a data constant, an operator (SK combinator or primitive function) or the label of another sequence.

Juxtaposition indicates application according to the usual convention of left association, e.g. the sequence *< w x y z >* represents the expression ((w @ x) @ y) @ z, where @ means application.

Any SK expression can be represented by a top-level sequence together with a sequence for each bracketed sub-expression contained in the main expression. For example the pattern-directed definition of the fibonacci function

```
DEF     fib 0 = 1
        fib 1 = 1
        fib n = fib(n-1) + fib(n-2)
```

compiles to the SK expression

*T(T(M01)(M11))(S(B*PLUSfib(CMINUS1))(Bfib(CMINUS2)))*

which is represented by the following set of sequences

```
#L1: < T  #L2  #L5 >
#L2: < T  #L3  #L4 >
#L3: < M   0    1 >
#L4: < M   1    1 >
#L5: < S  #L6  #L8 >
#L6: < B* PLUS #L1 #L7 >
#L7: < C  MINUS  1 >
#L8: < B  #L1  #L9 >
#L9: < C  MINUS  2 >
```

2.2 Classification of Sequences

Sequences may be classified according to the number and type of the items they contain, as follows:

Redexes contain n items, where $1 < n < 6$, and item 1 is an operator of arity n-1, and the arguments to the operator (items 2 to n) are in the required form (i.e. fully evaluated in strict argument positions).

Oversaturated Sequences can occur in two forms:

(a) n items, where $2 < n < 6$, and item 1 is an operator of arity less than n-1

(b) n items, where $2 < n < 6$, and item 1 is a label

Weak Head Normal Forms (WHNFs) can take one of three forms:

(a) a single item which is a data constant of primitive type (number, boolean or character)

(b) three items, of which the first is the pairing constructor

(c) n items, where $0 < n < 5$, and item 1 is an operator of arity greater than n-1

Intermediate Sequences (ISs) are those sequences that cannot be classified as any of the above. These occur in two forms:

(a) n items, where $1 < n < 6$, and item 1 is an operator of arity n-1, but whose arguments (items 2 to n) are not in the required form (i.e. remain unevaluated in strict argument positions)

(b) n items, where $n < 3$, and item 1 is a label

2.3 Operations on Sequences

Evaluation can now be described in terms of three basic operations on sequences. The classification of a sequence determines the operation that may be applied to it.

Reduction is applied to redex sequences as follows:

If a sequence *<x1 ... xn>* corresponds to the left hand side of a rewrite rule, then it is replaced by a sequence that corresponds to the right hand side of that rule. If the right hand side contains bracketed sub-expressions, then an additional sequence is generated for each one. For example, *#Lm:< S a b c >* is rewritten as *#Lm:< a c #Ln >* and *#Ln:< b c >* in accordance with the rule *Sabc => ac(bc)*.

Delta redexes are rewritten in the expected way, e.g. *#Lm:< plus 3 2 >* is rewritten as *#Lm:< 5 >*.

Decomposition is applied to oversaturated sequences and involves replacing the sequence by two shorter sequences. For example *#Lm:< I a b >* is replaced by the sequences *#Lm:< #Ln b >* and *#Ln:< I a >*, and *#Lm:< #Lx a b >* is replaced by the sequences *#Lm:< #Ln b >* and *#Ln:< #Lx a >*.

Dereference is applied to a pair of sequences, one of which is an IS and one of which is a WHNF, according to the following rule:

> *If Sm is a WHNF sequence labelled #Lm, and Sn is an IS sequence labelled #Ln which contains #Lm as its leftmost label item, then the label #Lm in Sn may be replaced by the entire sequence Sm.*

Sequence *Sm* is not changed by this operation. Sequence *Sn* is changed, but its meaning remains unaltered. Note that we assume that the expression being evaluated is well-typed, and that consequently the dereference operation cannot cause *Sn* to become ill-typed.

2.4 Evaluation

The evaluation of an SK expression consists of continually applying these operations to the given (and generated) sequences until the head sequence is reduced to a WHNF. All the operations preserve meaning, and may thus be applied at any time, without any need for synchronisation. Whenever a sequence is changed (or generated) by an operation, it is simply classified according to the rules given in section 2.2, and then input to the appropriate operation.

The process may be visualised as follows:

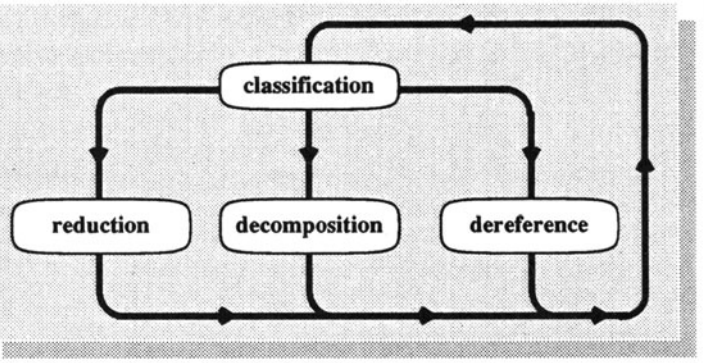


Figure 1 : Conceptual flow of sequences during evaluation.

In this abstract description of the evaluation model we can assume that resources are always immediately available to carry out the required operations. The model thus possesses the same termination properties as head reduction, in the sense that the head expression will be reduced to its WHNF if such a form exists.

3 A Hardware Implementation

The operations described in Section 2.3 are essentially very simple. It is reasonable, therefore, to consider constructing special hardware to support the operations, and to use Figure 1 as a starting point for the design of a physical (as opposed to abstract) reduction machine. This machine will take the form of a ring, containing independent units to provide the necessary operations. Sequences, in the form of packets, will be directed to appropriate units according to their current classification.

First we develop the design of a simple version of the machine. Later we refine this initial attempt in order to ensure termination properties and implement a safe form of parallelism.

3.1 Packet Format

Packets consist of a header, followed by the unique label, followed by the sequence itself. They therefore contain between 3 and 7 fields, each of which is 32 bits wide.

The header contains information about the length of the sequence, its current classification,

descriptors for each item field, (data constant, operator or label), and other control information.

3.2 Classification

Classification is carried out according to the rules given in Section 2.2. For some sequences this can be done by examining the header alone. For sequences containing an operator in the first item field, header information must be combined with information about the operator's arity, strictness and so on. This latter information is encoded into the operator's opcode in such a way as to make classification a fast bitwise logical operation. The unit that performs the classification also routes the classified sequences to their destination.

3.3 Reduction

Operators are split into three groups - SK combinators, ALU operators and others. (This latter group at present includes only the I/O primitives, but may later include operators which control other specialised devices, such as structure stores, etc.)

SK redexes are handled by a dedicated processor which implements the SK rewrite rules in the way outlined in Section 2.3. The fact that SK rewrites comprise a small fixed set of very simple manipulations is clearly appealing to hardware designers. In a sense they represent a natural instruction set for a reduction machine. At present we use Turner's set of combinators, together with some runtime optimisations which prevent the generation of oversaturated sequences in most cases.

ALU redexes are handled by an ALU manager, which accepts a packet containing an ALU operator with its arguments, sends these to the actual ALU, and builds an output packet containing the result. Note that sequences output from the ALU manager are always WHNFs. They therefore require no further classification, and can be directed immediately to the dereference unit.

A similar "manager" approach is taken for the other built-in primitives. Several of the operators in this class are required to produce some side-effect in the outside world. The issues of input/output, and the need to sometimes force an ordering on external side-effects, are best discussed within the framework of the operating system requirements for the machine. This is beyond the scope of this paper, and for simplicity we shall assume that the managers for these operators act in the same way as the ALU manager, i.e. they cause the required operation to take place and return some sequence to the dereference unit.

3.4 Decomposition

Decomposition is a relatively rare occurrence in the examples that we have examined so far, particularly if we include the runtime optimisations mentioned in section 3.3 above. The operation itself is very similar in nature to the SK rewrites, and has a similar requirement for new labels. It therefore makes sense to handle decomposition within the SK reduction unit, rather than by means of another specialised unit.

3.5 Dereference

Dereference is perhaps the most complex of the three basic operations. The essential task is that of matching the label of a WHNF with the leftmost label in an IS (which we call the *target* of the IS), and then, if the match is found, performing the desired dereference (as defined in section 2.3) In principle, either partner can arrive first, and must wait for the other. The dereference unit must therefore have a storage capability, and a fast matching capability. Since we are working in a graph reduction model, it is possible that more than one IS

shares the same target, so we must be able to perform multiple dereferences.

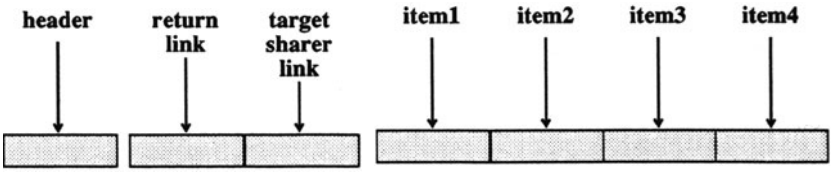
The basic algorithm for the dereference unit is as follows:

if the input sequence is classified as IS
- if the target is in memory and is WHNF
perform the dereference operation
and output the resulting sequence
else store the input IS sequence

if the input sequence is classified as WHNF
- for each stored IS having this WHNF as target
perform the dereference operation
and output the resulting sequence
- store the input WHNF sequence

The critical aspect of this unit's required functionality is the need for fast bidirectional matching. Our dereference unit (which we call the Dereference Memory, or DM unit) is clearly very similar to the wait/match stores found in DataFlow architectures. These have been identified as being difficult units to construct, requiring expensive associative hardware [Wat85]. The difficulty arises in the DataFlow model because a node in a DataFlow graph typically requires more than one data input before it can be fired. In the PACE model, an IS only receives one WHNF input during dereference, and the resulting sequence is immediately output for reclassification. In the case of ISs of the form < *SBO #Lm #Ln* > (where *SBO* is some strict binary operator) this entails extra work, but this inefficiency is more than compensated for by the fact that the DM unit can be constructed using available RAM. The required associativity can be provided by means of simple links, without any need for direct comparison or hashing of labels.

The DM unit's memory can be thought of as a heap of *slots* having the following internal structure:



(In practice this single slot would be spread across more than one memory in order to increase the potential for pipe-lining within the unit. The actual physical arrangement can be ignored for the purposes of this description.)

Labels are slot addresses, and are not held explicitly. Any stored sequence, whether WHNF or IS, occupies the slot corresponding to its own label. A slot may be *unused* (i.e. on the free list), *empty* (i.e. the label is in use but the sequence is elsewhere in the system) or *occupied* by an IS or a WHNF.

When an IS enters the store, the status of its target is determined by retrieving the target's header. If the target is in WHNF, the dereference can proceed immediately, and the result-

ing sequence is output. In this case nothing has to be stored. If the target is not in WHNF (i.e. it is either stored as an IS or it is elsewhere in the system), the incoming IS is stored in the slot corresponding to its own label and its target sharer link is set to the value found in its target's return link. The target's return link is set to be the label of the incoming IS. The list of target sharers is thus operated as a simple pushdown stack.

When a WHNF enters the store, its own return link is accessed to see if there is any IS waiting for it. If there is, a dereference can take place. While this is happening, the IS's target sharer link is examined. If it contains a label then another dereference can take place. This process is repeated until the current IS's target sharer link is found to be blank. The WHNF itself can be stored at any convenient time during this operation.

The dereference operation itself is straightforward. The merging of the two sequences is controlled by information held in the two headers (the necessary information being the length of the WHNF, and the position of the target field in the IS, which is set during classification).

3.6 Label Management and Garbage Collection

The SK Reducer requires a continuous supply of fresh labels, which is provided by a Label Manager/Garbage Collector unit. To ensure that the SK Reducer never has to pause, the Label Manager continually replenishes a buffer in the SK Reducer from the head of its free list. At this stage we have not fixed our garbage collection strategy, but we make the assumption that it can be implemented as a concurrent background process. For simplicity we have omitted reference count adjustments from our descriptions of the SK Reducer and DM unit algorithms, and for the same reason we do not show the Label Manager unit in the module diagrams.

3.7 A Simple PACE Module

In the preceding sections we have introduced hardware units to implement the basic operations required by the abstract model. We can now incorporate these units into a simple PACE module layout, as shown in Figure 2. Each unit is provided with an input buffer to smooth the flow of packets, and a main buffer is placed before the DM unit.

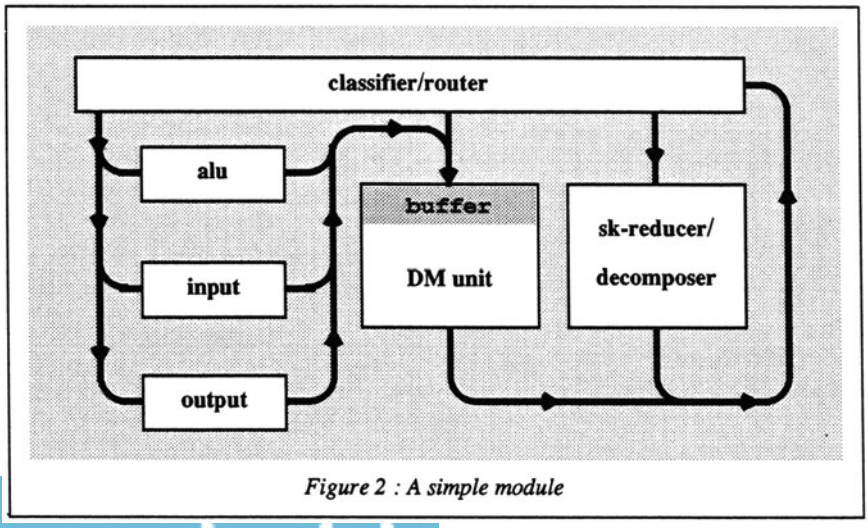


Figure 2 : A simple module

This design is an almost literal implementation of the abstract model shown in Figure 1. As such it is not practical, because it implements a fully eager evaluation process. As well as the head reduction sequence several other (possibly non-terminating) reduction sequences may be invoked. Such sequences could cause the machine to run out of resources before the head reduction sequence itself terminates. In order to retain head reduction termination properties, we have to make some changes to the model.

4 Refining the Basic Model

4.1 Implementing Lazy (Demand-Driven) Evaluation

In order to implement fully lazy head reduction, we must evaluate only the head expression (represented by the first sequence output by the SK Reducer), and delay the evaluation of any right sub-trees (represented by 2nd and 3rd sequences output by the SK Reducer) until such times as we know their value is required. We implement this as follows.

During reduction, any 2nd or 3rd sequences output by the SK Reducer are marked as *frozen* in their packet header. Following classification, these frozen sequences must be stored in the DM unit's memory until such times as they are either required or discarded.

For example, *#Lm:< S a b c > is* rewritten as *#Lm:< a c #L2 >* and *#Ln:< b c >*. The sequence labelled *#Ln* is marked as frozen. Following classification, sequence *#Lm* is sent to the unit indicated by its classification, whereas sequence *#Ln* is sent straight to the DM unit irrespective of its classification.

The DM unit's algorithm has to be extended to handle frozen sequences. We have to allow for the fact that these sequences could be redexes or oversaturated sequences as well as WHNFs and ISs. Furthermore we shouldn't assume that they arrive at the DM unit before any IS that may demand their value. To cater for this latter point, each slot in the memory is given a flag to indicate whether the value of the corresponding sequence has been demanded by being the target of some IS. The algorithm is extended as follows.

If an incoming packet is marked frozen and its demanded flag is not set, then the sequence is simply stored with no links set. If its demanded flag is set, then the frozen mark in the incoming packet's header is automatically cleared. WHNFs and ISs may then be treated just as if they never had been frozen. Redexes or oversaturated sequences are immediately output without being stored.

If an incoming IS is either not frozen, or is unfrozen on entry by the mechanism described in the preceding paragraph, then there are a number of possibilities. If the target sequence is not present, the target's demanded flag is set and the incoming IS is stored as for the original algorithm. If the target sequence is present, then the correct action depends upon the target's status. If it is classified as WHNF, dereference can proceed immediately, irrespective of the status of the target's demanded flag. If it is classified as IS, and its demanded flag has been set, the incoming IS is stored in the normal way and no further action is required. If it is classified as an IS but its demanded flag is not set, then the incoming IS is stored in the normal way, the target's demanded flag is set, and the target becomes the next incoming sequence. If it is classified as a redex or oversaturated sequence, the incoming IS is stored in the normal way, and the target sequence is immediately output with its frozen mark cleared.

These modifications change our original data-driven model to a fully lazy demand-driven model. We have regained the termination properties we desire (as well as other properties

such as the ability to handle infinite data structures), but in the process have lost parallelism.

4.2 Conservative Parallelism

In the example given above, the frozen label *#Ln* may appear as the *continuation* target of an IS (the leftmost label item in the IS), or it may appear as a *branch* target. For example, in the sequence *#La:< plus #Lc #Lb >*, *#Lc* is the continuation and *#Lb* is the branch.

The unfreezing of continuation targets is handled directly by the extension to the DM unit's algorithm given above. Demands for branch targets are handled by a Demand Control unit which is placed just before the DM unit. When the Demand Control unit discovers a label in a strict argument position that is not the continuation target, it issues a demand packet of the form DEM *<#Lb>* to the DM unit. When the DM unit receives such a packet, it outputs the demanded sequence and sets its demanded flag, unless it is already stored as a WHNF or has its demanded flag set.

When a demand releases a frozen sequence, this causes a new computational thread, (the head reduction of some sub-graph) to be started. This thread will be interleaved in the ring with the existing reduction sequences. We have achieved our objective of parallel evaluation of required values (conservative parallelism). However, unless we introduce some method for controlling the degree of parallelism, we shall reintroduce the termination problem by possibly flooding the machine with more threads than it can handle.

4.3 Counting Threads

In order to control the parallelism, we maintain an Active Thread Count (ATC), and inhibit or permit the issuing of demands according to its current value. Maintaining the ATC is straightforward - the releasing of a frozen sequence represents the starting of a new thread, the storing of a WHNF represents the completion of a thread, the storing of an IS represents the suspension of a thread, and the release of a suspended IS as a result of dereference represents the resumption of a thread. In many cases, increments and decrements cancel each other out and no change has to be made to the current ATC (e.g. when an incoming IS finds its target already in WHNF). Since all these operations are performed by the DM unit, it is the DM unit that is given responsibility for maintaining the ATC.

The ATC is held in a register that can be read by the Demand Control unit. The Demand Control unit will only issue demands for branch targets when the current ATC is below some preset limit. This simple mechanism not only provides control over the evaluation, it also provides us with some sort of metric for the module's current workload.

4.4 A Revised Module Design

Although the dereference algorithm is now more complex than for the original model, there are still only a few basic operations being performed. The additional complexity is mainly a matter of condition testing. Careful selection of classification codes, etc., allows much of this condition testing to be implemented by means of fast parallel logical operations. A hardware implementation is still viable, and in Figure 3 we show a revised module design that incorporates the changes made in this section.

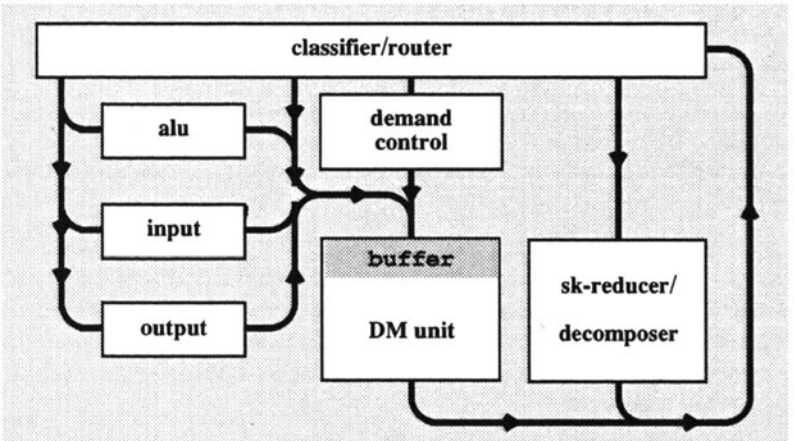


Figure 3 : The parallel demand-driven configuration.

4.5 Loading the Expression

Now that we have adopted a lazy model, the collection of sequences that constitutes the compiled code for the program must be marked as frozen before being loaded through the Classifier/Router. In order to start the execution, a DEM sequence containing the label of the head expression is added to the collection of sequences. Strictness analysis of the source program can be used to indicate which other parts of the overall expression can be safely evaluated. By including DEM sequences for these sub-expressions, we can extend the available parallelism.

4.6 Filling the Pipe

In order for the module to achieve its maximum reduction rate, its various units must be working concurrently. This is only achieved when there are sufficient active threads. Figure 4 below shows the relationship between the reduction rate and the preset limit to the ATC for executions of the fibonacci function on a typical module configuration.

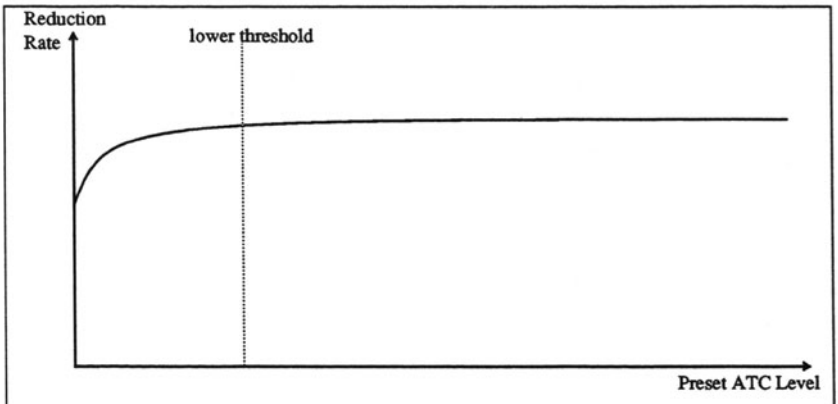


Figure 4 : Relationship between reduction rate and preset ATC level

Beyond a certain lower threshold, the number of currently active threads doesn't affect the reduction rate. The maximum reduction rate is fixed by the throughput rate of the DM unit. Once this maximum rate has been achieved, the effect of adding more threads is simply to increase the size of the input queue of this unit. In examples examined so far the relationship between the number of active threads and the maximum queue size has been linear. The size of the input buffer provided for the DM unit therefore fixes an upper limit to the number of active threads that can be supported at any one time.

5 An Extensible Architecture

In this section we address the question of extensibility, that is whether several modules can be combined so as to give a performance speedup which is linear with the number of modules used.

5.1 Some Preliminaries

We segment the label space by using the most significant part of the label to indicate a module, and the remaining part to indicate the slot number within the module. At present we devote 16 bits to the slot number, giving 64K slots per module. We assume that the definition part of the program is replicated on each of the modules, and that each copy of a particular function definition sequence occupies the same slot within each module. References to definition sequences are distinguished in the packet headers, i.e. the descriptor for an item may now indicate that it is a data constant, an operator, a *transient* label or a *definition* label. We introduce three new units into the module design - an Entry unit, an Exit unit and a Local Indirection Cache. The extended design is shown in Figure 5.

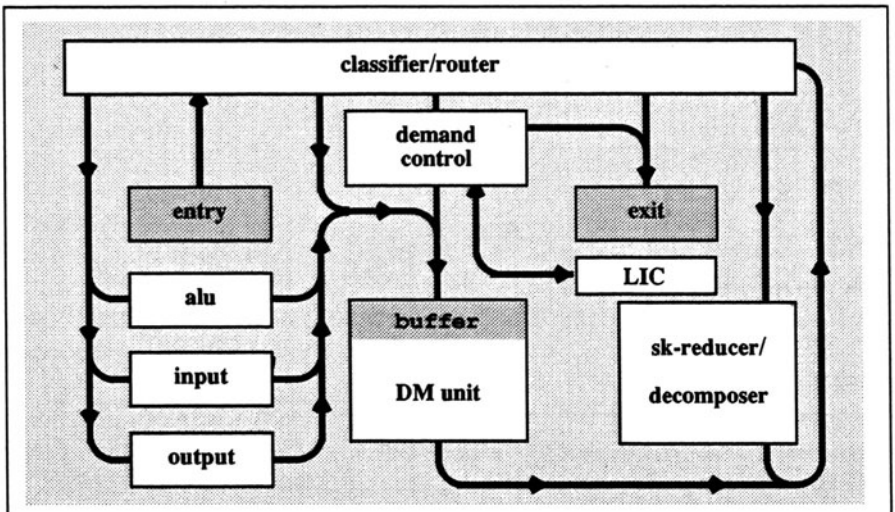


Figure 5: A Module for the extended architecture

We assume that the modules are connected in a way that enables any module to despatch a sequence to any other named module. We further assume that the communication medium is aware of current ATC levels, and is able to direct sequences to the module with the lowest current ATC.

5.1 Export, Return and Demand Mechanisms

Export

When an IS arrives at the Demand Control unit, it represents an ongoing computational thread. This thread can be exported by simply transferring the IS to another module. We proceed as follows.

When the Demand Control unit recognises that an IS should be exported it sends the IS to the Exit unit. The Exit unit decrements the local ATC, and puts the IS onto the communication medium which directs the sequence to the module with the lowest current ATC. When it receives an incoming IS, the Entry unit on the receiving module increments its local ATC. Any definition labels in the sequence are converted to local references by having the local module number masked onto the module part of the label. The sequence is sent to the Classifier/Router and from then on is treated just as any other sequence until such a time (if any) that it has to be stored as an IS. The DM unit recognises it as a foreigner from its label, swaps its label for a newly issued local label, stores the IS in the corresponding slot, and places its original label in the return link. The thread is now established on the destination module under a pseudonym label.

Return

When the exported and re-labelled sequence finally achieves WHNF, the DM unit recognises the return link as belonging to another module, and swaps the labels back, returning the pseudonym label to its free list. The sequence is immediately output to the Classifier/Router which recognises it as a foreign WHNF, marks it as having been "evaluated elsewhere", and routes it towards the Exit unit. When the sequence arrives back at its module of origin, it is treated exactly as any other WHNF, except that it does not cause an ATC decrement when it is stored.

Demand

When the Demand Control unit encounters an IS with a foreign target, it first searches the Local Indirection Cache. This is a fast associative store containing <foreign label, local label> pairs. If the foreign label is present, the associated local label is substituted for the foreign label in the IS. If it is not found, then a newly generated label is substituted for the foreign label, the new pair is entered into the cache and an IS of the form *#Ll:<#Lf>* (where *#Ll* is the newly generated local label and *#Lf* is the foreign label) is sent to the Exit unit and thence to the relevant module. In either case the modified IS is sent to the DM unit where it is stored in the normal way. When the sequence *#Ll:<#Lf>* arrives at its destination, it is treated just as any other incoming IS.

The mechanisms for export, return and demand are simple extensions to the existing model, and the overhead incurred is negligible. Nevertheless we must endeavour to keep such communications to a minimum.

5.2 Export and Branch Heuristics

To achieve linear speedup, each of the modules must be performing at its maximum reduction rate at any point during execution. This does not imply that threads have to be evenly distributed across the machine, but only that each module's ATC is kept above the lower threshold level shown in Figure 4.

When we commence execution, we would like each module to receive a thread as soon as possible. During the course of the execution, in order to maintain locality of reference, we

would prefer each module to maintain its own ATC level as a result of local branching rather than as a result of having threads exported to it. If each module's ATC is at a healthy level, it might be advantageous to delay branching, so as to maintain locality, rather than branch and be forced to export because of overloading. If at some point any module's ATC becomes dangerously low, the system must be able to react quickly by exporting work to it immediately. All this suggests that different run-time situations could be ranked in terms of the need to branch and the need to export. Furthermore, it suggests that ATC levels could be used to provide an up-to-date ranking value for the current situation.

Similarly, we could rank ISs according to how useful their export might be. For instance, a sequence like *#Lm:<#Lh x>* is more likely to spawn new threads on the receiving module if *#Lh* is the label of the head sequence of a divide-and-conquer function definition than it is if *#Lh* is the label of some other function definition sequence. Since labels are slot numbers rather than word addresses, we could afford to use part of the 32 bit label to encode this information, which could be derived from a static analysis of the program.

At the point when an IS enters the Demand Control unit, both sorts of information could be combined to decide whether it is appropriate to export the thread, or branch a new thread, or neither. Just how this might be achieved, and how it might be efficiently implemented at the hardware level, is the subject of our present investigations.

Clearly the question of export and branch heuristics is complex, but we have good reasons to be optimistic about this approach. Experience with heuristics in other contexts seems to indicate that simple heuristics very often yield near optimal results. Also, we are working with a tolerant and flexible model, in which "tasks" are interleaved reduction sequences. On the one hand we can allow several such tasks to become suspended before a module becomes idle. On the other hand, we can move a task from one module to another very rapidly at any point during its execution.

6. Simulated Performance

6.1 The Simulator

We have written a software simulator to enable us to examine the behaviour of the proposed architecture. The simulator resides in an environment which allows us to create different module configurations interactively and save them on file. We are able to write programs in SASL, compile them to the PACE format, load and run them on the configuration of our choice, and extract statistics. Using a monitoring/debugging tool [Hea90] we are able to observe the flow of packets, the activity of each unit, where bottlenecks occur, and so on.

The concurrent activity of the various units is modelled using an event queue. Events include requests to transmit sequences from one unit to another as well as the actual processes performed by each unit. It is therefore able to correctly model contentions on the (single) input to each unit. At present we assume that each unit has its own input buffer which is implemented as a queue, ordered on time of arrival.

In order to keep the simulator as simple as possible, we have used a module configuration similar to the one shown in Figure 5, but with two Classifier/Routers to ease the flow of packets. Connections between units are modelled as single 32 bit highways. Our basic unit of time is the clock period, or in other words the time taken to transmit one 32 bit word from one unit to another.

We have allowed certain straightforward operations within a process (e.g. condition test-

ing) to occur in parallel within one clock period, but have otherwise been cautious. Although there seems to be plenty of scope for pipelining within units, the simulator insists that any output sequences generated by a particular unit must be despatched to their destination before the unit can accept an incoming sequence from its input buffer.

The present simulator thus represents a very simple hardware implementation of the PACE model, and doesn't necessarily provide a very accurate prediction for the absolute performance of a fully engineered version. It does, however, fulfil its main purpose of enabling us to study the behaviour of a multi-node architecture.

6.2 Results

Initial results obtained from the simulator are encouraging. We have examined three aspects - the efficiency of the PACE evaluation model, the absolute performance of a single module, and the speed-up obtained in multi-module configurations. For all these experiments the size of the main buffer was limited so as to hold a maximum of 64 sequences at any one time.

Evaluation Model

We compared run-time statistics from a single module simulation with statistics extracted from a conventional SASL implementation [HaV88]. The table below shows an analysis of combinator usage during execution of fibonacci 6, using the pattern-directed definition given earlier.

	Operators	I	S S'	B B'	C C'	M	T	Dec
PACE	17.73	0	5.9	5.9	11.8	24.6	22.2	5.9
SASL	15.5	17.7	5.2	5.2	10.4	21.6	19.4	0

Figure 6 : Percentage of combinators executed

On average, three dereference memory operations were required for each reduction in the PACE model. One of the major sources of inefficiency in the binary graph model, the creation and subsequent reduction of indirection nodes, is completely eliminated. Since the I combinator doesn't appear in the function definition, no I reductions are performed in the PACE model. This has the effect of slightly increasing the percentage of useful operator reductions, even though we include decompositions in the PACE totals. The pattern-matchers T,M and U do not fit comfortably in the PACE model, and direct support for them will probably be abandoned in favour of compilation to conditionals. This will simplify the hardware design and at the same time yield more efficient computations (in executions of nfib using a conditional definition, useful operators accounted for over 30% of reductions).

Absolute Performance

To measure the absolute performance we used the nfib benchmark as defined in Version 0.99 of LML [AuJ90]. Running nfib28 on one module, with an assumed clock period of 100 nanoseconds, we recorded a reduction rate of 520K reductions per second, and a function call rate of 55K calls per second. According to figures quoted in [AuJ90] this is roughly equivalent to a sun 3/50 executing compiled LML.

We regard this figure as promising, since there appears to be considerable scope for improvement on the simple hardware design that we are currently simulating. For example,

the main constraint on the performance of a single module appears to be the bandwidth between units, and this could be increased significantly by using multi-layering techniques. We believe that by incorporating this sort of feature in a more sophisticated design, we could achieve execution times similar to those currently obtained by SPARC-based implementations.

Speedup

Figure 7 shows the speedup achieved for various executions of the nfib benchmark. For these simulations we assumed that the modules were connected to a bus. For inter-module communications, we assumed that each 32 bit word in the sequence took 200 nanoseconds to reach its destination and included a 200 nanosecond overhead for each communication.

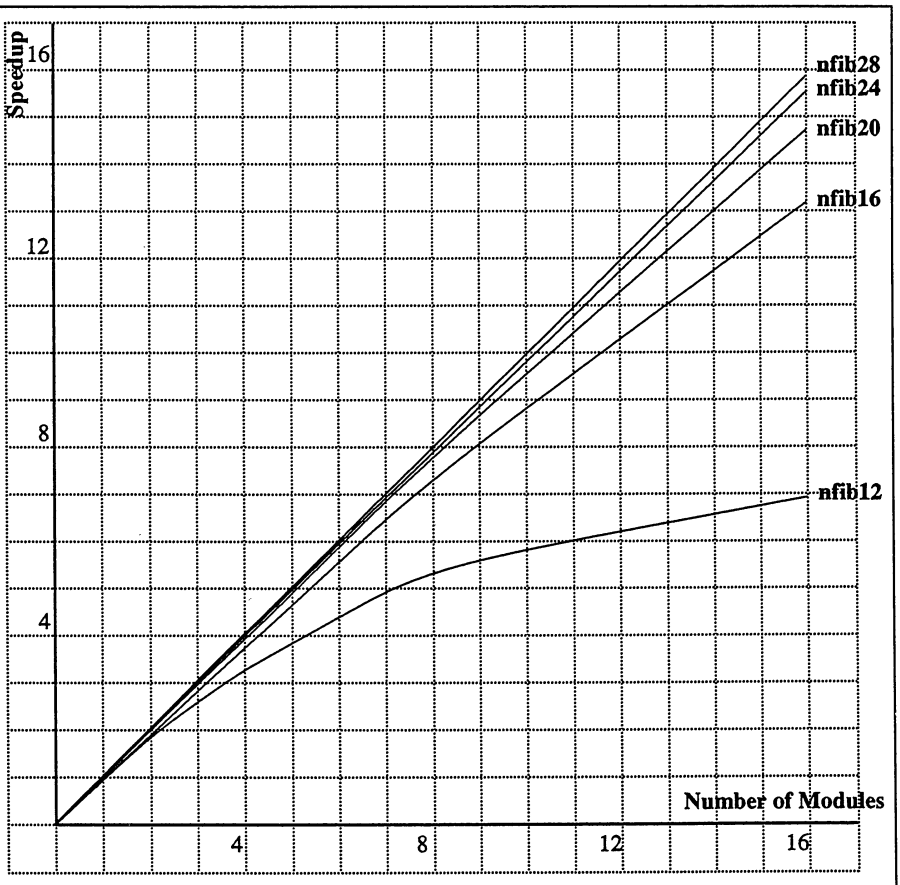


Figure 7 : Relationship between Number of Modules and performance speedup.

At nfib28 the speedup curve is indistinguishable from the ideal linear relationship (executing nfib28 on 16 modules gave a 99.14% efficient speedup). Further analysis is required to see how close the figures are to the theoretical maximum speedups. The maximum bus usage recorded was 63.5% busy for nfib12 on 16 boards.

These figures were obtained without any form of programmer annotation concerning task grain size. The compiler marked the head sequence of the nfib function definition as being the only candidate for export, and the branching and exporting of tasks was controlled entirely by the ATC-based heuristics outlined in Section 5.2

7. Conclusions

In recent years, the development of sophisticated compilation techniques has led to a marked improvement in the efficiency of functional language implementations. In the course of this development the SKI combinators have come to be regarded as inefficient and too fine-grained to be of practical use.

In the context of producing implementations for the von Neumann processor these criticisms are valid, and become even more so when considering implementations for multiprocessor architectures. If, however, we implement the abstract SK machine directly in hardware, instead of regarding it as a stage in the compilation to von Neumann code, we are presented with a different perspective.

The core of any functional language implementation is concerned with the task of reconstructing the right hand side of a function definition, with actual arguments substituted for bound variables, and then (or in the same process) evaluating the resulting expression. In current supercombinator-based implementations this is achieved by executing a sequence of von Neumann instructions. In the PACE processor, it is achieved by performing a sequence of SK reductions. If we compare the efficiencies of von Neumann instructions and SK reduction steps in fulfilling this basic task, then we see that the SK reductions are really quite efficient. Earlier we noted that over 30% of reductions performed during execution of the conditional version of nfib were useful arithmetic operations, the remainder being reductions that guided the arguments to the appropriate operators. It would be interesting to compare this ratio with the ratio of useful arithmetic operations to load/store/register moves in executions of the von Neumann code produced by a supercombinator-based implementation.

Of course, a comparison at this level only makes sense if the SK reductions can be performed at a sufficiently high rate. In [NoP88], the authors compare executions of a G-machine implementation with those of a SASL implementation. They note that there is a rough equivalence between the number of G-machine instructions and the number of SK reductions for the test programs they examined. This would suggest that a reduction rate in the order of several millions per second is required to rival the performance of a supercombinator implementation running on a single SPARC processor.

We believe that a fully engineered version of the PACE design would be capable of this sort of reduction rate. Clearly the design effort required would not be justified if the end result was a processor with the same capability as an existing product. However, our motivation is not to design a novel uniprocessor, but to develop a suitable processor for an extensible and scalable parallel architecture. In this context the PACE design appears to have significant advantages.

The overheads of task creation, task scheduling and network communication present problems for designers of parallel graph reduction implementations on current distributed architectures. In order to maintain a viable ratio between these overheads and the actual compu-

tation, the size of individual tasks needs to be quite large. This requirement introduces some adverse effects. Firstly there is a loss of available parallelism. (In [NoP88] the authors note that the SASL code exhibited a much higher degree of parallelism than the G-machine code.) Secondly, there is the problem of ensuring that task sizes are kept above some minimum level. Since it is in general impossible to determine the size of a task at the compilation stage, there is a growing tendency to require the programmer to supply extra information to the run-time system by means of annotations to the source text. Not only are such annotations machine-dependent, they clearly represent an unwelcome complication for the programmer.

In the PACE model, the creation and suspension of tasks is an integral part of the basic evaluation mechanism. A newly created task does not have to be scheduled as a separate process, but is simply interleaved with existing tasks on the same processor. When a task is suspended there is no costly context switch. As long as there are sufficient tasks, their creation and suspension has no effect on the useful work rate of the processor. This no-cost multi-tasking capability means that it is viable (indeed desirable) to spawn many small tasks, and thereby exploit the high degree of parallelism observed by the authors of [NoP88]. Furthermore, since a small task size can be tolerated, there is no need for the programmer to consider this aspect of the underlying architecture.

The use of interleaved reduction sequences also has an impact on task distribution in a multi-processor architecture. New tasks are spawned and become immediately active on the same processor as the task that demanded their evaluation. Locality of reference is thus maintained by default. It is only broken when tasks have to be moved in order to maintain minimum task levels (ATCs) on each processor. The high task generation rate helps to maintain local ATCs, and so task movements do not have to occur at the same rate as task generation. In other words the fine grain of parallelism does not imply a need for massive inter-module bandwidth. The actual mechanisms for exporting tasks, demanding remote values and returning results are simple, they integrate smoothly with the processor's internal evaluation mechanisms, and they operate concurrently with the ongoing computation. Just as with task creation, task movement can take place without affecting the processor's useful work rate.

8. Future Work

In this paper we have presented some results from a preliminary investigation into the behaviour of the proposed PACE architecture. In order to establish the viability of this approach, our immediate task is to run a much wider range of benchmark programs, particularly those that involve the manipulation of data structures. Instead of relying on simple delta parallelism, we intend to make use of strictness annotations in order to raise the amount of available parallelism. Our main goal in this line of research will be the development of more general load balancing heuristics. We also need to undertake a preliminary hardware design investigation, in order to see if the performance required from a single PACE module can be achieved. Later, if these initial studies prove successful, we shall move on to a concrete design proposal for a fully distributed architecture.

References

[ArN87] Arvind and R.S.Nikhil,
"Executing a program on the tagged-token dataflow architecture",
Proc. PARLE (Parallel Languages and Architectures, Europe)
Conference, Eindhoven. LNCS Springer Verlag, 1987.

[ArI87] Arvind and R.A.Ianucci,
"Two Fundamental Issues in Multiprocessing",
CSG Memo 226-6, Lab for Computer Science, MIT, 1987.

[AuJ90] L.Augustsson and T.Johnsson,
Version 0.99 of LML
Department of Computer Science, Chalmers University of Technology, Gothenburg, 1990.

[Bev89] D.J.Bevan, G.L.Burn, R.J.Karia and J.D.Robson,
"Principles for the Design of a Distributed Memory Architecture for Parallel Graph Reduction",
The Computer Journal, Vol 32 No 5, 1989.

[Cla80] T.J.Clarke, P.J.S.Gladstone, C.D.MacLean and A.C.Norman,
"SKIM - the S K I reduction machine",
Proc. ACM Lisp Conference, Stanford CA, 1980.

[HaV88] P.H.Hartel and A.H.Veen,
"Statistics on Graph Reduction of SASL Programs",
Software - Practice and Experience Vol 18(3), 239-253, March 1988.

[Hea90] M.J.Heaton,
"Pacetool: A Monitoring Tool for PACE",
Department of Computer Science, University of Essex, September 1990.

[Hug83] J.Hughes,
"The Design and Implementation of Programming Languages",
Ph.D. Thesis, Oxford University (1983).
(Published as Oxford University Computing Laboratory, Programming Research Group, Technical Monograph PRG-40, September 1984.)

[NoP88] E.Nocker and R.Plasmeyer,
"Combinator Reduction on a Parallel G-Machine",
in Parallel Processing and Applications,
E.Chiricozzi and A.D'Amico (eds), (North-Holland), 1988, 399-412

[Pey87] S.Peyton-Jones, C.Clack, J.Salkild and M.Hardie,
"GRIP : A high performance architecture for parallel graph reduction",
Functional Programming Languages and Computer Architecture, ed Kahn,
Portland, Oregon. LNCS 274 Springer Verlag, 1987.

[Pey89] S.Peyton-Jones,
"Parallel Implementations of Functional Programming Languages",
The Computer Journal, Vol 32, No 2, 1989.

[Joh84] T.Johnsson,
"Efficient Compilation of Lazy Evaluation",
Proc. ACM SIGPLAN '84, Symposium on Compiler Construction,
SIGPLAN Notice, Vol 19, No 6, June 1984.

[Sch86] M.Scheevel,
"NORMA - a graph reduction processor",
Proc. ACM Conference on Lisp and Functional Programming, Aug 1986.

[Tur79] D.A.Turner,
"A New Implementation Technique for Applicative Languages",
Software - Practice and Experience, Vol 9, 31-49, 1979.

[Wat85] I.Watson, P.Watson and V.Woods
"Parallel Data-Driven Graph Reduction",
University of Manchester, 1985.

Static Analysis of Term Graph Rewriting Systems

Chris Hankin
Dept. of Computing, Imperial College of Science, Technology and Medicine
180 Queen's Gate, LONDON SW7 2BZ, UK

ABSTRACT

In this paper we present a framework for the abstract interpretation of term graph rewriting systems. The framework is based on the approach taken by the Cousots for flowchart programs. We give an example of the use of the framework by presenting an interpretation which performs a form of type inference.

1. Introduction

In Term Graph Rewriting Systems (TGRS) [Bar87] rewrite rules relate (rooted) graphs rather than the finite trees which are used in Term Rewriting Systems(TRS). The main distinction between TGRS and TRS is that sharing is explicitly captured in TGRS by shared subgraphs and, since cyclic structures are permitted, finite term graphs can represent infinite terms. There is a strong correspondence between computation in a TGRS and in a modern graph reduction machine.

Viewing a graph rewrite system as a program raises the question of compilation and thus optimisation. Many of the classical analyses of functional programs [AH87] appear to be applicable in this new context. There may be some advantage to performing analyses at this intermediate level because, as it is "closer" to the machine, some opportunities for optimisation may be exposed which are hidden at the source language level. In this paper we will be concerned with semantics-based compile-time analysis of rewrite systems. The sort of

properties that we will be interested in include the following: Strictness/Neededness [Bar87a]; Store Usage [Hud87]; Complexity [GH85], [San90]; Relevant Rules [Mis84]; and Types [Ban89].

The rest of this paper is organised into three main sections. In the next section we introduce the necessary details of TGR systems; this work has been reported in [Bar87] and [Ken88] and the interested reader is referred to those sources for further information. Section 3 sets up a framework for abstract interpretation; we present a "collecting" semantics [Cou77] - the most precise semantics which records complete information about possible executions of a program. The final main section presents an approximation of the collecting semantics - an abstract interpretation - which provides information similar to that produced by the analysis of [Ban89]. Section 5 contains some conclusions and directions for future work.

2. Term Graph Rewriting

The definitions presented in this section are based on [Bar87]. We give an operational account of TGR; [Ken88] gives a categorical semantics for a similar system. Throughout this section we will use the following fragment of a (term) rewrite system as a running example:

upto 0	→	nil
upto (succ n)	→	concat (upto n) (cons (succ n) nil)
sum (cons a x)	→	a + (sum x)
sum nil	→	0
Init	→	sum(upto 4)

The intention is that *upto* generates a list of integers from 1 upto the given parameter using the built-in concat and cons "functions"; *sum* adds the elements of a list together; and *Init* defines the top-level expression to be evaluated. Notice that the example is first-order and presented in a functional notation [Klo85]; this is not a limitation of our general approach but the particular analysis of Section 4 is not relevant for applicative-style systems.

The main objective of TGR is to explicitly capture sharing as part of the rewrite mechanism. In TGR, rewrite rules effectively relate pairs of term graphs.

Definition 2.1 (Term Graphs)
A term graph is represented by the quadruple:

(N,lab,succ,r)

where N is a set of nodes

lab : N → Symbol is a partial labelling function (nodes corresponding to variables are not labelled, i.e. lab(n) = ⊥ for those nodes)

succ : N → N* is the node successor function (_* forms sequences)

r is a distinguished node called the *root*

We write Gln to denote the subgraph of G rooted at the node n.

◊

We will present an example shortly, but first we define the form that rules have in TGR:

Definition 2.2 (Term Graph Rewrite Rules)

A term graph rewrite rule is a triple:

(g,n,n')

where g is a graph, i.e. a structure (N,lab,succ)

n is a node in g representing the root of the left hand side of the rule

n' is a node in g representing the root of the right hand side of the rule

such that all variable nodes appear in gln.

◊

We illustrate these two definitions by showing the TGR representation of the second rule for the *upto* function. In term graph diagrams we will represent nodes by boxes, the name of a node may be written beside the box, the lab function is represented by writing symbols inside boxes and the succ function is represented by arrows between boxes. The diagram for the *upto* rule is:

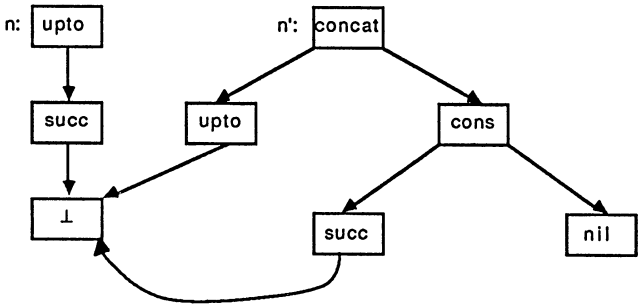


Fig. 1 A Rewrite rule

Notice that the graph structure explicitly captures the sharing that is implicit in the term rule.

In order to define a notion of reduction it is necessary to introduce a mechanism for identifying redexes in a graph. In order to do this, we have to introduce a number of definitions.

Definition 2.3 (Homomorphisms)

Given two term graphs, G and H, a mapping $g : G \rightarrow H$ (defined on the nodes of G and H) is homomorphic at a node n in G if:

either $lab_G(n) = \perp$

or $lab_H(g(n)) = lab_G(n)$

$g(succ_G(n)_i) = succ_H(g(n))_i$ for all i, $succ(n)_i$ is the i-th successor of n

◊

Definition 2.4 (Total Homomorphisms)

Given two term graphs, G and H, and a mapping $g : G \rightarrow H$, g is a total homomorphism if it is homomorphic at every node of G.

◊

Definition 2.5 (Redex Occurrence)

A redex occurrence in some graph G is a total homomorphism from the left hand side of some rule to G.

◊

For example there is a redex occurrence of the illustrated rule in the initial graph as shown in Figure 2.

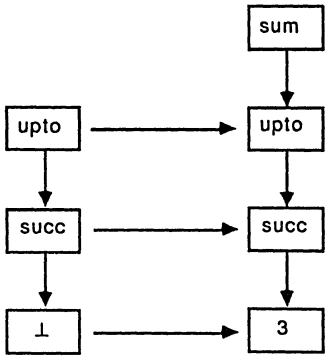


Figure 2: A redex occurrence

Graph rewriting is a three phase process: the *build* phase, the *redirection* phase and the *garbage collection* phase.

During the build phase, copies of the nodes in the right hand side of the rule which do not appear in the left hand side are added to the graph; pointers to variable nodes are fixed up to point at the matching subgraph. The result of the build phase is shown in Figure 3.

Definition 2.6 (The Build Phase)
Let H be the result of the build phase applied to a graph G, rewrite rule (g,n,n') and redex occurrence $f : g \rightarrow G$. We write:

$$H = G +_f (g,n,n')$$

and H is defined as follows:

$$N_H = N_G + (N_{g|n'} - N_{g|n}) \quad \text{+ is disjoint union, - is set difference}$$

$$lab_H(m) = \begin{cases} lab_G(m) & m \in N_G \\ lab_g(m) & \text{otherwise} \end{cases}$$

$$succ_H(m)_i = \begin{cases} succ_G(m)_i & m \in N_G \\ succ_g(m)_i & \text{both m and } succ_g(m)_i \text{ in } N_{g|n'} - N_{g|n} \\ f(succ_g(m)_i) & \text{m in } N_{g|n'} - N_{g|n} \text{ and } succ_g(m)_i \text{ in } N_{g|n} \end{cases}$$

$$r_H = r_G$$

◊

During the redirection phase all pointers to the root of the left hand side are redirected to point to the copy of the root of the right hand side. If the image of the root of the left hand side is the root of the whole graph, the root of the resultant graph will be the copy of the root of the right hand side of the rule; otherwise the overall root is unchanged.

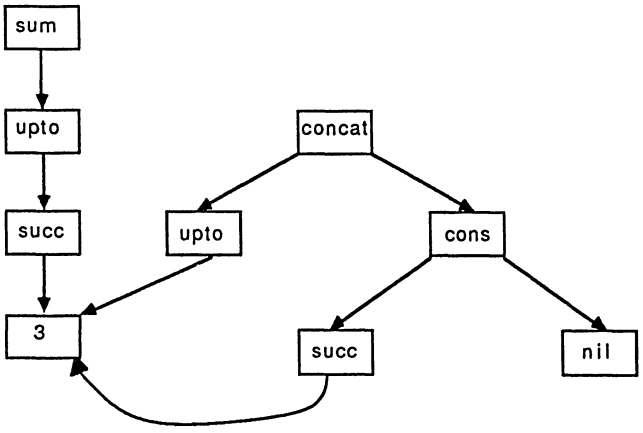


Fig 3. The Build phase

Definition 2.7 (The Redirection Phase)

Define H[a := b] to be the term graph (N_H,lab,succ,r) such that:

lab(n)	=	$lab_H(n)$	for all nodes in N_H
$succ(n)_i$	=	b	if $succ_H(n)_i = a$
		$succ_H(n)_i$	otherwise
r	=	b	if $r_H = a$
		r_H	otherwise

If H is the result of the build phase for an occurrence f and rule (g,n,n'), then:

J	=	H[f(n) := f(n')]	if $n' \in gln$
		H[f(n) := n']	otherwise[1]

is the result of the redirection phase and we write:

J = H[f,(g,n,n')]

◊

The final phase is garbage collection which simply discards all nodes which are not accessible from the root of the graph.

Definition 2.8 (The Garbage Collection Phase)

For some term graph G the result of garbage collection, written GC(G), is defined as:

$GC(G) = G \mid r_G$

◊

In our example, garbage collection removes the leftmost *upto* and *succ* nodes.

3. A Collecting Semantics

A collecting semantics collects complete information about possible executions of a program [Cou77]. Abstract interpretations may be expressed as abstractions of, and proved correct with respect to, a collecting semantics. In the classical setting, the collecting semantics associates a context vector, a mapping from program points to sets of possible environments, with a program. The collecting semantics is defined in terms of a state transition function.

[1] This definition is slightly different from that given in [Bar87] which does not seem to handle selector rules correctly.

In our presentation, we make use of the following two sets:

Rule the set of rules; elements are triples as presented in the last section

Graph the set of graphs; elements are quadruples (N,lab,succ,r)

We use the notation:

$$\text{Graph} \rightarrow \text{Graph}$$

to represent the set of total graph homomorphisms. We define the next "state" for term graph rewriting by formalising the notion of reduction strategy. Following [Bar87], a strategy is a mapping from a graph to a set of reduction sequences; we will represent a reduction sequence as a sequence of pairs:

(Occurrence,Rule)

Thus the sequence:

$$(g_1,r_1)(g_2,r_2)\ldots(g_n,r_n)$$

associated with some graph, G_1, stands for a reduction sequence of n reductions steps:

$$G_1 \rightarrow_{g_1,r_1} G_2 \rightarrow_{g_2,r_2} \ldots G_n \rightarrow_{g_n,r_n} G_{n+1}$$

where g_i is an occurrence in G_i of the left hand side of r_i. The formal type of strategies is:

$$S \in \text{Strategy} = \text{Graph} \rightarrow 2^{((\text{Graph} \rightarrow \text{Graph}) \times \text{Rule})^*}$$

We require that all sequences are nonempty [Bar87]. A strategy is *deterministic* if for all graphs, G, S(G) contains at most one element. S is a *one step strategy* (or *1-strategy*) if for all graphs, G, every member of S(G) has length 1[2]. If G is in normal form, then S(G) will be empty.

Next we define our notion of state. To record the state of the system, it is sufficient to know the current state of the top-level graph. Thus we have:

$$st \in \text{State} = \text{Graph}$$

For now we will restrict our attention to 1-step, deterministic strategies and we will abuse notation by writing:

$$(f,r) = S(G)$$

to identify the single (Occurrence,Rule) pair when S(G) is nonempty. The next state function, nstate, is defined as follows:

$$\text{nstate} : \text{State} \rightarrow \text{State}$$

$$\begin{aligned}
\text{nstate}(G) = \ & \textbf{let } seq = S(G) \\
& \textbf{in } \ \textbf{if } seq = \emptyset \\
& \qquad \textbf{then } G \\
& \qquad \textbf{else } \ \textbf{let } (f,r) = seq
\end{aligned}$$

[2]An example of a strategy which is not a 1-strategy is a strategy which reduces each redex to root-stable form (the graph analogue of weak head normal form)

in GC((G $+_f$ r)[f,r])

We are now ready to define the collecting semantics; we follow [Jon87] in identifying program points with rules. We consider a set of m rules $\{(g_i,n_i,n'_i) \mid 0<i\leq m\}$ and an initial graph, Init, which we assume is not in normal form. We associate a "context" [Cou77] with each rule which is a set of pairs, where each pair specifies a graph and an occurrence of the left hand side of the rule in that graph:

$C_r \in$ Context = $2^{\text{Graph} \times (\text{Graph} \rightarrow \text{Graph})}$

$C_r = \{(G,f) \mid n \geq 0, G = \text{nstate}^n(\text{Init}), (f,r) = S(G)\}$

Thus a context specifies, for each use of the rule, which graph was being reduced and which redex within the graph was contracted. A context vector associates a context with each rule:

Cv $\in$ Context-vector = Rule $\rightarrow$ Context

and we define the vector associated with a program to be:

$Cv = \lambda r.C_r$

This semantics is similar to that presented in [Jon87] but, in contrast to Jones' semantics ours is more directly based on the Cousots' work using state transition functions and also takes the reduction strategy into account.

The set of context vectors forms a complete lattice with the following characteristics:

$\bot = \lambda r.\emptyset$

$\top = \lambda r.\{(G,f) \mid G \in \text{Graph}, r = (g,n,n'),$ f an occurrence of g\|n in G$\}$

$Cv_1 \cup Cv_2 = \lambda r.Cv_1(r) \cup Cv_2(r)$

$Cv_1 \cap Cv_2 = \lambda r.Cv_1(r) \cap Cv_2(r)$

$Cv_1 \geq Cv_2 \Leftrightarrow \forall r \in \text{Rule}.\ Cv_1(r) \supseteq Cv_2(r)$

It will be convenient in the later development to have a fixed point definition of the Context vector associated with a program. We define the function:

F : Context-vector $\rightarrow$ Context-vector

$F(Cv) = \bigcup_{r \in \text{Rule}} \bigcup_{(G,f) \in Cv(r)}$(

let H = nstate(G) **in**

if S(H) = $\emptyset$

then $\bot$

else ([s := {(H,h)}] **where** (h,s) = S(H)))

$\cup$ ([t := {(Init,k)}] **where** (k,t) = S(Init))

where [s := ...] is a "small" context vector which maps the rule s to the specified context and every other rule to the empty context:

$\lambda r \in$ Rule.**if** r=s **then** {(H,h)} **else** $\emptyset$

note that the union operation is the operation from the lattice of context vectors. A consequence of this definition is the following:

Proposition 3.1

F is continuous

$\Diamond$

Thus F has a least fixed point which may be found by an iteration starting with the bottom context vector.

Proposition 3.2

fix $F = \lambda r.C_r$

Proof

We show by mathematical induction that:

$F^{i+1}(\perp) = \lambda r.\{(G,f) \mid 0 \leq n \leq i, G = nstate^n(Init), (f,r) = S(G)\}$

Basis: $F^1(\perp) = \bigcup_{r \in Rule} \bigcup_{(G,f) \in F^0(\perp)(r)}$ (

let H = nstate(G) **in**

if S(H) = $\emptyset$

then $\perp$

else ([s := {(H,h)}] **where** (h,s) = S(H)))

$\cup$ ([t := {(Init,k)}] **where** (k,t) = S(Init))

= ([t := {(Init,k)}] **where** (k,t) = S(Init))

$= \lambda r.\{(Init,k) \mid (k,r) = S(Init)\}$

$= \lambda r.\{(Init,k) \mid Init = nstate^0(Init), (k,r) = S(Init)\}$

Inductive Step:

Since we are considering deterministic 1-strategies, there is at most one G in $F^{i+1}(\perp)$ such that:

$G = nstate^i(Init)$

We therefore consider two cases:

(1) There is no such G

Then there is some normal form graph H such that:

$H = nstate^k(Init)$

form some $k \leq i$. $F^{i+2}(\perp)$ adds no new context vectors and the result follows trivially.

(2) There is such a G

Let H = nstate(G)

Again there are two cases to consider:

(2a) $S(H) = \emptyset$

$F^{i+2}(\perp)$ adds no new context vectors and the result follows trivially.

(2b) $S(H) = (h,s)$

$$F^{i+2}(\perp) = F^{i+1}(\perp) \cup [s := \{(H,h)\}]$$
$$= F^{i+1}(\perp) \cup \lambda r.\{(H,h) \mid H = nstate^{i+1}(Init), (h,r) = S(H)\}$$
$$= \lambda r.\{(G,f) \mid 0 \le n \le i, G = nstate^{n}(Init), (f,r) = S(G)\}$$
$$\cup \lambda r.\{(H,h) \mid H = nstate^{i+1}(Init), (h,r) = S(H)\}$$
by I.H.
$$= \lambda r.\{(G,f) \mid 0 \le n \le i+1, G = nstate^{n}(Init), (f,r) = S(G)\}$$
by defn. of $\cup$

The main result then follows by taking the least upper bounds of both sides of the equation.
◊

4. Approximating the Collecting Semantics

4.1 Banach's Analysis

The analysis in [Ban89] produces all possible (S,k,T) triples such that a node with label T can appear as the k-th successor of a node with label S during some reduction sequence:

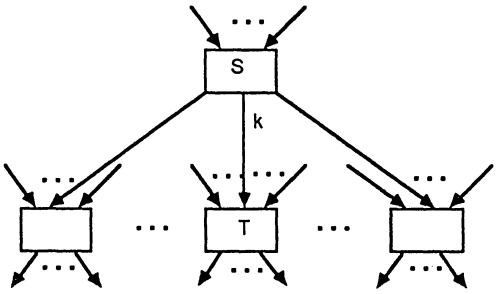


Fig 5. An SkT triple

The analysis is guaranteed to be finite by ensuring that the set of symbols is finite; under the assumption that the only infinite subset of symbols is the integers, Banach does this by defining abstract graphs such that:

lab n	=	(lab n) if (lab n) is not an integer, **Int** otherwise
succ n	=	succ n

The analysis is a form of type inference. For example, consider the program presented in Section 2; the symbol "sum" will have four triples associated with it as a result of the analysis:

{(sum,1,concat), (sum,1,cons), (sum,1,nil), (sum,1,upto)}

from which, combined with an syntactic analysis of the rules, it can be inferred that sum is a function on lists. This is rather incomplete type information but it is sufficient to provide information to a more sophisticated analysis, presented in [Ban89], which allows atomicity conditions on rewrite rules to be relaxed. Thus the analysis is a vital component of an analysis which leads to efficient parallel implementations of graph rewriting languages.

4.2 Banach's Analysis as an Abstract Interpretation

The approach of abstract interpretation involves the development of a semantics which approximates the collecting semantics and gives the required information. In the literature, we can identify two relevant approaches: grammar-based analyses as presented in [Jon87] and the more traditional approach based on Galois connections [Cou77] or logical relations [Abr90].

The former approach involves abstracting the collecting semantics by some form of grammar. For term rewriting systems this grammar normally takes the form of a regular tree grammar. Program properties are then expressed as decidable questions about the tree grammar. This approach is treated in detail in [Jon87], the results are encouraging and should be investigated further in the context of TGR systems. For the purposes of this paper, we will concentrate on the second approach using Galois connections.

A Galois connection between a pair of complete lattices is a pair of functions (α,γ) such that the following properties are satisfied:

α and γ are monotonic

$\alpha \circ \gamma \leq$ identity

$\gamma \circ \alpha \geq$ identity

In abstract interpretation, we normally use (α,γ) pairs such that the first inequality becomes an equality. The intention is that one lattice represents values in a real (concrete) computation, call it C, and the other lattice contains some abstract representation of the values, call the lattice A. Then $\alpha : C \rightarrow A$ is an *abstraction* function and $\gamma : A \rightarrow C$ is a *concretisation* function. Abstract interpretation proceeds by abstracting the initial state and then performing some abstract computation, the end result of which gives the required information. The correctness criterion for the analysis is that concretisation of the end result should give a superset of the values produced by the real computation.

The abstract interpretation will abstract a context vector by a set of triples:

$SkT = (Symbol \times Integer \times Symbol)$

$Acontext\text{-}vector = 2^{SkT}$

where Symbol is the finite set of node labels and elements in the set SkT are used to represent the same information as that produced by Banach's analysis. We ensure that Symbol is finite by using the same device as [Ban89] and assigning the symbol **Int** to any node labelled by an integer. The function $\underline{lab}$ maps each node to its associated abstract symbol:

$\underline{lab}(n) =$ **Int** if lab(n) is an integer, lab(n) otherwise

We will assume that symbols have a fixed arity, given by the following function:

$\rho : Symbol \rightarrow Integer$

These two constraints together ensure that the set SkT is finite. Acontext-vector is clearly a complete lattice.

In order to construct the abstraction and concretisation maps, we need to be able to abstract a single context with respect to a specified rule. This operation is achieved by the abs function:

$abs : Graph \times (Graph \rightarrow Graph) \times Rule \rightarrow Acontext\text{-}vector$

$$abs(G, f, r) = \bigcup_{m \in Glf(n)} \{(\underline{lab}(m), i, \underline{lab}(m_i)) \mid 0 < i \leq \rho(\underline{lab}(m)), succ(m)_i = m_i\}$$

where $r = (g,n,n')$

The abstraction map is defined as follows:

$\alpha : Context\text{-}vector \rightarrow Acontext\text{-}vector$

$$\alpha(Cv) = \bigcup_{r \in Rulc} \bigcup_{(G,f) \in Cv(r)} abs(G,f,r)$$

and concretisation is defined as follows:

$\gamma : Acontext\text{-}vector \rightarrow Context\text{-}vector$

$$\gamma(Av) = \bigcup \{ Cv \in Context\text{-}vector \mid \alpha(Cv) \leq Av \}$$

We then have the required adjointness property for our abstraction and concretisation maps:

Proposition 4.1

(α,γ) is a Galois connection and, moreover α and γ are continuous.

◊

The abstract interpretation involves a fixed point computation over abstract context vectors. We will construct it in a way which should be reminiscent of the *nstate* function of Section 3.

We need to introduce a number of definitions and auxiliary functions before we can define the abstract interpretation.

An assignment from an abstract contect vector is a mapping that maps variable nodes to symbols. If a particular variable node is the i-th child of a node labelled S and there is an (S,i,T) triple in the vector, then the assignment can map the variable node to the symbol T. An assignment is *consistent* if, whenever the variable node is a child of more than one node, there are appropriate (S_j,i_j,T) triples in the abstract context vector (i.e. all of the triples contain the same symbol T). We define the predicate *is-assignment* which returns true when a consistent assignment exists:

$$\text{is-assignment} : \text{Acontext-vector} \times \text{Graph} \rightarrow \text{Boolean}$$

$$\begin{aligned}\text{is-assignment}(Av,G) = {} & \forall n \in G.\forall 0<i\leq\rho(\underline{lab}(n)).\\ & ((\underline{lab}(succ(n)_i) = \perp) \Rightarrow\\ & \exists T.\forall m \in G.\forall 0<j\leq\rho(\underline{lab}(m)).\\ & \quad (succ(m)_j = succ(n)_i) \Rightarrow\\ & \quad (\underline{lab}(m),j,T) \in Av)\\ & \text{and}\\ & ((\underline{lab}(succ(n)_i) \neq \perp) \Rightarrow\\ & \quad (\underline{lab}(n),i,\underline{lab}(succ(n)_i)) \in Av)\end{aligned}$$

When consistent assignments do exist, the function *assignments* generates a "small" function which maps the variable nodes to the set of possible symbols which might be assigned to them:

$$\text{assignments} : \text{Acontext-vector} \times \text{Graph} \rightarrow (\text{Node} \rightarrow 2^{\text{Symbol}})$$

$$\begin{aligned}\text{assignments}(Av,G) = {} & \bigcup_{n\in G} \bigcup_{0<i\leq\rho(\underline{lab}(n))}\\ & \textbf{if } \underline{lab}(succ(n)_i) \neq \perp\\ & \textbf{then } \emptyset\\ & \textbf{else } \{[succ(n)_i = \{T \mid \forall m \in G.\forall 0<j\leq\rho(\underline{lab}(m)).\\ & \qquad (succ(m)_j = succ(n)_i) \Rightarrow\\ & \qquad\quad (\underline{lab}(m),j,T) \in Av\}]\}\end{aligned}$$

Finally we need to apply the given assignment and generate the set of (S,k,T) triples present in the resultant graph; this is done by the function *triples*, it takes two parameters, a graph and an assignment:

$$\text{triples} : \text{Graph} \times (\text{Node} \rightarrow 2^{\text{Symbol}}) \rightarrow 2^{SkT}$$

$$\begin{aligned}\text{triples}(G,f) = {} & \bigcup_{n\in G} \bigcup_{0<i\leq\rho(\underline{lab}(n))}\\ & \textbf{if } \underline{lab}(succ(n)_i) \neq \perp\\ & \textbf{then } \{(\underline{lab}(n),i,\underline{lab}(succ(n)_i))\}\end{aligned}$$

else $\{(\underline{lab}(n),i,S) \mid S \in f(succ(n)_i)\}$

Armed with these definitions we can define abstract versions of the three phases of graph rewriting.

Definition 4.2 (The Abstract Build Phase)

Given an abstract context vector G, an assignment f and rule r (= (g,n,n')):

$G \pm_f r = G \cup triples(g|n',f)$

◊

Definition 4.3 (The Abstract Redirection Phase)

Given an abstract context vector G, an assignment f and rule r (= (g,n,n')):

$G[\underline{f,r}] = G \cup \{(S,k,T) \mid (S,k,\underline{lab}(n)) \in G$ and
$(T=\underline{lab}(n')$ or $T \in f(n'))\}$

◊

Definition 4.4 (The Abstract Garbage Collection Phase)

Given an abstract context vector G:

$\underline{GC}(G) = G$

i.e. abstract garbage collection is the identity function.

◊

Finally we define the function $\underline{F}$, which models the function F (defined in Section 3):

$\underline{F}$: Acontext-vector → Acontext-vector

$\underline{F}(Av) = \bigcup_{r \in Rule}$ (if is-assignment(Av,g|n)
then $\underline{GC}((Av \pm_f r)[\underline{f,r}])$
else Av
where r = (g,n,n')
and f = assignments(Av,g|n))
$\cup$ triples(Init,λr.Ø)

$\underline{F}$ is clearly continuous and thus the abstract context vector associated with a program, which is the least fixed point of $\underline{F}$, may be found by iterating from Ø. Notice that our abstract interpretation takes no account of the reduction strategy or pattern matching; this is equivalent to Banach's analysis but there may be better analyses which do take some account of the strategy.

We close this section with the safety theorem. We need a number of technical results before we can state and prove the safety theorem. The first lemma establishes the monotonicity of the operations used in the abstract interpretation. We assume the following orderings for assignments and graphs:

$f' \geq f \Leftrightarrow dom(f') \supseteq dom(f)$ and $(\forall n \in dom(f).\ f'(n) \supseteq f(n))$

$G' \geq G \Leftrightarrow \exists n \in G'.\ G = G' \mid n$ i.e. G is a subgraph of G'

Lemma 4.5

1. triples is monotonic
2. assignments is monotonic
3. the abstract rewriting operations are monotonic

◊

For a graph G, occurrence f and rule r (= (g,n,n')), we define

assoc(G,f,r) = $\lambda m \in g|n.$ **if** $\underline{lab}(m) = \perp$ **then** $\{\underline{lab}(f(m))\}$ **else** $\emptyset$

and we call assoc(G,f,r) the assignment associated with the occurrence.

Lemma 4.6

For all graphs G, occurrences f and rules r such that f is an occurrence of r in G:

is-assignment(T,g|n) and (assignments(T,g|n) ≥ assoc(G,f,r))

where T = triples(G,λr.$\emptyset$)

◊

The next three lemmas relate the abstract and concrete rewriting operations.

Lemma 4.7

Let f be an occurrence of some rule r (= (g,n,n')) in some graph G. Let

T = triples(G,λr.$\emptyset$)

and f' = assignments(T,g|n)

then $T \pm_{f'} r \supseteq$ triples(G $+_f$ r, λr.$\emptyset$)

◊

Lemma 4.8

Let f be an occurrence of some rule r (= (g,n,n')) in some graph G. Let

T = triples(G,λr.$\emptyset$)

and f' = assignments(T,g|n)

then $T[\underline{f'},r] \supseteq$ triples(G[f,r], $\lambda r.\emptyset$)

◊

Lemma 4.9

$\underline{GC}$(triples(G,$\lambda r.\emptyset$)) $\supseteq$ triples(GC(G),$\lambda r.\emptyset$)

◊

The last lemma establishes a relationship between the abstract interpretation and the collecting semantics.

Lemma 4.10

$\forall i \geq 0.\ \forall r \in$ Rule. $\forall (G,f) \in F^i(\perp)(r)$.

is-assignment($\underline{F}^i(\perp)$,gln) and $\underline{F}^i(\perp) \supseteq$ triples(G,$\lambda r.\emptyset$)

where r = (g,n,n')

Proof

By induction.

Basis:

trivial

Inductive Step:

By definition, since we are considering deterministic 1-strategies, $F^{i+1}(\perp)$ has at most one new element. If it has none we are done. Otherwise, the new element will be of the form:

[s := {(H,h)}]

where H = nstate(G) for some $(G,f) \in F^i(\perp)(r)$ and h is an occurrence of some rule s.

Let r = (g,n,n') and s = (k,m,m').

Then is-assignment($\underline{F}^i(\perp)$,gln) by I.H.

Let f' = assignments($\underline{F}^i(\perp)$,gln)

then $\underline{F}^{i+1}(\perp) \supseteq \underline{GC}((\underline{F}^i(\perp) \pm_{f'} r)[\underline{f'},r])$ by defn.

$\supseteq \underline{GC}((\text{triples}(G,\lambda r.\emptyset) \pm_{f'} r)[\underline{f'},r])$

by I.H. and Lemma 4.5

$\supseteq$ triples(H,$\lambda r.\emptyset$) by Lemmas 4.7, 4.8, 4.9

and is-assignment($\underline{F}^{i+1}(\perp)$,klm) by Lemma 4.6

◊

Finally, we have:

The Safety Theorem

γ (fix $\underline{F}$) $\geq$ fix F

Proof

First, it follows as a direct corollary of Lemma 4.10 and the fact that:

$$abs(G,f,r) = triples(Glf(n), \lambda r.\emptyset)$$

that for all $i \geq 0$: $\underline{F}^i(\perp) \geq \alpha(F^i(\perp))$

We now concretise both sides of the inequality:

$\gamma(\underline{F}^i(\perp)) \geq \gamma(\alpha(F^i(\perp)))$ by monotonicity of γ

$\geq F^i(\perp)$ by Proposition 4.1

and take the least upper bounds of both sides:

$\bigcup_i \gamma(\underline{F}^i(\perp)) = \gamma(\bigcup_i \underline{F}^i(\perp))$, by continuity of γ $= \gamma(\mathbf{fix}\ \underline{F})$, by defn

$\geq \bigcup_i F^i(\perp)$, lhs $= \mathbf{fix}\ F$, by defn

◊

5. Conclusions

We have sketched the development of a systematic approach to the static analysis of TGR systems and illustrated it in an application related to type checking. Our main contribution is that we have shown how the standard framework for abstract interpretation can be applied to Term Graph Rewriting Systems; the details of the particular abstract interpretation are secondary.

The style of the analysis presented in this paper is operational. [Ken88] presents a categorical semantics for graph rewriting in which the rewriting process is modelled by a pushout construction. This formalism allows elegant proofs for some standard results (e.g. Church Rosser properties) and may hold the key to a more algebraic approach to static analysis.

The approach to abstract interpretation which is based on the analysis of certain forms of grammar derived from a collecting semantics also leads to a very different style of analysis and is worthy of further investigation.

Acknowledgements

Thanks are due to Sebastian Hunt for his encouragement and constructive criticism of earlier

drafts of this paper and to Richard Banach, Jesper Jørgensen, Muffy Thomas and the PARLE reviewers for helpful comments. The author was partially funded by ESPRIT Basic Research Action 3074 : SemaGraph - The Semantics and Pragmatics of Generalised Graph Rewriting.

References

[AH87] Abramsky S. and Hankin C. L.(eds) *Abstract Interpretation of Declarative Languages*, Ellis Horwood, 1987.

[Abr90] Abramsky S. *Abstract Interpretation, Logical Relations and Kan Extensions*, to appear in Logic and Computation, **1** 1.

[Ban89] Banach R. *Dataflow Analysis of Term Graph Rewriting Systems*, PARLE '89, Volume II, LNCS 366, Springer Verlag, pp 55-72.

[Bar87] Barendregt H. P., Eekelen M. C. J. D., Glauert J. R. W., Kennaway J. R., Plasmeijer M. J. and Sleep M. R. *Term Graph Rewriting*, PARLE '87, Volume II, LNCS 259, Springer Verlag, pp 141-158.

[Bar87a] Barendregt H. P., Kennaway J. R., Klop J. W. and Sleep M. R. *Needed Reduction and Spine Strategies for the Lambda Calculus*, Information and Computation, **75** 3, pp 191-231.

[Cou77] Cousot P. and Cousot R. *Abstract Interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points*, 4th POPL, pp238-252, 1977.

[GH85] Goldberg B. and Hudak P. *Serial Combinators: Optimal Grains for Parallelism*, 2nd FPCA, LNCS 201, Springer Verlag, pp 382-399.

[Hud87] Hudak P. *A semantic model of reference counting and its abstraction*, in [AH87], pp45-62.

[Jon87] Jones N. D. *Flow analysis of lazy higher-orderfunctional programs*, in [AH87], pp103-122.

[Ken88] Kennaway J. R. *On "On Graph Rewritings"*, Theoretical Computer Science, **52**, pp37-58.

[Klo85] Klop J. W. *Term Rewriting Systems*, Notes for the seminar on graph reduction machines, Ustica, September 1985. A revised version to appear in the "Handbook of Logic and Computer Science", Oxford University Press, 1991.

[Mis84] Mishra P. and Keller R. M. *Static inference of properties of applicative programs*, 11th POPL, 1984.

[San90] Sands D. *Complexity Analysis for a Lazy Higher-order Language*, ESOP '90, LNCS 432, Springer Verlag, pp 361-376.

Scheduling of OR-parallel Prolog on a Scalable, Reconfigurable, Distributed-Memory Multiprocessor *

J. Briat, M. Favre, C. Geyer†
Projet CMaP, LGI
BP 53 X,
38041 Grenoble Cedex
E-mail: {briat, favre}@imag.fr

J. Chassin de Kergommeaux
CAP-Gemini-Innovation
7, chemin du Vieux Chêne, ZIRST
F-38240 Meylan
E-mail: chassin@capsogeti.fr

Abstract

The OPERA project aims at efficiently implementing Prolog on a scalable, reconfigurable distributed-memory architecture. The OPERA computational model exploits OR-parallelism following a classical multisequential approach: each processor executes a complete Prolog engine based on the WAM; inter-processor communication is reduced to work installation, the complete state of an active Prolog engine being copied to an idle one. Scheduling is performed by a hierarchy of specialized processors, operating in parallel of the computation of the Prolog program. To avoid costly synchronization, schedulers use an approximate representation of the state of the system. Because of the important overhead of task installation in a distributed-memory system, only workers having a large amount of work to execute can give work to idle workers. Several dynamic work regulation strategies have been designed and are currently being tested. The prototype implementation of OPERA on a transputer-based Supernode is one of the most efficient existing Prolog implementations on the transputer and reaches effective speed-ups in parallel over efficient sequential Prolog systems.

Keywords: OPERA, OR-parallel Prolog, WAM, distributed-memory, Supernode, scalable multiprocessor, reconfigurable multiprocessor.

*This work has been partially sponsored by the Centre National d'Etude des Télécommunications (CNET) and ESPRIT project P1085.

†University of Porto Alegre, CPGCC-USRGS, caixa postal 1501, 90000 Porto Alegre RS Brasil, geyer@SBU.USRGGS.ANRS.BR

1 Introduction

The aim of the OPERA project is to use the important computing power offered by scalable distributed-memory multiprocessors to efficiently execute Prolog programs in parallel. The transputer architecture is suitable to the building of inexpensive massively parallel architectures such as the dynamically reconfigurable Supernode [HJMWS86], in the development of which part of the OPERA team was involved. Prolog has been chosen because of the inherent parallelism of the language, which allows the execution of ordinary Prolog programs in parallel. Contrary to the Concurrent Logic Languages approach [Sha89], which aims at developing new languages for parallel processing, the goal of OPERA is to speed-up the computation of "standard" Prolog programs by use of parallelism. Among the potential sources of parallelism offered by Prolog programs, OPERA exploits OR-parallelism, expected to require less communications than AND-parallelism and therefore be more suitable for existing distributed-memory architectures.

The computational model of OPERA follows a classical multisequential approach: each processor (worker) executes a Prolog engine which computes portions of the search tree defined by the program. Parallelism is initiated by otherwise idle workers: no more parallelism than available resources is generated. The OPERA Prolog engine is based on the most efficient compilation technique used for Prolog, known as the *Warren Abstract Machine* (WAM) [War83]. The WAM has been extended into a "TWAM" (*Transputer Warren Abstract Machine*), tailored to a distributed-memory architecture. When an "idle" worker takes work from an active one, the stacks describing the state of the computation are *copied* from the active to the "idle" worker.

Scheduling is one of the most important issues to be solved by parallel Prolog systems. The scheduler must keep the workers as busy as possible while limiting the overhead generated by parallelism. The first constraint implies generating enough parallelism but the second implies that the granularity of each new parallel activity remain bigger than the overhead it introduces. Scheduling is more important in distributed-memory than in shared memory architectures because of the higher cost of task creation in the former. Scheduling is also more difficult in distributed-memory architectures because of the lack of global state for the parallel Prolog system, unless extremely costly synchronization algorithms are used. The computational complexity of task scheduling increases with the number of processors used by the multiprocessor, since the amount of exploited parallelism should increase as well. Because of the complexity of scheduling, important computing resources are devoted to schedule tasks in OPERA. The structure of the OPERA scheduler is hierarchical and mimics the architecture of the control network of the Supernode, used for the implementation. Specialized scheduler processes keep an approximate record of the states of their slave schedulers or workers and balance the workload among these slaves. Several load balancing strategies, based on dynamic criteria, have been designed and are currently being tested.

A prototype implementation of OPERA is now running on Supernode. The sequential efficiency of this implementation is one of the best sequential Prolog implementations on transputer. First experimental results show good performance speed-ups for parallel computations over sequential ones, for programs providing a large enough search space.

The organization of this paper is the following. After this introduction, the architecture of the Supernode is briefly described. The following section presents the computational model of OPERA. The scheduling issues and their solutions in OPERA are then described. The next section sketches the current implementation and gives the preliminary results obtained by the OPERA prototype. The two last sections of the paper compare OPERA to similar approaches and conclude the paper.

2 Architecture of the Supernode

The target multiprocessor architecture of the OPERA project is the Supernode [HJMWS86] developed in the ESPRIT project 1085. The Supernode architecture is a dynamically reconfigurable array of transputers. It is based on a general purpose parallel processor, T800 floating point transputer, together with a specific programmable switch allowing dynamic reconfiguration of the links between worker transputers. The Supernode architecture is a two-level architecture, where Tnodes can be considered as building blocks for two level multiprocessors called Meganodes.

A basic Tnode module is composed of 16 working transputers in the basic version and 32 transputers in the "tandem" version. Interprocessor communication is provided by a crossbar switch controled by a specialized control transputer. Working transputers are connected to the control transputer by a dedicated control bus, used to pass communication requests as well as miscellaneous control commands for the working transputers (initialization, termination, etc.). A 16 transputer Tnode is represented in figure 1.

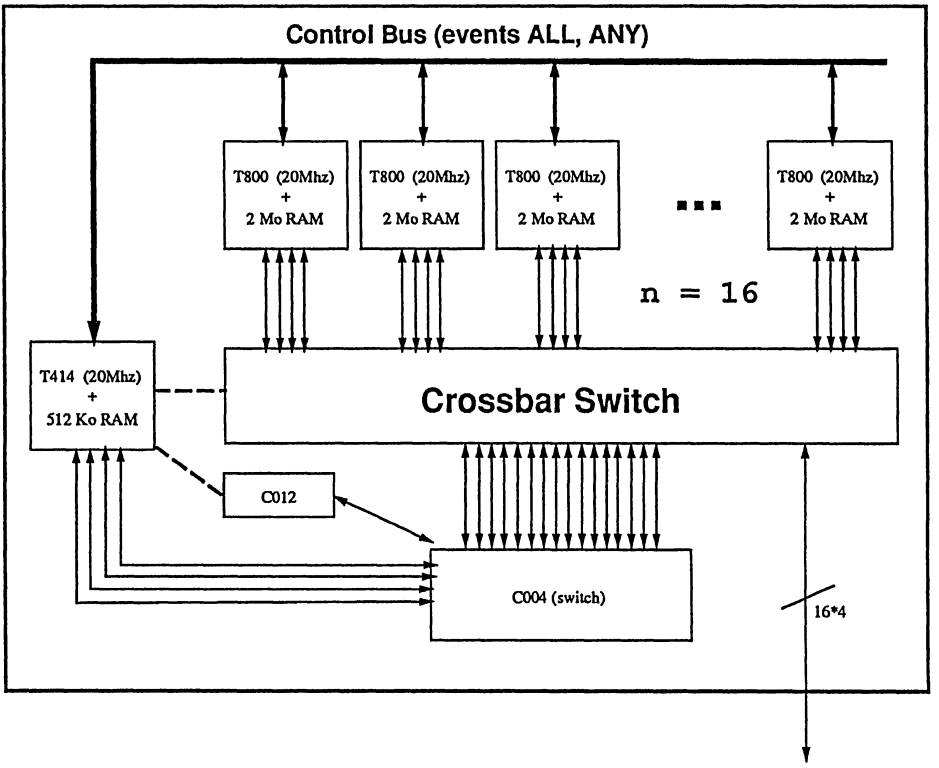
Several Tnodes can be interconnected by a second-level network of crossbar switches in order to build a Meganode. The structure of the network of switches of a Meganode is organized as a three stage Clos network [CLO84] in which the crossbar switches of the Tnode modules participate as first and third stages. The network of switches is controled by a hierarchy of control tansputers interconnected by a hierarchy of identical control buses.

The interconnection network topology can be defined statically as a four degree graph: two dimensional mesh, tore, etc. Connections can also be modified dynamically, depending on the needs of the working transputers. Interconnection requests are then transmitted using the control bus. In any case, the control of the connections is left to the application which must implement a deadlock-free connection algorithm in the control transputers.

The exchange of small data messages between working transputers is very inefficient compared to the copy of large blocks of memory. This inefficiency stems from the inefficiency of the initialization of the connection before the actual transfer of the message. The connection of two working transputers, belonging to the same Tnode, will take in the order of 250 micro-seconds in the best case and much more in the worst case. Once a direct connection is established, the time taken to transfer n bytes between two working transputers in the same Tnode is [TP90]:

$$T_{trans}(n) = 4.85 + 1.125 \times n\ \mu s.$$

When the message transfer takes place between two transputers belonging to different



On request from two processing transputers, the control transputer T414 connects them directly through the crossbar switch. Sixty four links connect the Tnode to the external world, used to interconnect Tnodes to obtain a Meganode. The Meganode architecture is scalable up to 1024 T800 processors.

Figure 1: Architecture of the Supernode

Tnodes, the initial connection time is much longer, while the relation between the transfer time and the message length is:

$$T_{trans}(n) = 5.61 + 2.2 \times n\ \mu s.$$

Because of the inefficiency of the dynamic interconnections at the second level, OPERA uses static interconnections between Tnodes.

Transfer of messages is performed in parallel to normal computation by the sender and receiver working transputers, using a DMA mechanism. Since DMA accesses to memory are interleaved with local accesses, the overhead of the copy operation remains limited for both transputers.

The software environment available for the Supernode at the beginning of the OPERA project was very limited. Therefore, important efforts had to be devoted to basic system work at the beginning of the project [BFF+89] to be able to use the Supernode.

3 The OPERA computational model

3.1 OR-parallelism in Prolog programs

Potential OR-parallelism appears during the computation of a Prolog program when several clause heads can be unified with the current goal. In a sequential system, clauses are evaluated following their lexical order in the program, a new clause being tried after a backtracking operation. In an OR-parallel Prolog system, these clauses are tried in parallel. For example, let p(X,Y), q(Z,X) be the current resolvent, p being defined by:

p(X,Y) :- ..., X=a, s_1(X,Y).
p(X,Y) :- ..., X=b, s_2(X,Y).
p(X,Y) :- ..., X=c, s_3(X,Y).

All three clauses can be executed in parallel in an OR-parallel Prolog and different *bindings* for X passed to the s_i and q. Even if the "size" of p is small, the granularity of OR-parallel tasks can be important if solving q requires important computations: all parallel branches will indeed compute q in parallel for different values of X.

3.2 Overview of the OPERA computational model

The computational model of OPERA is multi-sequential: each processor executes a Transputer Warren Abstract Machine (TWAM) on a local copy of the Prolog program. The TWAM is based on the Warren Abstract Machine (WAM) [War83], regarded as the most efficient implementation technique for Prolog. OPERA executes sequential (deterministic) programs at an efficiency close to the best sequential Prolog implementation based on the WAM. Idle workers request work from active workers having untried alternatives recorded in choice points. New tasks are created on idle workers, using complete choice-points of active workers.

When taking work from an active worker, an idle worker copies the state of the active TWAM, recorded in several stacks, when the choice-point providing work was created. Variables created before this choice-point but bound after must be unbound. To distinguish valid and invalid variable bindings, all bindings are tagged with a date. Other solutions performing sharing of stacks [War87,WRCS87] cannot be applied because of the inefficiency of small message passing, compared to copying of large chunks of data, in a distributed-memory architecture such as the Supernode.

3.3 Process structure of a worker (TWAM)

Each worker is composed of four processes: the *Solver*, the *Exporter*, the *Importer* and the *Spy* (see figure 2). The Solver implements a sequential Prolog engine similar to the WAM. The Exporter transfers stacks when exporting work to the Importer of an idle worker (see section 3.5). The Spy informs the scheduler of the state of the TWAM (see section 4).

3.4 TWAM data structures

The local and global stacks of the WAM have been reorganized into four stacks in the TWAM. Choice points are managed in a special stack of the TWAM, instead of being interleaved with clause activation records in the local stack of the WAM. The main advantage of this solution is to isolate the communication between the export and the solver processes, which takes place when an untried alternative is exported. Another additional data structure is the variable stack, used to store all the Prolog variables. Each variable of the variable stack is tagged with a "date", which is actually the depth of the current OR-node in the search tree as in the Kabu-Wake model [Mea86]. Grouping the variables in a special stack improves the efficiency of the copy operation (see section 3.5), at the expenses of memory usage and access time to the variables (one more level of indirection is required).

3.5 Copying of stacks

When backtracking, the Solver of a worker W_2 may find an empty choice point stack and therefore become *idle*. The Importer of W_2 then requests work from its Scheduler and waits until it is granted work from an export worker W_1. The relevant portions of the stacks of W_1 are then copied on W_2. The order of the copy operations is the following (see figure 2):

1. The transfer of stacks using DMA mechanisms is initiated by the Exporter of W_1 and the Importer of W_2.

2. The portion of the variable stack existing at the creation of the choice-point having given work is copied from the workspace of W_1 to the workspace of W_2. The use of dates allows the Solver of W_1 to remain active during the whole copy operation, without requiring any synchronization, when binding a variable on one of its stacks.

3. The relevant portions of the other stacks of W_1 are copied on W_2 while, *simultaneously*, the Importer of W_2 tests the date of each variable copied in phase 2 to check whether it belongs to the resolvent being copied. Variables bound after the creation of the choice-point of W_1 which has given work, are reset by the Importer of W_2.

4. When the transfer is over, the Exporter of W_1 and the Importer of W_2 acknowledge the transfer to the Scheduler and the Solver of W_2 resumes work.

Since the Solver and the Importer are never active at the same time, they can be executed as a unique process. The overhead of stack copying can be reduced by remarking that most of the time W_1 and W_2 already share a portion of the program search tree. In this case, only part of the stacks of W_1 need to be copied on W_2. This incremental copying optimisation is currently exploited by the MUSE system [AK90].

4 Scheduling of work

Scheduling is difficult in OPERA because of the high cost of task creation on the target multiprocessor architecture. The structure of the OPERA scheduler is hierarchical: specialized master schedulers schedule slave workers or schedulers. The schedulers of OPERA are designed to use an approximate representation of the system, to avoid costly synchronizations.

4.1 Task granularity issue

It is possible to give the conditions under which it is worthwhile to export work in an OR-parallel Prolog system, where the computation of both the exporter and the importer workers will proceed independently, without any synchronization or communication. Let T be the time necessary for a worker W_1 to complete a task. Let us assume that this task can be split in two subtasks of durations T_1 and T_2 and let E_2 be the overhead caused to W_1 by exporting the task T_2 from W_1 to W_2, and I_2 the time required by W_2 to import the task T_2. Obviously: $T = T_1 + T_2$. In order for the parallel computation of T_1 and T_2 to terminate faster than the sequential computation of T, the two following inequalities must hold:
$T > (T - T_2) + E_2$ (time necessary for W_1 to complete)
$T > I_2 + T_2$ (time necessary for W_2 to complete)
These inequalities can be simplified to:

$$E_2 < T_2$$

$$I_2 < T_1 = T - T_2$$

These constraints mean that the granularities of both the exported and remaining tasks depend on the time taken to transfer (export and import) a new task. This transfer time depends a lot on the multiprocessor architecture. It is much longer on a distributed-memory than on a shared-memory architecture. In the Supernode, DMA accesses to

the memory are interleaved with local accesses and therefore the export time overhead E_2 can be expected to be low; on the contrary the speed of the links - 1 MegaByte per second - is limited compared to the speed of a standard bus (such as VME for example) and the import time I_2 can therefore be expected to be fairly important. To limit both the export and import times, the scheduler must select, among the possible exportable tasks, the task requiring the smallest stack transfers. In addition, because of the second inequality above, the scheduler should insure that an active worker, giving work to an idle worker, keeps enough work to remain active after the initialization of the new task.

To stress the importance of the task granularity problem, early measurements indicate that, on average, it takes as long to transfer a task between two transputers of the *same* Tnode module, as to execute 200 inferences with a TWAM Prolog engine. This problem is going to be even more serious in a Meganode, since the time required to set up a connexion between two transputers of different Tnode modules is much longer.

4.2 Structure of the OPERA scheduler

Several possible structures have been considered for the OPERA scheduler:

centralized: scheduling is performed by a unique process. This is the simplest solution to implement, although the scheduler process may become a bottleneck, especially in a scalable multiprocessor including a potentially high number of processors such as the Meganode.

distributed: scheduling is distributed among the workers. This solution is more difficult to implement since there is no shared data. In addition, it generates some traffic control overhead on the communication links.

hierarchical: scheduling is implemented as a hierarchy of specialized scheduler processes. At the lower level, inside a Tnode module, this solution is equivalent to the centralized structure. In a Meganode, the schedulers of the Tnode modules are hierarchically connected, mimicing the architecture of the machine. The hierarchical structure does not introduce any bottleneck since each scheduler controls a limited number of workers or schedulers of a lower level. This structure also creates a limited overhead since the control traffic, on the control buses, remains independent of the data traffic on the links.

The hierarchical structure being best suited to the scalable distributed-memory architecture in general and to the architecture of the Supernode in particular, it has been retained for the OPERA scheduler.

4.3 Work selection strategies

4.3.1 Classification of workers and clusters

On each worker, the Spy process (see section 3.3) informs the scheduler of the activity of the worker. Three classes of workers have been defined:

idle: an idle worker does not have any Prolog task to compute and after having sent a request for work to its scheduler, it is expecting an import autorisation.

quiet: a quiet worker is active but does not have enough work to share with an idle worker without taking the risk of becoming rapidly idle. As long as it remains quiet, a quiet worker does not communicate with the rest of the system.

overloaded: an overloaded worker is active and has enough work to share with an idle worker.

The Spy estimates the workload of the active workers: workers are classified quiet or overloaded depending on whether they are below or above the threshold. If all workers are overloaded, more resources can be used. If some workers are idle, too many resources are used. Maximum efficiency in the use of the multiprocessor is reached when all the workers are quiet.

This classification is generalized to the Tnode modules of a Meganode, the threshold between quiet and overloaded Tnodes being of course different.

4.3.2 Evaluation of workload

There are no simple and exact criteria to evaluate the load of a worker. Compile time analysis of the granularity of Prolog programs is currently an active research topic [SKD90]. In OPERA, the evaluation of workload is performed dynamically. The simplest heuristic, used in the current OPERA prototype implementation, is to measure the workload by the number of choice points of the worker. This measure is maintained by the Spy and transmitted to the Scheduler. The threshold between quiet and overloaded workers is then simply a number of choice points.

4.3.3 Evaluation of the state of an OR-parallel Prolog system in a distributed-memory computer

Maintaining the exact state of a distributed-memory system amounts to the costly problem of global synchronization in a message passing architecture [Ray88]. Therefore a more pragmatic solution was devised for OPERA. The OPERA strategy is first described for a Tnode and then generalized to a Meganode.

The Spy samples the measure of the number of choice-points and filters the "noise" caused by small and unsignificant variations of the load, the frequency of which increases with the efficiency of the Prolog engine and is therefore high in OPERA. The filtered values are then transmitted to the Scheduler. The measure of the workload of a worker is sampled and filtered to limit the activity of the Spy, source of overhead for the system, and the number of messages from the Spies to the Scheduler, to avoid saturation of the Scheduler.

Because of the filtering and of the transmission delays from the Spies to the Scheduler, the global state managed by the Scheduler can only be approximate. The discrepancy between the approximate and the real state of the system may be the origin of inappropriate scheduling decisions: an overloaded worker may become quiet or idle

after it has been selected by the scheduler to give work to an idle worker. Experimental results indicate that the frequency of this case depends on the value of the threshold and remains very low for sufficient threshold values.

The previous solution can be generalized to the Meganode, where the Schedulers of the Tnode modules play the part of the Spies on the workers, reporting to the second-level Scheduler.

4.3.4 Communication protocol between Scheduler, Importer and Exporter

To cope with the inaccuracy of state information maintained by the Schedulers, a protocol has been defined between the three entities involved in the creation of a new task. This protocol can be decomposed into four major steps (see figure 2):

1. A Scheduler receives, via the control bus, a request for work from the Importer of a worker W_2 and an overload signal from the Spy of W_1.

2. The Scheduler establishes a communication link through the crossbar switch (switches in the case of a Meganode). It then sends export and import autorisations to W_1 and W_2, giving the identification of the ports to use for sending and receiving the TWAM stacks.

3. Transfer of the stacks between W_1 and W_2 takes place (see section 3.5). In case where W_1 has become quiet or idle in between, W_1 simply sends a NOWORK signal to W_2.

4. After completion of the transfer or transmission of the NOWORK signal, both the Exporter of W_1 and the Importer of W_2 release their communication link by sending their *precise load* (neither sampled nor filtered) to the controler/Scheduler.

At the end of step 4, receiving the exact load values of both W_1 and W_2 allows the Scheduler to update its approximate state and avoid consecutive inappropriate decisions. In particular, if no task could be transferred (NOWORK signal in step 3), the Scheduler classifies W_1 as quiet or idle, depending on its load, W_2 remaining idle. If a transfer took place, the state of W_1 may remain overloaded or become quiet, while the state of W_2 is changed from idle to quiet. As a consequence, the more scheduling decisions made, the more accurate the state of the system maintained by the scheduler.

4.3.5 Selection of work

Idle workers obtain from their scheduler the address of a choice point (of an active worker) containing at least one untried alternative. Two conflicting criteria are used by the scheduler to select work in the search tree:

1. maximize the benefit provided by the selected work, which is equivalent to maximizing the granularity of this work. To reach this goal, the OPERA scheduler selects the highest possible work in the tree, following the heuristic "the higher the work, the larger the granularity".

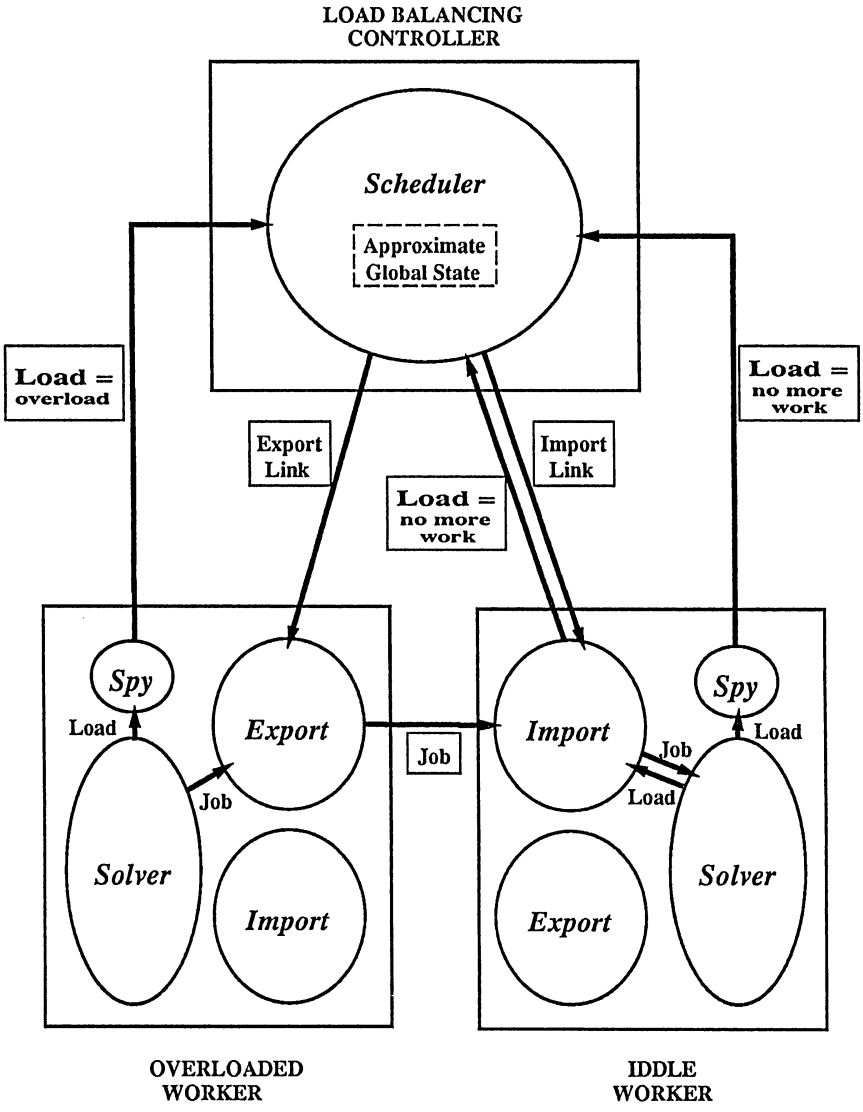


Figure 2: Software structure of OPERA

2. minimize the overhead generated by the initialization of the new task. This can be obtained by selecting, among the possible works, the work that minimizes the size of the stack segments to be copied, at the possible expense of the granularity. To further minimize the copying overhead, which mainly derives from initializations of transfers, several choice-points can be used to create a new task. This number of choice-points must be carefully chosen to leave enough work for the overloaded worker.

Finding the best possible compromise is an open problem. The current prototype selects the highest work in the search tree, the height in the search tree being measured by the value of the counter of choice-points.

5 Implementation and preliminary results

5.1 Implementation

Because of the lack of software environment available on the Supernode, much basic system work has been necessary prior to the implementation of OPERA [BFF+89]. In spite of this difficulty, special care has been taken to depart from a very efficient sequential implementation in order to provide *effective speed-ups* over efficient sequential Prolog systems. Therefore the WAM code generated by the compiler is expanded into a mixture of in-line code and runtime subroutine calls, both written in the assembly code of the transputer. In the existing prototype, cut and side-effects have not yet been included in the parallel Prolog engine.

5.2 Sequential efficiency

The TWAM is one of the most efficient existing Prolog implementations on transputer. Measured on the standard *naive-reverse* benchmark, OPERA runs at 34 KLIPS (KiLo Inferences per Second). This performance is confirmed by the results of several other classical benchmarks. Because of the specificity of the transputer architecture, it is difficult to relate the performance of OPERA with the performance of other Prolog systems running on different hardware. The transputer delivers 10 RISC Mips if all program and data are located in the internal 4 KBytes RAM. Processor address and data paths being multiplexed, the actual efficiency of the transputer is much lower for programs using external memory intensively such as Prolog systems. The transputer architecture is indeed optimized for context switching at the expense of sequential performance: few registers are available and bit field manipulations - intensively used for tag management -, are very expensive. Altogether, transputers do not execute Prolog systems better than 1 Mips CISC processors. The actual efficiency of the TWAM depends a lot on the use of the internal RAM. In the sequential version of the TWAM, it is used to store the TWAM registers and the runtime system. This optimization has not yet been included in the parallel TWAM which runs sequentially at 21 KLIPS on *naive-reverse*.

5.3 Parallel benchmarks

Several small benchmarks, mainly provided by ECRC, have been used to measure the efficiency of OPERA running in parallel. Because of the lack of "one-solution" predicate or commit in the current OPERA prototype, some programs had to be slightly modified. All the pragmas indicating parallel predicates have also been removed in order to use ordinary Prolog programs. The programs used for table 1 are:

hamilton: the problem is to find a closed path in a graph. The graph used in the benchmark includes 20 vertices and 60 edges. All solutions are computed (460,000 inferences).

map: computes all solution to a small map coloring problem (22,800 inferences).

queens1: this is a solution to the *queens* problem where the rows and columns already used in a partial solution are recorded to avoid using them again and therefore reduce the search space. All solutions (92 for eight queens) are computed (approximately 103,000 inferences for eight queens).

queens2: this is a more "natural" solution to the *queens* problem, where each possible row is tested for each column. All solutions (92 for eight queens) are computed (approximately 233,000 inferences for eight queens).

5.4 Parallel results

As is classical for parallel systems, the speed-ups of OPERA running in parallel, relative to sequential execution, depend on the size of the problem being solved. Almost linear for large size problems, they level off rapidly for the "smallest" benchmarks (see table 1). When the number of workers available implies to use too small problems for task creation, the speed start decreasing.

6 Related work

Several OR-parallel Prolog systems have already been efficiently implemented on shared or distributed-memory architectures. In most cases, scheduling is distributed among workers and uses global data.

The OPERA computational model is mainly inspired from the Kabu-Wake model [Mea86], the first system to copy stacks when a task is created and to date the variable bindings to discard invalid bindings. The scheduler is distributed and performed by otherwise idle processors. Processors having work to share are interrupted and copy their state to idle ones. The implementation, on a specialized distributed-memory architecture, is based on a rather slow interpreter and provides linear speedups for large search space programs.

The Muse model [AK90] also performs copying of the stacks from active to idle workers, but it assumes that the multiprocessor provides some global address space. In Muse, active workers do not *give* work, but *share* it with idle workers. Aurora [LBD+88]

Table 1: Execution times of benchmarks running in parallel

The first line gives the execution time, measured in milliseconds. The second line is the speed-up of parallel execution relative to the parallel system running on a single processor. The measures of OPERA are compared to Quintus Prolog release 2.4.2 running on a 1.5 MIPS SUN3/50 and on a 12.5 MIPS SUN4/60.

Program	1	2	4	8	12	16	Quintus Sun3/50	Quintus Sun4/60
ham	31271	16937	8122	4320	3260	2791	27100	6000
	1	1.85	3.85	7.24	9.59	11.2	1.15	5.2
map	1568	893	515	378			1350	300
	1	1.76	3.04	4.15			1.16	5.2
queens1-8	3890	2060	1112	682	592	540	4350	930
	1	1.89	3.50	5.70	6.57	7.20	0.89	4.2
queens1-10	90926	45628	23029	11663	8275	6433	103300	21900
	1	1.99	3.94	7.79	10.98	14.13	0.88	4.15
queens2-8	8571	4397	2380	1319	1085	933	8650	1867
	1	1.95	3.60	6.50	7.90	9.18	0.99	4.6

and PEPSys [BdKH+88] do not copy the state of active workers to idle ones but they share it. Limited work has been devoted to scheduling in the PEPSys implementation, since the cost of task installation is independent of the respective positions of the idle and active workers in the search tree. This is not the case in Aurora where four schedulers have been implemented [CS89] [BDL+88] [Bra88] [BMS90]. Compared to the scheduler of OPERA, the schedulers of Muse, Aurora and PEPSys share the caracteristics of being executed by otherwise idle workers and using an exact representation of the program search tree in shared memory.

K-LEAF [BCM+90] differs from multisequential systems in that process creation is performed "eagerly" by active workers and not "lazily" by idle ones. Scheduling is distributed, employing an idle mask message circulating along a virtual ring. Workers having sendable processes send work to idle ones, identified from the mask, before propagating the idle mask. K-Leaf is implemented on an experimental transputer-based architecture, fully connected by a Delta network, the software of which provides virtual global address space.

The closest approach to the Opera scheduler is perhaps followed by the Japanese Fifth Generation Computer Systems project where it is considered [Chi90] that "social systems provide good models for parallel processing". The hierarchical structure of the scheduler of OPERA is similar to the hierarchical structure of the management in a company.

It is difficult to compare the efficiency of OPERA and other OR-parallel Prolog systems since, because of the lack of predefined predicates in the current OPERA prototype, so-far it has only been possible to run a small number of benchmarks in parallel. Another difficulty is that the other systems mentioned above have been implemented

on different multiprocessor architectures. However, preliminary results available indicate that OPERA compares favorably to the most efficient existing OR-parallel Prolog systems.

OPERA runs sequentially approximately three times faster than the emulated version of K-LEAF[1]. OPERA seems also more efficient than Muse, Aurora and PEPSys, although it is difficult to compare one processor of a Sequent to a transputer! In parallel, the speed-ups of OPERA are lower than the speed-ups provided by the other systems. Several reasons may explain these lower results. The main reason is probably the high cost of inter-processor communication in the Supernode compared to shared-memory or fully connected multiprocessors. Another possible explanation is the high efficiency of the sequential Prolog engine of OPERA, which increases the relative overhead of task scheduling and requires larger benchmarks (the speed-ups of queens1(10) are good) to deliver good speedups. The experimental conditions are also different, since, contrary to the other systems, all programmer annotations of "parallel" predicates have been removed from the benchmarks used by OPERA, which are *plain* Prolog programs. In addition, a large number of optimizations have not yet been included in the current OPERA prototype, such as incremental copying of the stacks and giving several choice points to idle workers, to reduce the task installation overhead.

7 Conclusion

The OPERA project aims at implementing Prolog efficiently on scalable, distributed-memory architectures. Prolog is a good candidate for parallel programming because of the inherent parallelism of Prolog programs. OPERA exploits OR-parallelism, expected to require less communication than AND-parallelism and therefore more suitable to distributed-memory architectures. The computational model of OPERA is based on efficient sequential Prolog engines running almost independently on each processor. Scheduling is performed by specialized processors, structured hierarchically. Each scheduler controls several slave workers executing a Prolog engine or several slave schedulers at a lower level. To avoid costly synchronization, the schedulers use an approximate representation of the state of the system, using sampled and filtered measures performed by their slaves. In spite of the lack of software on the Supernode, an efficient prototype of OPERA has been implemented on Supernode. The efficiency of the sequential Prolog engine is close to the best existing Prolog implementations while parallel execution provides good speed-ups for large search space programs.

Future developments of OPERA include extensions and improvements of the current prototype. The basic Prolog engine and scheduler of OPERA will be extended to support more standard Prolog predefined predicates, especially cut or commit, to allow running a larger number of existing Prolog programs. Implementing side-effect predicates will change dramatically the scheduling since executing these predicates restricts the possible execution schedules. Other extensions will support the use of a large scale

[1]provided that *queens1* and *queens2* of K-LEAF are respectively *queens2* and *queens1* from ECRC and considering that the *hamilton* problem of K-LEAF, using a ten nodes graph, is much smaller than the *hamilton* problem of ECRC, using a 20 nodes graph.

Meganode, composed of interconnected Tnodes. Several improvements can also be done to increase the efficiency of the OPERA prototype. To improve the speed of the Prolog engine of OPERA, optimal use of the internal RAM should be incorporated in the parallel engine. The overhead of task creation can be decreased by using incremental copying optimisation and more sophisticated work selection strategies.

Acknowledgements

The implementation of OPERA on Supernode would not have been possible without the invaluable help of Jacques Eudes, Philippe Waille and Miguel Santana who contributed to the implementation of the software development environment of the Supernode and of Patrick Poissonnier who implemented the measurement tools.

References

[AK90] K. A. M. Ali and R. Karlsson. The Muse or-parallel prolog model and its performance. In *Proceedings of the NACLP'90*, Austin, 1990.

[BCM+90] P.G. Bosco, C. Cecchi, C. Moiso, M. Porta, and G. Sofi. Logic and functional programming on distributed memory architectures. In *Proceedings of the 6th International Conference on Logic Programming*, pages 325–339, Jerusalem, June 1990.

[BdKH+88] U. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J.C. Syre, and H. Westphal. The parallel ECRC prolog system PEPSys: An overview and evaluation results. In *Proceedings FGCS'88*, Tokyo, Nov-Dec 1988. International Conference on Fifth Generation COmputer Systems.

[BDL+88] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, and R. Stevens. Scheduling or-parallelism: An argonne perspective. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1590–1605, Seattle, August 1988.

[BFF+89] J. Briat, M. Favre, D. Fort, Y. Langué, and M. Santana. Parx: a parallel operating system for transputer-based machine. In *Proceedings 10th. Occam User Group*, 1989.

[BMS90] Anthony Beaumont, Muthuraman, and Péter Szeredi. Scheduling or-parallelism in Aurora with the bristol scheduler. Technical Report TR-90-04, University of Bristol, March 1990.

[Bra88] Per Brand. Wavefront scheduling. Internal report, SICS, Gigalips project, 1988.

[Chi90] T. Chikayama. Current status of research and development of parallel inference systems in fifth generation computer systems. Pre-conference Workshop on Parallel Logic Programming, 7^{th} International Conference on Logic Programming, ICLP'90, Eilat, june 1990.

[CLO84] Charles CLOS. A study of non blocking switching networks. In Chuan-Lin Wu and Tse-Yun Fen, editors, *Tutorial Interconnexion Networks for parallel and Distribued Processsing.* IEEE Computer Society Press, 1984. republished from The Bell System Technical Journal, March 1953 pp406-424.

[CS89] A. Calderwood and P. Szeredi. Scheduling or-parallelism in Aurora. In *Proceedings of the* 6^{th} *International Conference on Logic Programming*, Lisboa, June 1989.

[HJMWS86] J.G. Harp, C.R. Jesshope, T. Muntean, and C. Whitby-Stevens. The development and application of a low cost high performance multiprocessor machine. In *Proceedings ESPRIT'86: Results and Achievements.* Elsevier Science Publishers, 1986.

[LBD+88] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D.H.D. Warren, A. Calderwodd, P. Szeredi, S. Haridi, P. Brand, M. Carlson, A. Ciepielewski, and B. Hausman. The Aurora or-parallel prolog system. In *Proceedings FGCS'88*, Tokyo, Nov-Dec 1988. International Conference on Fifth Generation Computer Systems.

[Mea86] H. Masuzawa and et al. Kabu wake parallel inference mechanism and its evaluation. In *1986 FJCC*, pages 955–962. IEEE, November 1986.

[Ray88] Michel Raynal. *Networks and Distributed Computation. Concepts, Tools and Algorithms.* Computer Systems Series. The MIT Press, 1988. ISBN 18130-4 RAYNH.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. *acm computing surveys*, 21(3), september 1989.

[SKD90] M. Hermenegildo S. K. Debray, N-W Lin. Task granularity analysis in logic programs. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, White Plains, New York, June 20-22 1990.

[TP90] A. Touzene and B. Plateau. Mesures de performance des communications du meganode à 128 transputers. Technical report, LGI-IMAG, projet CMaP, 46, avenue Félix Viallet, 38031 Grenoble Cédex, France, 1990.

[War83] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Report tn309, SRI, October 1983.

[War87] D.H.D. Warren. The SRI model for or-parallel execution of prolog. Abstract design and implementation issues. In 4^{th} *Symposium on Logic Programming*, pages 46–53, San Fransisco, Sept. 1987.

[WRCS87] H. Westphal, P. Robert, J. Chassin, and J.-C. Syre. The PEPSys model: Combining backtracking, and- and or-parallelism. In 4^{th} *Symposium on Logic Programming*, pages 436–448, San Fransisco, Sept. 1987.

Flexible Scheduling of Or-parallelism in Aurora: The Bristol Scheduler

Anthony Beaumont, S Muthu Raman*, Péter Szeredi†
and David H D Warren
Department of Computer Science, University of Bristol,
Bristol BS8 1TR, U.K.

Abstract

Aurora is a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors, based on the SRI model of execution. It consists of a Prolog engine based on SICStus Prolog and several alternative schedulers. The task of the schedulers is to share the work available in the Prolog search tree

This paper describes the Bristol scheduler. Its distinguishing feature is that work is shared at the bottom of partially explored branches ("dispatching on bottom-most"). This can be contrasted with the earlier schedulers, which use a "dispatching on topmost" strategy. We argue that dispatching on bottom-most can lead to good performance, by reducing the overheads of scheduling.

Our approach has been to find the simplest scheduler design which could achieve performance competitive with earlier more complex schedulers. This design gives us a flexibility in deciding strategies for sharing work and allows us to examine ways of improving the performance on both non-speculative and speculative work. We note that in speculative regions the priority of some work is higher than others. We have investigated strategies which help workers to avoid low priority work.

We present the basic design of the Bristol scheduler, discussing the data structures and the main algorithms. We also present performance results for the new scheduler using a number of benchmark programs and large applications. We show that the performance of the Bristol scheduler compares favourably with other schedulers. Our work also shows that special treatment of speculative work leads to improved performance.

Keywords: Implementation, Or-parallelism, Multiprocessors, Scheduling.

1 Introduction

Aurora is a prototype or-parallel implementation of the full Prolog language for shared-memory multiprocessors, currently running on Sequent and Encore machines. It has been developed in the framework of the Gigalips project, a collaborative effort between

* *Visiting from (and present address)* National Centre for Software Technology, Gulmohar Cross Road 9, Juhu, Bombay 400 049, India

† *On leave from (and present address)* SZKI, IQSOFT, Donáti u. 35–45, Budapest, Hungary

groups at Argonne National Laboratory, University of Bristol, and the Swedish Institute of Computer Science (SICS). A full description of Aurora can be found elsewhere [8].

Aurora is based on the SRI model [13] in which or-parallel execution of Prolog programs consists of the exploration of a search tree in parallel by a number of *workers*. A worker is defined as an abstract processing agent. During execution, a tree of *nodes* is created, where each node represents a Prolog choicepoint. A worker will begin working on a *task* by taking an alternative from a node, creating a new arc of the tree. The task will be explored in the normal sequential Prolog manner, and will end when the worker runs out of work. The way workers move around the tree and communicate with each other in order to find tasks is determined by some *scheduling strategy*.

Branches of the tree are extended during resolution and destroyed on backtracking. A major problem introduced by or-parallelism is that some variables may be simultaneously bound by workers exploring different branches of the tree. The SRI model dictates that each worker will maintain a *binding array* to hold the bindings associated with its current branch. We can say that a worker is *positioned* at a node, when its binding array holds the bindings associated with the path between the root and the given node. Moving up a branch involves removing bindings, while moving down involves adding bindings to the binding array.

In Aurora the search tree is divided into *public* and *private* regions, the boundary between the two being marked by a *sentry node*. Private regions contain nodes which are explored by a single worker, and for workers to be able to share work at a node, that node has to be made public. Another distinction is that nodes can be either *parallel* or *sequential* through user declarations. Alternatives from sequential nodes can only be executed one at a time. We can also think of each node as being either *live* (i.e. having unexplored alternatives) or *dead* (no alternatives to explore). A node is called a *fork node* if it has more than one child.

An Aurora worker consists of two components: the *engine*, which is responsible for executing the Prolog code and the *scheduler*, responsible for finding work in the tree and for synchronising with other workers. There is a strict interface between these components [12] which enables independent development of different schedulers. Aurora execution is governed by the engine: whenever the engine runs out of work in its private region it will ask the scheduler to find more; a process called *task switching*. The Aurora engine is based on SICStus Prolog version 0.6 which has been extended to comply with both the SRI model and the engine/scheduler interface.

2 Scheduling Strategies

We now discuss the problem of finding a new task for a worker which has run out of work. We have already stated that work can only be taken from public nodes, therefore idle workers must find a live, parallel, public node. If we assume that initially all work is public then we could allow idle workers to search the tree to find work. This will allow work to be found from any branch but not without some cost. An idle worker may have to search a large number of nodes before work is found, and also the search will require some synchronisation to avoid searching branches as they are being reclaimed by backtracking workers.

To focus the search into areas of the tree where work may be more likely to be found we could search only those branches which are currently being extended by busy workers. Selected workers could be scanned to assess whether the branch they are working on contains work or not.

We should note however that a branch can be explored quicker by a single worker if that worker keeps the task private, rather than making some or all of it public. This indicates that if all workers are busy then there is no reason to make work public and therefore it would be better to assume that initially all work is private and that workers make work public on demand only.

Following this approach, an idle worker searching for a new task might select a worker which has private work and ask it to share some or all of it. We must remember however that searching for work only on branches which are currently being explored assumes that all live nodes have at least one busy worker positioned below them, if this assumption is not true, some nodes will become inaccessable.

Another consideration is which of the available tasks an idle worker should prefer. When earlier schedulers were designed it was thought that a worker should keep most of its work private to make its task as large as possible. Only the topmost task was made available to other workers. If the busy worker kept the topmost task public and that task was not quickly exhausted then the worker would not be interfered with as it explored the rest of the branch. This is known as *topmost dispatching.*

An alternative strategy investigated in this paper is *bottom-most dispatching* when work is shared at the bottom of partially explored branches and we will discuss this later.

2.1 Topmost dispatching schedulers for Aurora

There were three earlier schedulers for Aurora, all using topmost dispatching.

The Argonne scheduler [4] uses local information that is maintained in each node to indicate whether there is work available below the node. Workers search the tree, using this local information to migrate towards regions of the tree where work is available. The workers always take the topmost task from a branch since this is always the first task found as they move down. A bitmap in each node indicates which workers are positioned at or below the node and workers are required to update these bitmaps as they move around the tree. Information in the bitmaps can be used to locate other workers, for example in the case of pruning a subtree it is necessary to inform the workers which are positioned in that subtree that they have been pruned.

The Manchester scheduler [5] tries to match idle workers with the nearest available outstanding task, where "nearness" is measured by the number of bindings to be updated between the worker's current position and the available work. Minimising the distance between worker and task means that the worker will not consider any task below the topmost one on each branch. Again bitmaps are employed to mark the presence of workers on a branch. The Manchester scheduler uses them both for matching idle workers to available work and for locating workers during pruning.

The Wavefront scheduler [3] employs a data structure known as the wavefront which links all the topmost live nodes together. Workers find work by traversing the wavefront. As nodes are exhausted the wavefront is extended to allow access to the next live parallel node.

Topmost dispatching, used by all of these schedulers, has the disadvantage that unless the topmost task is large it will be quickly exhausted and the worker will have to repeat its search for work. This leads to relatively high task switching costs for fine granularity programs and also slows down the busy workers since they have to spend more time maintaining a live public node at the top of their branch.

2.2 The Muse Scheduler

Another approach to the or-parallel implementation of Prolog is the Muse system [1][2] which is based on having several sequential Prolog engines, each with local address space and some shared memory space. Workers in Muse copy each other's state rather than sharing it as is the case in Aurora. Potentially increasing the overheads involved in task switching. Therefore Muse requires a way of reducing the frequency of task switches involving copying.

The Muse scheduler uses bottom-most dispatching, so that a busy worker, when interrupted for work, will share all nodes on its branch. This allows an idle worker to begin work at the bottom-most of these nodes. The advantage of this strategy is that once the work at the bottom-most node is exhausted, more work can be found by simply backtracking to the next live parallel node, further up the branch. Backtracking to a public node is more expensive than backtracking to a private one, however these *minor* task switches are much less expensive than the *major* task switches which require a wider search for work. It has been found that bottom-most dispatching can reduce scheduling overheads by increasing the number of minor task switches and reducing the number of major task switches.

To help an idle worker decide which busy worker to interrupt for work, Muse introduces the concept of *richness*. Each branch of the tree has an associated richness, which is an estimate of the amount of work on that branch. In the muse system, richness is based on the number of unexplored alternatives on the branch. An idle worker will choose a busy worker from the subtree below the idle worker's current node, the choice of worker depends on the richness of each busy worker's branch. The idle worker will interrupt the worker which is working on the richest branch, ie. has the most work to share, which will further help in increasing the ratio of minor to major task switches..

3 Principles of the Bristol scheduler

In designing the Bristol scheduler we took into consideration the results from performance analyses of earlier schedulers and used this information to try and incorporate the best features of other schedulers into our design.

A performance analysis of the Manchester scheduler [11] indicated that the migration of workers to new tasks was not a significant overhead and that much more important was the administrative overhead associated with task switching, estimated to be equivalent to 4-7 Prolog calls per task. The conclusions of this analysis are that simplifying the scheduling algorithm and tuning the scheduler could reduce the costs of task switching and, more importantly, minimise the number of major task switches. Keeping this in mind we have tried to keep the design of the Bristol scheduler as simple as possible.

One of the requirements of the Bristol scheduler is that it should be flexible enough for us to try different scheduling strategies and this will allow us to compare bottom-most and topmost dispatching using the same scheduler. However, based on the good results of the Muse scheduler, we decided to use bottom-most dispatching as the default strategy. The key overhead in earlier Aurora schedulers is the major task switch. If bottom-most dispatching reduces the number of major task switches, and if minor task switches are not very expensive then the total scheduling overheads will be reduced.

A second reason for using the bottom-most dispatching strategy is its suitability for scheduling *speculative* work. This is illustrated by the following program:

```
p:- condition, !, pred1.
p:- pred2.
```

All work in the second clause is said to be speculative because if *condition* succeeds then the second branch will be pruned away. Intuitively, it would seem better to direct workers to help in evaluating *condition*, rather than *pred2*. Similarly, one would want to give higher priority to work which is further to the left within *condition* [6]. Therefore, in a speculative subtree the deepest work on the left-hand branch is the least speculative and should have the highest priority.

Earlier schedulers could not handle speculative work at all effectively. Our aim is to implement an effective speculative scheduling technique within the Bristol scheduler. The bottom-most dispatching strategy helps in directing workers to deeper regions of the search tree but this is not sufficient on its own as a scheduling strategy since the deeper branches may not be the least speculative. What we require is some way of concentrating workers in the leftmost region of a speculative subtree.

This suggests that rather than rely on taking work from busy workers, idle workers would need to scan a speculative subtree to find the least speculative available work. Our design allows us to experiment with such a strategy.

A problem with bottom-most dispatching is that it increases the size of the public region of the tree and backtracking through this region (*public backtracking*) is more expensive than private backtracking. We will try to analyse the effect of this problem by comparing bottom-most and topmost dispatching strategies using the Bristol scheduler.

4 Implementation of the Bristol scheduler

During this section we will discuss some of the issues involved in the implementation of the Bristol scheduler.

4.1 Data structures

We include the notion of richness introduced by the Muse scheduler and use an estimate of the number of live nodes on a branch as the richness of each branch. Actually, each node is given a richness which is an estimate of how many live nodes there are above it.

Primarily a worker wants to know if another worker has work available and must be able to send a message to it, for example, to ask for work. In our implementation, each

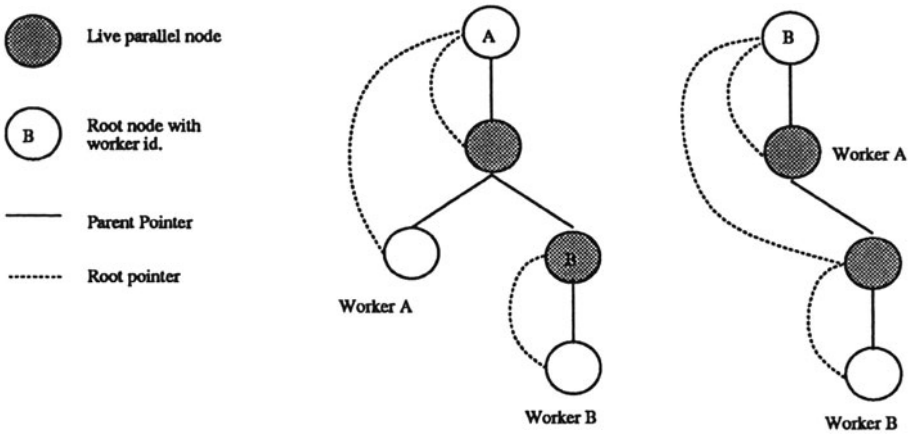


Figure 1: WORKER A BACKTRACKING TO A FORK NODE

worker has a message area, enabling other workers to send messages to it, and a record of the richness of its current branch which can be read by other workers.

To give some indication of a worker's position in the tree, each worker has an associated *root* node, which is defined to be the root of the subtree in which that worker is leftmost. Initially, this is equal to the workers' sentry node when it first starts work. The identifier of the worker is stored in its root node, and that worker will be known as the *owner* of the node. All nodes subsequently created will contain a pointer (so called *root pointer*) to that workers' root node.

A worker's root may change due to the actions of other workers in backtracking. When a worker backtracks to a fork node from the first child, leaving another worker below, then the backtracking worker must move the root of the remaining worker up the tree to reflect the change. This is illustrated in Figure 1 where worker A backtracks out of the left subtree, leaving worker B as the new leftmost worker (the letters in the root nodes show the identity of the worker they belong to). In this case, worker A will put B's identity into its old root and make that node the new root of worker B. B's old root has its root pointer set to point to the new root.

In a speculative region we will want to find the leftmost (least speculative) task, and therefore will need some way of searching a subtree from left to right. By following the root pointers to a root node and finding its owner, we have a way of finding the leftmost worker in any subtree, and we can also tell if that worker has work available. To continue searching for work, a worker requires some way of finding the bottom of the leftmost branch and moving right. Each worker maintains a pointer to its sentry node which marks the leaf of the public branch. After identifying the owner of a root node and if that worker has no private work then the worker's sentry node pointer is used as a way of gaining access to the bottom of that worker's branch. The owners identity in a root node acts as a *leaf pointer*.

In order to simplify the further search for work the notion of *right pointer* is introduced. This is illustrated in Figure 2 where we can see how the tree is organised. The right pointer points to the next sibling, if there is one. For the rightmost child of a live node the right pointer will point to itself, indicating the potential work present there. For the rightmost

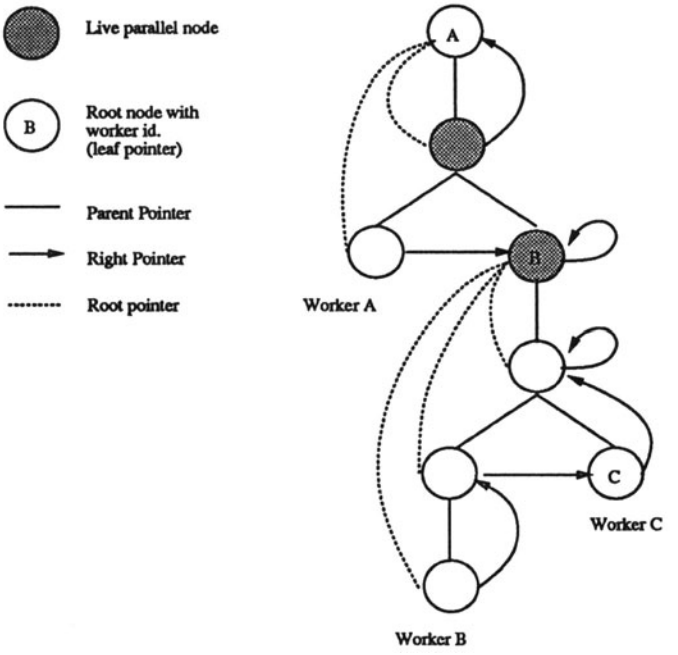


Figure 2: ORGANISATION OF THE TREE

child of a dead node the right pointer will point upwards to the first ancestor which has either a right sibling or a live parent.

We use a flag to indicate that the right pointer of a node points to a right sibling and not to some other node. Using the root, leaf and right pointers a worker can search around the tree from left to right. This method of linking the nodes of a tree to allow left to right traversal was taken from the data structure used in Andorra-1 for maintaining the goal list [9].

4.2 Looking for work

The engine will hand over control to the scheduler when the worker backtracks to its sentry node and the scheduler will be responsible for finding work in the public region of the tree. We are exploring two different strategies which the scheduler will use to find work, the *richest worker* strategy and the *left-to-right search* strategy.

4.2.1 The richest worker strategy

Following the richest worker strategy, the scheduler will attempt to find work in two ways; Firstly it will backtrack through the nodes above the worker's current position to see if any work can be found nearby. If a live parallel node can be found then the scheduler takes an alternative from it and returns control back to the engine to restart work. If no work can be found then the other workers will be scanned to see if they have work

available. The idle worker will identify the worker which is working on the richest branch and interrupt it for work. That worker will then make all of its private nodes public and the idle worker can begin work at the bottom of that branch. Note that the number of nodes being made public is flexible in that we could limit this number and therefore bottom most dispatching need not necessarily be used.

This strategy has the advantage that work is made public only on demand, so if a worker is not interrupted for work it will not make any nodes public. Workers may have to migrate further when taking work from another worker. We hope that the bottom-most dispatching strategy will minimise the number of major task switches, and most work will be found quickly from a node just above the workers current position.

4.2.2 Left to right search

We want to get some indication of how much can be gained by treating speculative work differently from non-speculative work and have explored an alternative strategy, which we call left-to-right search. The assumption behind this strategy is that the whole tree is speculative and that work on the leftmost branch has the highest priority, and the priority of work decreases the further away it is from the leftmost branch. This assumption clearly represents too narrow a view of speculative work; in general all work is non-speculative unless it is in the scope of a pruning operator. `Setof` (and `bagof`) create subtrees which are locally non-speculative even if the setof itself is speculative, ie all branches should have equal priority. We describe a more refined strategy for handling speculative work in Section 6.

Related work by Sindaha [10] uses a similar method to search for work in a speculative subtree. However the left to right search is implemented by explicitly linking the sentry nodes of all branches to create a data structure similar to the wavefront. Workers find work by traversing this data structure. This work is not as far advanced as the Bristol scheduler but we hope later to compare this approach with our own.

The left to right search strategy also uses bottom-most dispatching. Finding work will again begin by backtracking through the nodes immediately above the worker's current position to find nearby work. If no work can be found the worker will search the tree from left to right, by following the root, leaf and right pointers. The leftmost worker can be identified as the owner of the root of the tree, the scheduler will search right from the leftmost worker until it finds either a worker with private work which it can share, or a live parallel node.

4.3 Side-effects and suspension

A goal of the Aurora system is to implement all standard Prolog built-in predicates, preserving their sequential semantics. To achieve this, we delay the execution of a call to a side-effect predicate until it becomes leftmost in the whole tree. We implement this delay by allowing workers to suspend a branch when executing a side-effect predicate. There are some special predicates, for example those used in the implementation of `setof`, where it is sufficient to ensure that the branch is leftmost within some subtree. Checking whether a worker is leftmost within some subtree is done simply by testing if the worker's root node is at or above the root of the subtree. Asynchronous versions of these side-effect

predicates are also provided and these suspend only if they occur within the scope of a cut and may be pruned.

To suspend a branch, all a worker must do is to mark its root node as suspended and make the root of the suspended node point down to the sentry node on the suspended branch. The scheduler will then find a new task for the suspending worker to begin.

We next look at how suspended branches can be restarted. If a worker backtracks out of, and reclaims, the leftmost child of a node, it will check to see if there is a suspended right sibling of that node. If there is, it will delete the suspended flag and proceed to check if the branch has now become leftmost in the subtree in which it was suspended. If this is the case the worker will restart the suspended branch. If the branch cannot be restarted the worker will set the suspended flag in its own root and carry on looking for work. The suspended branch will wait until some other worker notices the new suspended node while backtracking.

4.4 Cut and commit

Aurora supports two pruning operators: the conventional Prolog *cut*, which prunes all branches to its right and a symmetric version of cut called *commit*, which prunes branches both to its left and right. To preserve the sequential semantics, a pruning operation will not go ahead if there is a chance of it being pruned itself by a cut with a smaller scope. It may be possible to improve on this and we are investigating the method which has been implemented in Muse where the worker will not suspend the branch but partially do the pruning and leave the rest to be done as and when it ceases to be endangered by the cut.

A pruning operation should suspend only if a branch to its left leads to a cut of smaller scope. But, to determine whether a particular pruning operation should suspend or not we need some information about the presence and scope of cuts in the tree.

The engine-scheduler interface [12] makes the necessary information available to the scheduler and we use it to implement the Bristol scheduler's pruning operators in the following way.

The scheduler decorates the tree with information about the presence and scope of cuts. When a node is created which has parallel alternatives containing cut, then that node is marked as a cut boundary node. Each node contains a cut counter, which indicates the number of cuts in the worker's continuation when the node was created. The worker also keeps a cut counter and this is incremented when a clause containing cuts is entered and decremented whenever a cut is executed. When executing a cut, a worker will check if any of the nodes below the cut boundary have children to the left and are either marked as a cut boundary or have a cut counter greater than the workers current cut counter. If such a left sibling is found then the cut will be suspended. For a more detailed description of decorating the tree with cut scope information the reader is referred elsewhere [6].

We will now look at how pruning is implemented in the Bristol scheduler. A pruning worker will visit each node below the boundary node of the cut (or commit), first removing all unexplored alternatives at the node's parent, and then pruning all the right siblings of the node. If the pruning operation is commit then left siblings must be pruned too. All siblings except the leftmost will be root nodes.

To prune a sibling, the worker will mark the node as pruned and try to identify the worker which will take responsibility for clearing the pruned subtree away. Unless the

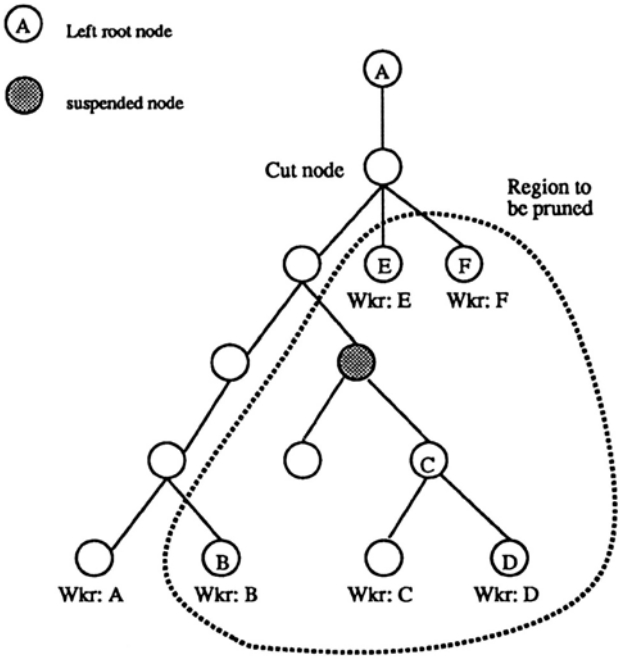


Figure 3: WORKER A ABOUT TO PERFORM A CUT

sibling node is suspended, this means the worker will interrupt the owner of the node. That worker will then pass on the interrupt to any other workers in the subtree. To prune the leftmost sibling in the case of commit, then the worker must follow the node's root pointer to find its root. If its root is not marked suspended then the owner of the root will be interrupted with the information that its subtree is pruned.

If any of the siblings are suspended, it is not possible to identify workers which are in the subtree below that node. These subtrees must be searched to inform any workers which are working there that they have been pruned, although all the pruning worker will do is to marked them as unsearched. The search will be carried out by one of the pruned workers as it moves out of its own pruned subtree. The pruning worker will only search pruned subtrees if no other worker can be found to do the job for it. In this case it will search until it finds the first worker to be pruned and then that worker will take over any further searching.

Figure 3 shows an example where worker A wants to perform a cut up to the cut boundary and there are five other workers in the region which is to be pruned. Worker A will be able to identify and interrupt workers B, E and F. Worker B will search the subtree whose root was suspended at the time worker A did the pruning and will interrupt worker C. That worker will in turn interrupt worker D. The interrupted workers will then backtrack out of the pruned region and look for work elsewhere.

5 Performance results

To assess the performance of various aspects of the Bristol scheduler, we have used a number of benchmarks and application programs, descriptions of which can be found elsewhere [11][7]. All the programs were run on a Sequent Symmetry using the Foxtrot version of Aurora.

To discover which dispatching strategy should be preferred, we have run the Bristol scheduler on a number of programs using 10 workers under both topmost and bottom-most dispatching. We obtained figures on the frequency and duration of task switches in each program. Task switching begins when a worker backtracks to its sentry node and ends when a new task is found and control passes back to the engine. A distinction is drawn between minor task switches, which end when work is discovered on the same branch, and major task switches, when the worker finds that no work is left on its current branch and it must find work from another part of the tree. Bottom-most dispatching should reduce the number of expensive major task switches, while increasing the number of minor task switches. Table 1 shows for each program the average number of task switches made by each worker, and also gives the average duration of each task switch in microseconds.

	Number				Average duration			
	Bottom-most dispatching		Topmost dispatching		Bottom-most dispatching		Topmost dispatching	
Program	Major	Minor	Major	Minor	Major	Minor	Major	Minor
parse1	7.7	12.5	8.5	11.5	2315	310	2037	232
parse2	11.0	26.0	14.2	19.7	1554	219	1377	187
parse3	7.2	12.0	7.7	10.5	2431	209	2342	202
parse4	12.2	47.2	34.7	46.5	1929	229	1560	212
parse5	13.0	87.2	58.7	75.0	2114	213	1707	187
db4	4.5	21.5	5	14.7	1808	196	1386	175
db5	5.0	25.7	6.5	17.7	1465	191	1351	170
farmer	3.7	4.7	4.0	4.2	3298	192	3596	180
house	4.2	13.2	4.0	3.2	1756	255	1873	184
8queens1	1.5	9.25	10.2	10.2	2623	247	1293	204
8queens2	1.7	18.7	16.0	15.5	2674	217	1096	165
tina	21.5	92.7	17.7	52.7	3925	208	4072	168
saltmustard	1.0	3.7	1.0	1.5	2256	231	2276	190
protein	12.0	163.0	74.0	232.0	2804	103	1264	97
warplan	12.7	60.0	30.5	37.2	5347	247	2864	209

Table 1: TABLE SHOWING THE AVERAGE NUMBER OF TASK SWITCHES MADE BY EACH WORKER AND THE AVERAGE DURATION IN MICROSECONDS, OF EACH TASK SWITCH

We find that the number of major task switches is significantly reduced while the number of minor task switches is slightly increased. Also the average duration of both

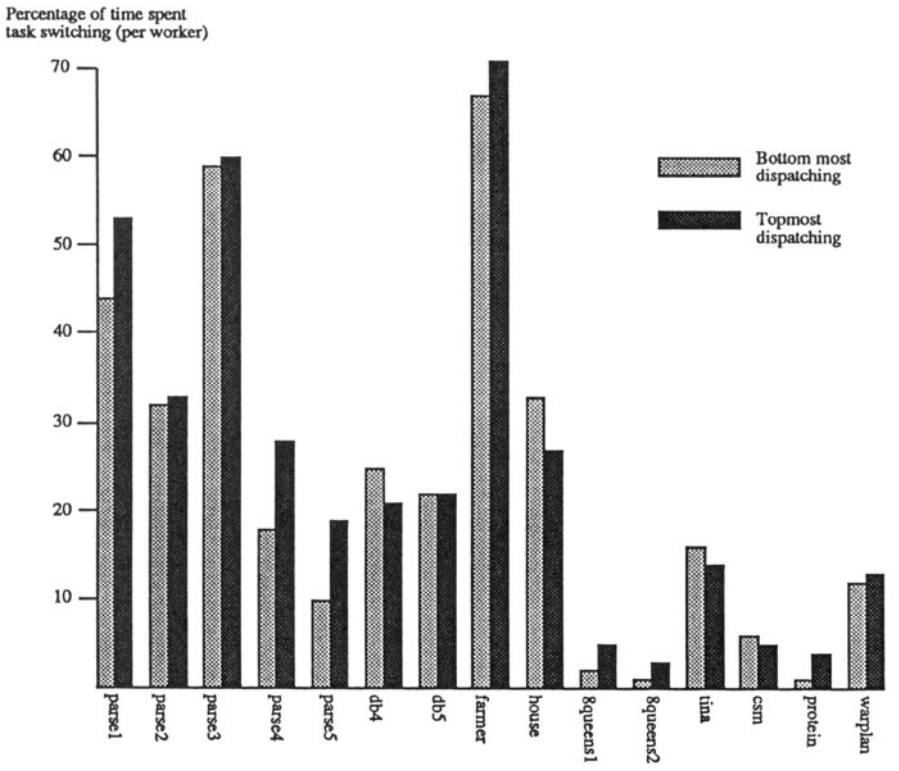


Figure 4: GRAPH OF THE PERCENTAGE OF TOTAL TIME EACH WORKER SPENDS IN TASK SWITCHING

major and minor task switches tends to increase somewhat. This reflects the increased size of the public region of the tree, under bottom-most dispatching there will be more public backtracking to be done before the worker eventually finds a new task.

To get a clearer picture of how much time was spent in task switching we computed the percentage of total time, which on average, each worker spent in task switching. This information is shown in Figure 4. We can see from this graph that bottom-most dispatching leads to less time spent task switching in all but five of our programs. From this we can conclude that in general it will be better to employ bottom-most dispatching, although there will be some occasions when we will lose out. It may eventually be possible for the scheduler to recognise which dispatching strategy should be used but currently we do not have enough information to implement this.

We next compare the Bristol scheduler with the Manchester scheduler, a topmost dispatching scheduler which is the most developed of the existing Aurora schedulers. The results, given in Table 2, reflect fairly closely what we would expect from the comparison of topmost and bottom-most dispatching in the Bristol scheduler, although the Manchester scheduler performs better on a couple of the benchmarks where we might have predicted equal or better performance by the Bristol scheduler. This may be partly due to the

Manchester scheduler's strategy of matching idle workers to the nearest available work whereas, with the Bristol scheduler, this does not happen.

	Bristol Scheduler		Manchester Scheduler	
Goals [*Times]	1wkr	10wkrs	1wkr	10wkrs
parse1 *20	1.95	0.74(2.62)	1.94	0.81(2.38)
parse2 *20	7.37	1.50(4.90)	7.35	1.69(4.34)
parse3 *20	1.68	0.67(2.49)	1.66	0.72(2.29)
parse4 *5	6.85	1.04(6.57)	6.80	1.24(5.47)
parse5	4.96	0.64(7.78)	4.79	0.85(5.61)
db4 *10	3.11	0.44(7.06)	3.06	0.41(7.53)
db5 *10	3.79	0.52(7.30)	3.72	0.51(7.34)
farmer*100	3.77	1.95(1.93)	3.80	2.26(1.68)
house*20	5.55	0.94(5.88)	5.17	0.85(6.05)
8-queens1	8.48	0.88(9.65)	8.28	0.83(9.93)
8-queens2	21.89	2.21(9.91)	21.66	2.16(10.0)
tina	19.72	2.14(9.23)	18.77	1.99(9.41)
sm2 *10	11.65	1.32(8.82)	11.27	1.23(9.19)
protein	28.56	3.01(9.49)	27.66	2.94(9.41)
warplan (blocks)	2.73	0.33(8.27)	2.50	0.38(6.58)
warplan (strips)	42.44	4.62(9.19)	40.40	4.46(9.06)
AVERAGE		(6.94)		(6.64)

Table 2: TABLE COMPARING AURORA UNDER THE BRISTOL AND MANCHESTER SCHEDULERS (RUNTIMES IN SECONDS WITH SPEEDUPS IN BRACKETS)

To obtain a comparison of two similar bottom-most dispatching schedulers, we have compared Aurora under the Bristol scheduler with Muse 0.6 (Figure 3). This version of Muse supports some form of delayed release which is not yet supported by the Bristol scheduler. Also, the muse system has been compiled using the GNU C compiler, which allows use of inline declarations, while the Bristol scheduler was not.

Generally the Bristol scheduler produces better speedups than the Muse scheduler, but this is generally somewhat outweighed by the faster engine performance of Muse. Muse's faster engine performance is due to the overhead of the SRI model in Aurora which adds about 25% to the single worker runtime. This is much greater than the corresponding overhead in Muse which is around 5%.

The next part of our performance analysis looks at our strategies for handling speculative work. The left to right search strategy is used here to find out what the benefits might be of treating speculative work differently and what the overheads are. We will later propose a better strategy for adapting the Bristol scheduler for speculative work. We first look at the overheads of the left to right search strategy in finding all solutions. Table 4 shows that the 10 worker performance is degraded by about 11%, and this is due both to the extra synchronisation required in searching for work in this manner, and also

	Aurora		Muse	
Goals [*Times]	1wkr	10wkrs	1wkr	10wkrs
parse1 *20	1.95	0.74(2.62)	1.58	0.58(2.72)
parse2 *20	7.37	1.50(4.90)	5.89	1.19(5.03)
parse3 *20	1.68	0.67(2.49)	1.36	0.60(2.27)
parse4 *5	6.85	1.04(6.57)	5.53	0.82(6.74)
parse5	4.96	0.64(7.78)	3.91	0.51(7.67)
db4 *10	3.11	0.44(7.06)	2.38	0.35(6.80)
db5 *10	3.79	0.52(7.30)	2.91	0.42(6.93)
farmer*100	3.77	1.95(1.93)	3.12	1.90(1.64)
house*20	5.55	0.94(5.88)	4.35	0.89(4.89)
8-queens1	8.48	0.88(9.65)	6.64	0.70(9.49)
8-queens2	21.89	2.21(9.91)	17.14	1.77(9.68)
tina	19.72	2.14(9.23)	14.79	1.66(8.91)
AVERAGE		(6.28)		(6.06)

Table 3: AURORA UNDER THE BRISTOL SCHEDULER, COMPARED WITH MUSE (RUN-TIMES IN SECONDS WITH SPEEDUPS IN BRACKETS)

due to the fact that taking the leftmost available work may not give the worker access to as many live nodes as the deepest available work would have done. Therefore the number of major task switches is increased.

We now take a number of application programs and look at the speedups obtained when finding the first (leftmost) solution, comparing the Manchester scheduler with three versions of the Bristol scheduler; richest worker, an improved version of richest worker where idle workers try to find work from the leftmost worker before trying the richest worker and the left-to-right search. The results are shown in Table 5. Since speculative computation always gives some variation in runtimes, depending on how well the workers were utilised during the computation, we present the speedups a ranges of best–worst. The results are given in Table 5, and show that when the computation involves speculative work, the bottom-most dispatching strategies all perform better than the Manchester scheduler. The improved richest worker strategy gives some improvement over the original richest worker strategy but the left-to-right search gives the best performance.

6 A strategy for scheduling speculative work

In order to improve on our relatively crude left to right search strategy for speculative work, we have designed a better strategy which will be implemented in a future version of the Bristol scheduler.

For this more general strategy, we no longer treat the whole tree as being speculative but allow for intermingling of speculative and non-speculative subtrees. We will want to distribute workers evenly among speculative subtrees so as not to focus too many

		Richest	Left to
		worker	right search
Goals [*Times]	1wkr	10wkrs	10wkrs
parse1 *20	1	2.62	2.44
parse2 *20	1	4.90	4.14
parse3 *20	1	2.49	2.24
parse4 *5	1	6.57	5.71
parse5	1	7.78	6.79
db4 *10	1	7.06	6.78
db5 *10	1	7.30	6.65
farmer*100	1	1.93	1.68
house*20	1	5.88	4.44
8-queens1	1	9.65	8.83
8-queens2	1	9.91	9.43
tina	1	9.23	7.89
sm2 *10	1	8.82	7.77
AVERAGE		6.47	5.75

Table 4: SPEEDUPS FOR DIFFERENT SCHEDULING STRATEGIES (BRISTOL SCHEDULER)

	Aurora	Scheduling strategy			
	one	Manchester	richest	leftmost	left to
Application	worker	scheduler	Worker	then richest	right search
Protein	1	2.90–2.65	2.30–1.96	3.28–2.97	4.46–4.36
Puzzle	1	1.13–1.09	1.33–1.25	2.64–1.77	6.10–5.06
Warplan	1	1.15–1.08	1.11–1.06	1.12–1.10	1.57–1.36
16Queens	1	1.05–1.05	3.35–2.31	3.38–3.30	6.40–3.78
triangle	1	6.44–6.00	7.06–6.60	7.20–6.60	7.68–7.34
tina	1	4.56–4.41	4.56–4.22	4.70–4.48	4.86–4.63
Average	1	2.87–2.71	3.28–2.90	3.72–3.37	5.18–4.42

Table 5: SPEEDUPS (BEST–WORST) WITH 10 WORKERS FINDING THE FIRST SOLUTION

resources into one particular subtree. We will also want workers to be able to reassess how speculative their current branch is. For example, a task which was the least speculative of a particular subtree at one moment may later become the most speculative if branches appear to its left. Workers should be able to suspend such a task in favour of a task on one of the higher priority branches to its left, an operation known as *voluntary suspension*. We believe that voluntary suspension is crucial for effective handling of speculative work.

Since it is very difficult to compare the speculativeness of two tasks in separate speculative subtrees we will not allow workers to move from one speculative subtree to another.

Our strategy can then be summarised as follows:

- First try to obtain non-speculative work.
- If only speculative work exists then find work from the speculative subtree containing the smallest number of workers.
- Always search speculative subtrees from left to right.
- Allow workers to periodically consider voluntary suspension of speculative work, in order to find less speculative work.

7 Conclusions

We presented a simple, flexible scheduler based on the principle of "dispatching on bottom-most". We have described the algorithms for finding work, public backtracking, pruning and suspension. The current implementation supports the full Prolog language.

We have presented figures to show that bottom-most dispatching generally produces better performance in Aurora than topmost dispatching, since it decreases the duration of time workers spend in task switching.

The results from running benchmark and application programs show that it is possible to get very good speedups for non-speculative computation from the Bristol scheduler using the richest worker strategy. Comparing that version of the Bristol scheduler with the Manchester scheduler we note that the Bristol scheduler's bottom-most dispatching strategy pays off on the parsing examples where the search trees are deep and narrow. The Manchester scheduler performs better on those examples where the search tree is shallow and broad.

Speedups from the Bristol scheduler are generally better then those obtained from the Muse system, although that system obtains somewhat better overall speed because of the lower overhead involved in adapting Sicstus Prolog to the Muse model.

When working on programs with large amounts of speculative work we can benefit by employing a strategy which prefers to schedule work on left of the speculative region. We can conclude that even though there is an overhead associated with using the left to right search strategy, which is due to the need for synchronisation during the search, we can benefit by using it to schedule work from regions where work on the left side is of higher priority.

We have described a general strategy for handling speculative work, which, based on the results presented here, we believe will give improved performance on speculative work. Our future work will center on implementing this strategy and analysing its performance.

8 Acknowledgements

The Authors are indebted to other members of the Gigalips project for careful reading and invaluable comments on this paper, to Mats Carlsson for his work on The Aurora engine and interface, to Bogdan Hausmann for his work on speculative scheduling, and to Khayri Ali and Roland Karlsson for their comments and for providing the benchmark timings from Muse.

This work was supported by ESPRIT projects 2471 ("PEPMA") and 2025 ("EDS"). S Muthu Raman was supported by a UN Development Programme Fellowship.

References

[1] Khayri Ali. *Or-parallel execution of Prolog on BC-Machine.* SICS Research Report, Swedish Institute of Computer Science, 1987.

[2] Khayri A. M. Ali and Roland Karlsson. The Muse or-parallel Prolog model and its performance. In *Proceedings of the North American Conference on Logic Programming*, MIT Press, October 1990.

[3] Per Brand. Wavefront scheduling. 1988. Internal Report, Gigalips Project.

[4] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens. Scheduling OR-parallelism: an Argonne perspective. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1590–1605, MIT Press, August 1988.

[5] Alan Calderwood and Péter Szeredi. Scheduling or-parallelism in Aurora – the Manchester scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435, MIT Press, June 1989.

[6] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG.* PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[7] Feliks Kluźniak. *Developing Applications for Aurora.* Technical Report TR-90-17, University of Bristol, Computer Science Department, August 1990.

[8] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[9] V. Santos Costa, D. H. D. Warren, and R. Yang. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, 1991.

[10] Raed Sindaha. Scheduling speculative work in the Aurora or-parallel Prolog system. March 1990. Internal Report, Gigalips Project, University of Bristol.

[11] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732, MIT Press, October 1989.

[12] Péter Szeredi, Mats Carlsson, and Rong Yang. Interfacing engines and schedulers in or-parallel prolog systems. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, Springer Verlag, June 1991.

[13] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

Virtual Memory Support for OR-Parallel Logic Programming Systems

André Véron, Jiyang Xu,
S. A. Delgado-Rannauro, K. Schuerman
Distributed and Parallel Systems Group
European Computer-Industry Research Center
Arabellastr. 17
D-8000 München 81, FRG
email : elipsys@ecrc.de

Abstract

Most previous parallel logic programming systems have been built on top of classical operating systems. The advances in the area of parallel operating systems have made it possible to explore new execution models that take advantage of their features. In this paper we propose a family of execution models (VM,VMBA,VMHW) that make use of the new virtual memory technologies such as copy-on-write, memory inheritance, and distributed shared memory. Preliminary results from our implementations and simulations are reported.

1 Introduction

There has been a great deal of research in the field of parallel logic programming in the past few years, and many systems have been developed ([19, 5, 3, 2, 4, 16]) and interesting results reported. The ElipSys (ECRC Logic Inference Parallel System) project at ECRC, a component of ESPRIT II Project 2025 EDS ([20, 14]) is integrating mechanisms for exploiting OR-parallelism, solving constraint satisfaction problems and supporting a tightly coupled KBMS. The research reported here has been carried out within the ElipSys project.

Most previous research on parallel logic systems has been based on the currently standard Unix[1] like environments, probably with extensions to deal with multiprocessors. The recent advances in parallel operating systems (OS), especially in virtual memory systems such as *copy-on-write* techniques, lazy memory allocation and sparse memory maps, have motivated us to investigate new parallel execution models that take full advantage of these new concepts. In addition, we also make proposals for general purposes new operating systems paradigms which would support cleanly our execution models.

[1] Unix is a registered trade mark of AT&T.

In this paper we present several execution models that make use of the new virtual memory technologies and report the preliminary results from our implementation and simulations. In Section 2 we review a few existing parallel execution models, in order to provide adequate background for our own proposals. Possible virtual memory support and the virtual memory execution model are discussed in Section 3, and a series of other execution models based on the virtual memory concepts are presented in Section 4. Results from our simulation and implementation are reported in Section 5.

2 Review of Execution Models

The processing of a query in a Prolog-like system amounts to traverse a directed tree exploring the paths from the root down to the leaves. The nodes in the tree which have more than one outcoming arcs are called branching nodes. As the processing starts from the root and flows down along one path and reaches nodes, variables are created. A newly created variable is not immediately given a value. When it is created, it is added to an environment of variables which goes along with the processing. Traversing one node can give a value to a variable which has been allocated when traversing a node upper in the tree. Giving a value to such a variable is called *binding* the variable.

An AND-parallel system is able to traverse concurrently the nodes along one path. An OR-parallel system is able to traverse different paths at the same time. This paper deals with problems specific to the design of OR-parallel systems.

Branching nodes of the tree are called OR-nodes. The tree to be traversed is called an OR-tree. Two paths in the tree have the same beginning sequence of shared nodes upto their first common OR-node. If those two paths are to be processed in parallel, it is taken as an assumption that the shared nodes must be processed only once. The parallel execution only begins when the last node of the shared sequence has been traversed or equivalently when the first common OR-node has been traversed.

The problem then is that a variable in the environment built so far can be bound independently by the traversing processing of both branches. Hence a single memory location is not sufficient enough to implement the variable. Solutions to the multiple value representation problem are called *binding schemes* and are one major issue for OR-parallel execution models of logic systems.

Starting from a naïve execution model, we briefly present in the following a few well-known techniques and binding schemes for designing OR-parallel systems.

2.1 Sequential Execution Model (WAM)

Prolog is the most well-known logic programming language on sequential machines, and a standard basis of Prolog implementations is the WAM ([23]).

The WAM is a thoughtful abstract machine and includes many optimizations which are not detailed here. It is a stack based abstract machine. Variables and terms are allocated in two stacks, the *global* and *local* stack that we will call *term* stacks from now on. Variables are updated as the execution flow proceeds along one path. The machine avoids the retraversing of shared sequences of nodes when processing two different paths:

An additional stack called the *trail stack*, is used to record all modifications done to variables created after the last traversed OR-node: After a path is explored, all bindings made since the last traversed OR-node are undone (i.e. *backtracked*) and the next branch of the last reached OR-node is processed.

2.2 Parallel execution models: The Naïve approach

A naïve but direct way to implement the computation model is to fork subprocesses whenever an OR-node is reached. The tree of processes reflects directly the OR-tree. Each subprocess receives a copy of the parent's environment and works independently. Since the variables are physically duplicated in each subprocess, the multiple value problem is naturally solved.

Although it is very simple, theoretically neat, and intrinsically parallel, this model is not realistic due to its inefficiency. The very fine granularity of computation implies intensive forking operations, which may take far more time than normal computations in classical operating systems. This kind of overhead is not offset by parallelism.

2.3 Parallel execution models: User level processes

The naïve execution model maps branches in the computation tree directly onto OS processes in order to achieve parallelism. Standard OS process creation is, however, normally a costly operation when it is used as intensively as implied by our computation model. A large portion of the cost is due to the copying of the parent environment (memory contents), although only a small part of it is usually accessed by the children.

Other source of per-process overhead come from the fact that the *fork*-like primitive is offered as a mean to create processes that are intended to be independent tasks. Compared to our needs it incurs additional overhead stemming from the management of the execution environment, such as file descriptors, communication ports, etc. Clearly, our computation model is a single task with multiple branches and most of the functionality that causes the overheads is not needed.

A common technique to tackle the mismatch is to simulate processes at the user level, rather than using OS processes. In order to avoid confusion, we use another term, *el-thread* (for ElipSys-thread), to refer to the (logical) process concept we have discussed so far, i.e., a *sequential piece of work* between two OR-nodes. The term *process* is reserved to refer to OS entities.

In the simulation approach, a number of OS processes, called *workers*, are created at system initialization time, possibly one per processor. Each el-thread has then a user-level thread control block and a user-level implemented scheduler allocates each el-thread to a worker when an idle worker is available ([9, 11, 6, 21]).

2.4 Parallel execution models: Reducing the number of processes

Even when el-threads are simulated at the user level, their management (creation, scheduling, completion) is costly. This cost is proportional to the number of el-threads generated.

Because the number of el-threads is normally much higher than the number of available processors, an obvious improvement is to throttle the number of el-threads generated.

The OR-nodes are partitioned into *branch points*, whose branches will be processed in parallel by forking el-threads, and *choice points*, whose branches will be processed sequentially. This can be achieved either statically by annotations of certain OR-nodes, or dynamically by using some load information, or both. When the branch points are appropriately chosen, we can reduce the number of el-threads significantly without observing a significant loss in parallelism.

Consequently the execution of an el-thread does not correspond to the traversing of a sequence of nodes between two OR-nodes but rather to the traversing of an "included" subtree of the original tree.

The machinery for traversing this subtree can be then based on a sequential one for instance the WAM. As a matter of consequence, from now on, it is considered that an el-thread executes a WAM machinery and therefore owns some area of (virtual) memory where its WAM stacks and data structures are implemented. A variable is said to be *shared* or *non-local* if has been allocated before the last traversed branch point.

2.5 Shared Binding Environments

The copying of parent stacks is the most important overhead associated with each el-thread creation. There have been several schemes that are based on *shared environments* ([15]). The common property of these schemes is that the parent environment is shared by all children, rather than copied. However the shared environments are read-only for the children. The parent is idle when there are live children and hence does not modify the shared environment either. Modifications to a shared environment are achieved by using auxiliary data structures, which can be viewed as virtual copies of the environment. Because it is often the case that only a small portion of the parent environment is actually accessed by the children, the copying overhead of these schemes is significantly reduced.

The Hash Window model. In the hash window model ([7]), the parent stack space is not copied at fork time, instead, a private data structure (a hash table), called the *hash window*, is associated with each el-thread and is initialized to empty.

When binding a non-local variable, instead of modifying the variable directly, the value of the variable is entered in the hash window (with the address of the variable as its key). Accordingly, to retrieve the value of a non-local variable, a search in the hash window of the current el-thread has to be done. Note that if the variable has not been previously cached in the hash window, the search must go through the hash windows of the parent, the grandparent, and so on, until a hash window holding a binding for the variable is found. Only if no such hash window is found (i.e no ancestor of the el-thread has bound the variable), the variable is considered free. Figure 1 shows the structure of hash windows.

The hash window binding scheme is used in Argonne National Lab's parallel Prolog ([10]). An optimization of the scheme based on caching accesses and sequentialization has been used in the PEPSys system and shown to be a satisfactory approach ([5]).

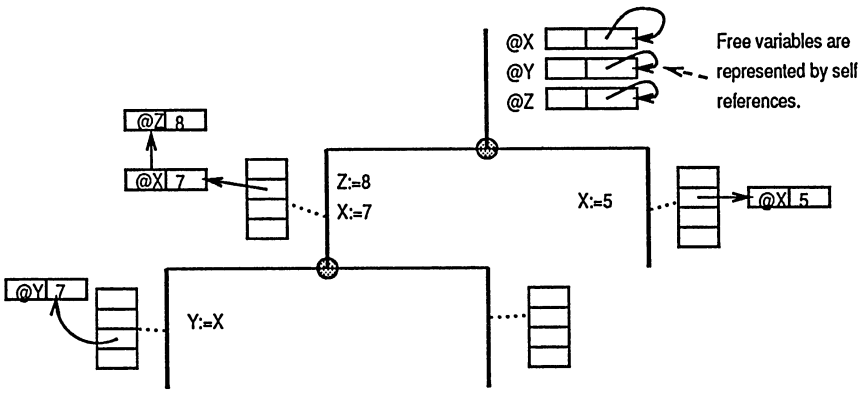


Figure 1: The hash window model

The Binding Array model. The binding array model is another scheme based on shared environments. This method has been independently proposed by D. S. Warren ([24]) and by D. H. D. Warren, and is the basis of the Aurora Parallel Prolog implementation ([19]).

A *binding list* is associated with each el-thread. The elements of the binding list are pairs of the form $(Vid, Value)$, which records the modifications to non-local variables Vid. Bindings to non-local variables are achieved by appending new binding pairs to the binding list, without modifying the variables themselves; bindings to local variables are made by modifying the variables in place.

Since the values of non-local variables are stored in the form of a list, retrieving the value of a non-local variable would require a linear search in the binding list. As a standard optimization technique, a data structure called the *binding array* is associated with each *worker* to cache the bindings. The binding array must be initialized from the binding lists when context switching from one branch to an other. Each non-local variable is assigned an index, starting from 0 and up. The indices can be stored in the variables themselves. When the binding to a non-local variable is made, beside updating the binding list, the corresponding entry in the binding array is also updated. Therefore, the value of a non-local variable can be obtained in the binding array immediately without searching the binding list (Figure 2).

3 Virtual Memory Execution Model

3.1 Virtual Memory Support

Many advances have emerged in the field of parallel operating systems during the last decade. Examples include memory sharing, copy-on-write techniques, lazy memory allocation and lazy page mapping, sparsely used address space, shared virtual memory on distributed memory machines, and light-weight threads.

As already mentioned the naïve execution model could be realized by implementing

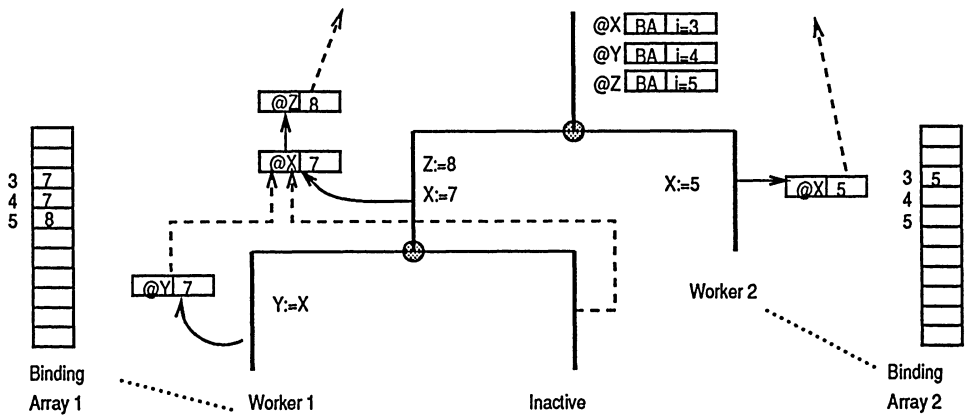


Figure 2: The binding array model

el-threads with "standard" processes, each having its own address space, but the overhead of process creation (a large part of which is due to memory copying) made the approach unrealistic. Modern operating systems such as Chorus[2] ([1]) and Mach ([22]) make aggressive use of *copy-on-write* techniques to reduce the cost of copying data in operations such as message passing, accessing memory-mapped files, creating new processes, and so on. With the copy-on-write technique, it is now possible to consider the naïve model seriously, because the cost of memory copying can be reduced dramatically for the patterns of memory references in the computation model.

3.2 Virtual Memory Model

Let us take Mach, which is an existing operating system that implements the idea of memory inheritance, and show by an example how the naïve model can be implemented. In Mach, the traditional concept of *processes* is split into two parts: *tasks* and *threads*. A task is a unit of resource allocation, which includes an address space, a number of threads, file descriptors, communication ports, and so on. A thread is like a process without resources; it cannot exist on its own but must be associated with a task. A traditional Unix process is therefore a Mach task with a single thread and the target onto which an el-thread is mapped.

The address space of a task can be inherited by its child tasks according to the inheritance property associated with each region of its address space. A region is a block of contiguous memory locations, subject to the restriction that the starting and the ending addresses must be on page boundaries. A memory region can have one of the following three classes of inheritance property:

- `VM-INHERIT-SHARE`. The region is shared by its child tasks.

[2]Chorus is a registered trademark of Chorus Systèmes.

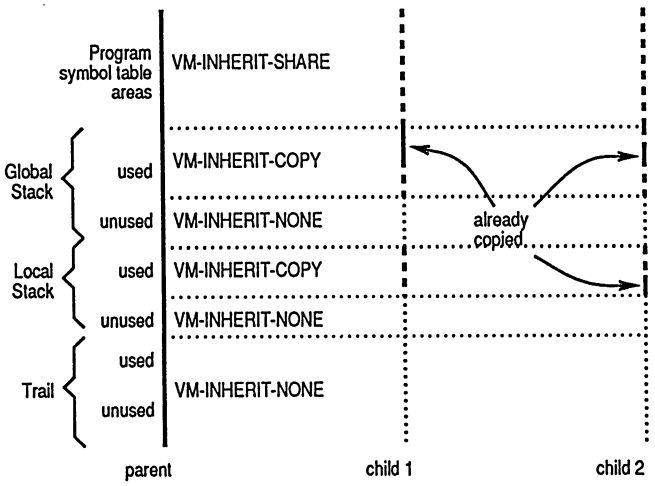


Figure 3: Inheritance Properties of Different Regions

- `VM-INHERIT-COPY`. The region is copied (lazily, i.e., *copy-on-write*) by its child tasks.
- `VM-INHERIT-NONE`. The region is unmapped in the child tasks.

With these mechanisms, we can implement the naïve model by mapping an el-thread onto a Mach task with a single thread. Since the virtual memory inheritance is the key issue, we rename the model as the *virtual memory* execution model from now on. All three types of inheritance property are needed for our execution model (see Figure 3). The regions for the program area, the symbol tables and the internal databases are always shared. The terms stacks are divided into two parts: the used part and the unused part at the time of forking. The former is (lazily) copied and the latter is unmapped in the child's address space. The trail stack is, however, private to the owner and is not mapped in the children address spaces. Figure 3 gives an intuitive view, where dashed lines represent pages that are shared with the parent, solid lines represent pages that are already copied, and dotted lines are unmapped pages.

Figure 4 shows the difference between the naïve model implemented with traditional processes and the virtual memory model implemented with memory inheritance technology. In the latter case pages that are not modified are not copied, shown as dashed lines.

3.3 Owning Threads

When the copying overhead is reduced by means of copy-on-write, other sources of overhead of process creation become significant. These heavy-weight processes are the basic units of resource allocation and offer a multi-purpose set of functionalities which are not

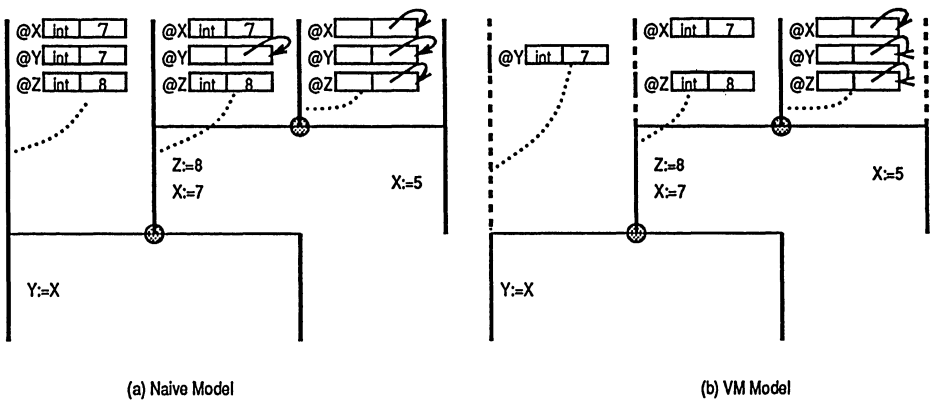


Figure 4: The Virtual Memory Execution Model

needed by the el-threads (files, communication ports, threads). Their management (creation, scheduling, termination) thus incurs an unnecessary overhead. It is for this reason that the user-level execution models reviewed in the previous section have been introduced. The time for a Mach task creation operation on a SUN3 is in the order of tens of milliseconds ([22]). It is estimated in ([25]) that on the same machine a forking time of less than a millisecond is needed for an OS process based implementation. The above proposal for the implementation of the virtual memory model is therefore only practical for parallel computation with a large grain size.

Threads have been introduced in operating systems as alternative entities to heavy weight processes for writing software where the use of simultaneously and concurrent active computations is desirable. Threads are light-weight with respect to scheduling, creation and destruction compared to the higher level entity they belong to (task in Mach or actor in Chorus).

While threads allow a new programming style which is not affordable with standard processes, they lack some of their features and cannot replace them. In particular, all threads within one task or actor share the same address space, making it impossible to implement the virtual memory model presented above, mapping an el-thread onto an OS thread.

Therefore, we propose a new type of thread, called an *owning thread*, which is mainly a Chorus or Mach thread but owns an address space. The address space is inherited by the child owning threads in a similar way to that of a Chorus actor or a Mach task. A further restriction we impose is that a parent owning thread always sleeps until its offspring die. This restriction fits with our computational model and the computational models of many other parallel applications.

Like Mach threads, owning threads are running inside a higher level abstraction (actors or tasks) which provides them with common resources (such as open file table and message ports) and their protection. A fixed region of each owning virtual space is used to store the common resources and is forcibly shared.

3.4 Implementation Notes and Optimizations

Like in the virtual memory system of Mach or Chorus, pages to be copied to child threads are set to read-only at forking time. This operation has a cost linear to the size of the region to be copied. It could be thought to be a flaw in the scheme as it is claimed in Mach or Chorus that the time needed to fork a new task or actor is almost independent of the size of the space to be copied ([1, 22]). One must bear in mind that it comes from the small cost of scanning the memory map (that these systems do as well to protect the copied region from further write) compared to the cost of managing other structures and handling all the complexity of a general virtual memory system.

Owning threads are scheduled in or out during the quantum of time allocated to their task or actor. Switching between two owning threads (within a quantum of time) is first done by unmapping the copied parts of the address space of the active one and then mapping the copied address space of the one to be scheduled in. The shared parts of the address space are not touched.

Putting a parent owning thread to sleep until its offspring die means that a piece of memory copied to a child is not modified until all children terminate. When an update occurs, the original is directly available for copying through the sleeping parent's control block. The mechanism of *history objects* ([1]) is not needed.

All the virtual memory handling is done when the operating system performs a scheduling operation or answers a fork call, that is, when the processor is in kernel mode. An immediate consequence of this is that only one call to the kernel is needed to fork and copy the memory regions. There is no need to make additional system calls to manage the address space. This is a result of the integration of virtual memory handling in the concept of owning threads.

The memory inheritance mechanism we have discussed so far is very general: there can be several different regions to be (lazily) copied, each of which can start anywhere in the user address space and have a variable size. We observe that this generality is not necessary for our execution models: in the virtual memory model above, the number of regions to be copied is fixed (one for each Prolog stack segment) and their starting addresses are also fixed; in the VMBA model that we will present in the next section, the number of copy-on-write regions is reduced to one, although still having a variable size; and in the VMHW model, also presented in the next section, there is only one region (for the hash window) to be (lazily) copied, which starts from the same address and has a fixed size. The simpler the layout of the copied address space is, the simpler and the more efficient the forking and the scheduling can be made.

4 VMHW and VMBA Models

The underlying idea of the execution models proposed in the following is to map el-threads onto owning threads. The virtual memory mechanisms associated with owning thread allows to achieve lazy copying of a parent's environment into the children's environment. This has somehow the same flavor as the binding technique presented in [12] for the Token Machine. The approach depicted here differs in that it bridges the gap between binding schemes which are normally considered different.

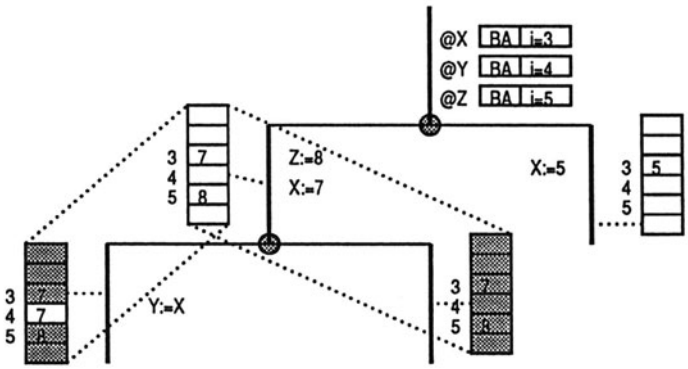


Figure 5: The Virtual Memory Binding Array Model

4.1 VMBA Model

In the virtual memory model, one region is used for each stack of the execution model. The cost of copy-on-write is somewhat proportional to the number of regions to be copied and the number of pages in each region.

We propose here another virtual memory based execution model, called the *virtual memory binding array* model (VMBA model), which reduces the number of copied regions to one.

In the VMBA model, each variable is assigned a binding array index as in the BA model. Unlike the BA model, binding lists are not used, and the original information of bindings to shared variables are directly stored in the binding array. A binding array is therefore needed for each el-thread, unlike in the BA model, where only a single BA is needed for all el-threads running in the same worker and where the contents of the BA is updated during el-thread context switches.

Since the bindings made in the binding array of the parent el-threads are valid for the current el-thread, its binding array should be initialized to the same state of its parent. Apparently we can again use the copy-on-write technology for the binding array region. Note that all the Prolog stack regions are read-only now and do not need to be copied. We therefore manage to reduce the inherited regions to one, and the size of the region can be smaller than in the VM model. Figure 5 shows the VMBA model and its memory inheritance pattern. To simplify the picture, the page size is assumed to be one word in the figure. The pages which have not actually been copied are shaded.

The most striking difference between the VMBA model and any other binding array based models is the low cost of context switching from one branch of the OR-tree to another. It is done in constant time thus giving the same freedom for scheduling the el-threads processing as for instance in the classical hash window based schemes.

The VMBA model has been presented as an improvement of the VM model in that the region to be copied at forking time is smaller. Nevertheless it has still potentially an unbound size. This is a drawback also to be encountered in classical binding array

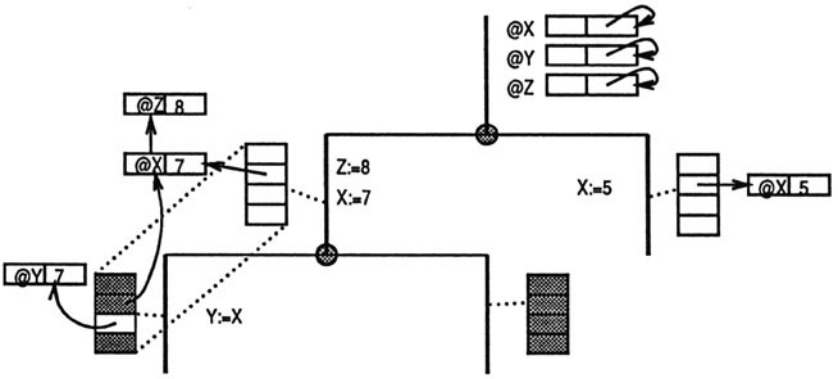


Figure 6: The Virtual Memory Hash Window Model

implementations. A zone of memory in virtual space has to be statically dedicated to the binding array and given a size hopefully big enough to cater for the majority of the computations to be run. As a first consequence the system may run out of memory with a full binding array and some available space elsewhere in the stacks or vice versa. Secondly, the flexibility required for efficiently and dynamically modifying regions layout is a strong requirement that might be not fulfilled by the operating system.

4.2 The VMHW Model

In Section 2.5 we reviewed the hash window model. We now introduce the *virtual memory hash window* (VMHW) model, which fits better with the concept of memory inheritance. Instead of being initialized to an empty value and linked with the preceding hash window, an el-thread's newly created hash window is initialized with the *value*,i.e with a *copy*, of the parent's hash window. The copying is performed lazily by using owning threads. *One region of fixed size* is dedicated in the virtual space to the implementation of the hash window and is copied by copy-on-write at forking time.

Note that in the HW model, getting the value of a shared variable may require a two-level search: searching the chain of hash windows from the current el-thread to the el-thread where the variable is created, and searching the collision chain of each hash window in this chain. In the VMHW model, a single level search (through the collision chain in the current hash window) is enough, because the collision chains of ancestor hash windows are inherited (Figure 6).

A major overhead of the hash window model or VMHW model is that the collision chains may be very long as the computation proceeds and the time to access a shared variable is unbound. A larger hash window size can reduce the problem, but then the cost of hash window initialization increases, consequently its memory consumption increases.

The contradiction over the hash window size can be partly solved using the copy-on-write technology for the VMHW model. Although it is still not cost-free to use an arbitrarily large hash window size, a much larger size than in the "standard" hash window model becomes feasible now.

The size of the hash windows can be used as a *tuning parameter* for exploiting at its best the implementation of owning threads furnished by the operating system. This model inherits its flexibility from the hash window scheme, while using owning threads brings the possibility of using the virtual memory hardware in a *dedicated* and *protected* way for performing operations previously done by software at user level.

As we have mentioned in the previous section, the memory inheritance used in the VMHW model is in a very restricted form. Only the region for the hash window needs to be copied, and its starting address and size are fixed (the collision chains are stored in the global stacks, not in the hash window regions). It is then possible to implement owning threads more efficiently by dedicating to the style of inheritance.

4.3 Implementation Issues

Shared memory machines. On this type of machine two different processing elements can make references to the same piece of physical memory. A child thread running on a different processing element than its parent can therefore reference in its private memory a piece of physical memory initially allocated to its parent. Hence it is possible to implement lazy-copying by using copy-on-write techniques as described previously in Section 3.

Distributed memory machines. A priori these share-nothing machines only allow the use of copy-on-reference for achieving lazy copying. An owning thread is created with an invalid private memory. A first access to it triggers the copying from the remote parent. In such an environment the owning threads implementation must be coupled with a Distributed Shared Memory system ([18, 17, 8]) and enable the reuse of the copies of parents' private memories which have already been fetched in (caching) between different owning threads.

User level implementation of memory inheritance. The virtual memory models we presented above and the copy-on-write techniques can be implemented (simulated) at user-level by emulating with software the address translations done by the virtual memory hardware. Tests for checking whether the accessed locations are in valid pages and are writable must also be performed by software. The tests are still executed even when the accessed location has already been copied thus yielding an unnecessary overhead. Write accesses to a location which has not yet been copied trigger its copying from the parent.

Because the address mapping must be simulated in software, multiple level address mapping is very costly. Therefore the user-level implementation of virtual memory inheritance is only of useful efficiency when the regions to be copied are small, for instance, the VMHW model configured with a hash window size of a few tens of entries.

However, the user level implementation can be used as a simulation tool to study the performance of the VMBA model or the VMHW model with a large hash window size. In the following section we report some results obtained from our user level implementation.

Program	Sequential	1 processor	8 processors	speed-up
8 queens	7.95	9.54	1.29	7.40
hamilton	48.80	69.50	8.87	7.84
farmer	4.84	6.41	2.24	2.86
map	2.73	5.44	0.77	7.06
mandelbrot	9.57	10.01	1.32	7.58
16 queens(1st. solution)	120.21	145.42	7.18	20.25
hamilton(1st. solution)	0.51	0.75	0.28	2.68
biological		1290.40	187.02	6.90

Figure 7: VMHW: speed up by using 8 processors; execution times in seconds

5 Experimental Results

It is not possible to implement the execution models proposed in this paper before operating systems with the required support are available. To test our ideas, we have implemented an ElipSys prototype system based on an old version of SB-Prolog ([13]). The virtual memory system is simulated, using software virtual address translation. Copy-on-write is also simulated. Both the VMHW and the VMBA models have been implemented with this simulated virtual memory system.

The prototype system, which we call PSB-Prolog, is currently running on a 12 processor Sequent Symmetry.

5.1 Speed Ups

As we have said, when a small hash window size is used for the VMHW model, a user level implementation of memory inheritance is good enough (actually better than OS supported virtual memory systems) as a real implementation.

Figure 7 shows the execution times for a set of benchmark programs using the VMHW model with a hash window size of 32 words (128 bytes). The data for both a single processor configuration and an 8 processor configuration as well as the data for a pure sequential implementation of SB-Prolog are listed to show the speed up.

Figure 8 compares the execution times of PSB-Prolog using the VMHW model and the VMBA model with that of PEPSys, the previous ECRC OR-parallel system [5], on the same set of benchmark programs, all using 8 processors. The page size simulated in the VMBA model is 32 words or 128 bytes. Although the two abstract machines of PEPSys and PSB-Prolog are not the same the figure shows that our simulations are comparable with a classical OR parallel logic system.

The efficiency of the user-level VMBA model is not as promising as that of the user-level VMHW model. This is partly due to the overhead of simulating address translation and copy-on-write in software.

Program	PEPSys	VMHW-32	VMBA
8 queens	2.49	1.29	1.74
hamilton	9.35	8.87	12.18
farmer	3.10	2.24	2.77
map	0.87	0.77	1.33
mandelbrot	1.78	1.32	1.49
hamilton(1st. solution)	0.30	0.28	–
16 queens(1st. solution)	–	7.18	7.06

Figure 8: Comparing VMHW and VMBA models with PEPSys; execution time in seconds

Program	NLD-rate	Hit-rate	deref/thread	page/thread
8 queens	7.6%	84.2%	49	0.77
hamilton	15.8%	82.4%	24	0.93
farmer	17.7%	87.0%	32	0.86
map	18.0%	70.7%	10	0.76
mandelbrot	2.2%	97.6%	1013	0.72
16 queens (1st. sol)		86.3%		0.91

Figure 9: Memory Reference Patterns in VMBA

5.2 Memory Reference Behavior

In Figure 9 some memory reference data for the VMBA model are shown. The page size used for simulation is 32 words (128 bytes), and the underlying machine is assumed to have a shared memory architecture. *NLD-rate* refers to the percentage of non-local dereferences among the total number of dereferences. *Hit-rate* refers to the percentage of non-local accesses (non-local reads and writes) to copy-on-write regions which do not trap for copying. *Page/thread* is the average number of pages actually copied by each owning thread. Finally, *deref/thread* is the average number of dereferences performed by each owning thread and is used as a measure of the size of the thread (granularity).

Figure 10 shows the memory reference data collected for the VMHW model on a shared memory machine. Hash-windows are 32 bytes big, the same as a page. *NLD-rate* and *deref/thread* are the same as for the VMBA model and are not listed here. *Ave. Search Length* is the average length of search in the collision chains of hash windows for every non-local variable access. The average search length is practically short although potentially unbound. The *page/thread* ratio of VMHW is better than the one of VMBA. Considering additionally two these two results that the cost of handling inheritance is higher in VMBA than in VMHW, the latter seems to be a better choice than the former.

From the figure we see that the expected length of search to the collision chains of hash windows is only slightly more than one. This is partly due to the reason that all the benchmark programs but the biological one are small, but on the other hand, it indicates that a good locality is kept by inheriting hash windows.

Program	Ave. Search Length	page/thread
8 queens	1.05	0.76
hamilton	2.10	0.85
farmer	1.11	0.77
map	1.01	0.73
mandelbrot	0.81	0.67
16 queens (1st. sol)	1.59	0.85
biological	1.18	0.58

Figure 10: Hash window copying and search in VMHW

5.3 Analysis and Comparisons

The VMBA model can be viewed as an extreme case of the VMHW model when the size of the hash windows is infinite (indeed, as big as the number of non-local variables). In the other extreme, the VMHW model becomes the standard binding array model when the size of the hash window is reduced to one (corresponding to a binding list), although the latter uses a per-worker binding array to buffer the "collision" chains (binding lists).

As shown in Figure 11, the smaller the hash window size, the smaller the inherited regions, the more efficient a user level virtual memory simulation is and thus the support required from the underlying operating system is reduced. On the other hand, the larger the hash window is, the shorter the collision chains are and thus accesses to non-local variables are faster. An exception is the standard binding array model, in which accesses to non-local variables are made very fast by caching the binding lists in the binding arrays at the expense of updating binding arrays on el-thread context switches.

It is not clear where the "best" point is, and this best point may be dependent on the level of support from the operating system. It is our future work to find through experiments the relationship between the efficiency and the various factors. Our initial results suggest that that the VMHW model with a tunable hash window size may be the appropriate choice for ElipSys. The VMHW model also demonstrates a good locality of references.

6 Conclusion and Future Work

We have proposed a series of executio models for OR-parallel logic systems based on the use of hardware support for a well-defined and simple operating system paradigm, owning threads. This paradigm is a new kind of thread allowing lazy copying of data between parents and children. The requirements put on those threads the sufficient minimum to implement the models. The efficiency of standard thread handling (creation, switching) is thus maintained. User-level simulations of this proposal show that it compares favorably to other classical models. This is an incentive for pushing forward the idea and realizing it at the operating system level.

It is our next goal to analyze the behavior of the VMHW model with bigger of hash-

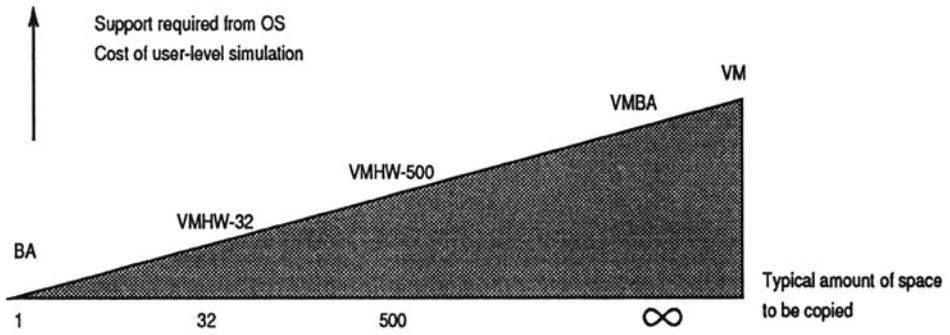


Figure 11: A general view of different binding schemes

window sizes, to estimate the cost of owning thread creation and its influence to the virtual memory based models more precisely, and to study the behavior of the models in a distributed memory architecture.

Acknowledgments

This work was partially funded by the CEC as part of ESPRIT II project EP2025, European Declarative System (EDS).

We want to thank Mike Reeve and Michel Dorochevsky for their useful comments on previous versions of this paper.

References

[1] Vadim Abrossimov and Marc Rozier. Generic virtual memory management for operating system kernels. In *Symposium on Operating Systems Principles*, pages 123–136, December 1989.

[2] K. Ali. Muse. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1531–1545, Seattle, August 1988.

[3] K. Ali. OR-parallel execution of Prolog on theBC-machine. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1531–1545, Seattle, August 1988.

[4] H. Alshawi and D. Moran. The DELPHI Model and Some Preliminary Experiments. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1578–1589, Seattle, August 1988.

[5] U.C. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J.C. Syre, and H. Westphal. The parallel ECRC Prolog system PEPSys: An overview and

evaluation results. In *Proceedings FGCS'88*, Tokyo, November 1988. International Conference on Fifth Generation Computer Systems.

[6] A. Beaumont, S. Muthu Raman, and P. Szeredi. Scheduling OR-Parallelism in Aurora with the Bristol scheduler. In *Proceedings PARLE*, Eindhoven, June 1991.

[7] P. Borgwardt. Parallel Prolog stack segments on shared-memory multiprocessors. In *Proceedings Symposium on Logic Programming*, pages 2–11, February 1984.

[8] Lothar Borrmann. A virtually shared memory model with customized coherency. In *11th ITG/GI Conference on Architecture of Computing Systems*, March 1990.

[9] R. Butler, T. Disz, E. Lusk, R. Olson, R. Overbeek, and R.Stevens. Scheduling OR-Parallelism: An Argonne Perspective. In K. Bowen R. Kowalski, editor, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1590–1605, Seattle, August 1988.

[10] R. Butler, E.L. Lusk, R. Olson, and R.A. Overbeek. ANLWAM - A Parallel Implementation of the Warren Abstract Machine. Internal report, Argonne National Laboratory, 1986.

[11] A. Calderwood and P. Szeredi. Scheduling Or-Parallelism in Aurora - the Manchester Scheduler. In *ICLP'89*, pages 419–435. Univ. Manchester, June 1989.

[12] A. Ciepielewski and S. Haridi. A Formal Model for OR-parallel execution of logic programs. In *Proceedings Information Processing*, pages 299–305, 1983.

[13] Saumya Debray. SB-Prolog system, Version 2.5, a user manual. Technical report, Department of Computer Science, University of Arizona, September 1988.

[14] S. Delgado-Rannauro, Kees Schuerman, and Jiyang Xu. The Elipsys computational model. EDS Deliverable EDS.DD.5E.CA-51, Computer Architecture Group, ECRC, December 1989.

[15] S. A. Delgado-Rannauro. *A Message Driven OR-Parallel Logic Architecture.* PhD thesis, University of Essex, England, December 1989.

[16] S.A Delgado-Rannauro. Computational Models of Parallel Logic Languages. Computer Architecture Group, technical report 46, ECRC, February 1989.

[17] Brett D. Fleisch and Gerald J. Popek. MIRAGE: A coherent distributed shared memory design. In *Symposium on Operating Systems Principles*, pages 123–136, December 1989.

[18] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[19] E. Lusk, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D.H.D. Warren, A. Calderwodd, P. Szeridi, S. Haridi, P. Brand, M. Carlson, A. Ciepielewski, and B. Hausman. The Aurora OR-parallel Prolog system. In *Proceedings FGCS'88*, Tokyo, November 1988. International Conference on Fifth Generation Computer Systems.

[20] EDS Project. European Declarative System, Executive Summary. Technical report ESPRIT II EP 2025, EDS Deliverable, 1989.

[21] P. Szeredi and M. Carlsson. The Engine-Scheduler Interface in the Aurora Or-parallel Prolog System. In *distributed in NACLP Workshop on parallel execution*, Austin, Texas, October 1990.

[22] Avadis Tevanian, Jr. *Architecture-Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach.* PhD thesis, Carnegie Mellon University, december 1987.

[23] David H.D. Warren. An abstract Prolog instruction set. Technical report, 309, Artificial Intelligence Center, SRI International, 1983.

[24] David S. Warren. Efficient Prolog memory management for flexible control strategies. In *The International Symposium on Logic Programming*, pages 198–202, 1984.

[25] Jiyang Xu. Elipsys execution models: A preview. Computer Architecture Group, internal report: Elipsys-017, ECRC, February 1990.

Interfacing Engines and Schedulers in Or-Parallel Prolog Systems

Péter Szeredi*, Rong Yang
Department of Computer Science
University of Bristol, Bristol BS8 1TR, U.K.

Mats Carlsson
Swedish Institute of Computer Science
P.O. Box 1263, S-164 28 Kista, Sweden

Abstract

Parallel Prolog systems consist, at least conceptually, of two components: an engine and a scheduler. This paper addresses the problem of defining a clean interface between these components. Such an interface has been designed for Aurora, a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors.

The practical purpose of the interface is to enable different engine and scheduler implementations to be used interchangeably. The development of the interface has, however, contributed in great extent to the clarification of issues in exploiting or-parallelism in Prolog. We believe that these issues are relevant to a wider circle of research in the area of or-parallel implementations of logic programming.

We believe that the concept of an engine-scheduler interface is applicable to a wider range of parallel Prolog implementations. Indeed, the present interface has been used in the Andorra-I system, which supports both and- and or-parallelism.

Keywords: Or-Parallel Execution, Multiprocessors, Implementation Techniques, Scheduling.

1 Introduction

Parallel Prolog systems consist, at least conceptually, of two components: an *engine*, which is responsible for the actual execution of the Prolog code, and a *scheduler*, which provides the engine component with work. This paper addresses the problem of defining a clean interface between these components. We focus on a particular interface which has evolved within the implementation of an or-parallel Prolog system, Aurora. The interface has successfully been used to connect the Aurora engine with four different schedulers. It has subsequently been applied in the implementation of the and-or-parallel language Andorra-I, thus proving that its generality extends beyond or-parallel Prolog.

Aurora is a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors, based on the SRI model of execution [16], and currently running on Sequent and Encore machines. It has been developed in the framework of the Gigalips project [11], a collaborative effort between groups at the Argonne National Laboratory in Illinois, the University of Bristol (previously at the University of Manchester) and the Swedish Institute of Computer Science (SICS) in Stockholm.

The issue of defining a clear interface between the engine and scheduler components of Aurora was raised in the early stages of the implementation effort. Ross Overbeek made the first attempt to

* *On leave from (and present address)* SZKI IQSOFT, Donáti u. 35-45, Budapest, Hungary.

formulate such an interface and Alan Calderwood produced the version [7] used in the first generation of Aurora (based on SICStus Prolog version 0.3).

A fundamental revision of the interface was necessitated by several factors. Performance analysis work on Aurora [14] has shown that some unnecessary overheads are caused by design decisions enforced by the interface. Development of new schedulers and extensions to existing algorithms required the interface to be made more general. The Aurora engine has also been rebuilt on the basis of SICStus Prolog version 0.6.

The new interface, described in the present paper, is part of the second generation of Aurora. The major changes with respect to the previous interface are the following:

- execution is governed by the engine, rather than the scheduler;
- the set of basic concepts has been made simpler and more uniform;
- several potential optimisations are supported;
- the interface is extended to support transfer of information related to pruning operators [10].

The paper is organised as follows. Section 2 summarises the SRI model and defines the necessary concepts. Section 3 gives a top level view of the interface. Section 4 presents the data structures involved in the interface, while Sections 5 and 6 describe engine-scheduler interactions in various phases of Aurora execution. Section 7 shows the extensions: handling of pruning information and various optimisations. Section 8 discusses the major issues involved in implementing the Aurora engine side of the interface. Section 9 describes how the interface was utilised to introduce or-parallelism into the Andorra-I system [12]. Section 10 presents preliminary performance results from the Aurora implementation. We end with a short concluding section.

A complete description of the interface is contained in [15].

2 Preliminaries

Aurora is based on the SRI model [16]. According to this model the system consists of several *workers* (processes) exploring the search tree of a Prolog program in parallel. Each node of the tree corresponds to a Prolog *choicepoint* with a branch associated with each alternative clause. A predicate can optionally be declared *sequential* by the user, to prohibit parallel exploration of alternative clauses of a predicate. Corresponding nodes are also annotated as sequential. All other nodes are *parallel*.

As the tree is being explored, each node can be either *live*, i.e. have at least one unexplored alternative, or *dead*. A node is a *fork node* if there are two or more branches below it; otherwise, it is a *nonfork node*. A fork node cannot be sequential. Live parallel nodes, and live sequential nodes with no branches below them, correspond to tasks that can be executed by workers. Each worker has to perform activities of two basic types:

- executing the actual Prolog code;
- finding work in the tree, providing other workers with work and synchronising with other workers.

In accordance with the SRI model each worker has a separate *binding array*, in which it stores its own bindings to potentially shared variables (conditional bindings). This technique allows constant time access to the value of a shared variable, but imposes an overhead of updating the binding arrays whenever a worker has to move within the search tree.

The or-tree is divided into an upper, *public*, part accessible to all workers and a lower, *private*, part accessible to only one worker. A worker exploring its private region does not have to be concerned with synchronisation or maintaining scheduling data; it can work very much like a standard Prolog engine. The boundary between the public and private regions changes dynamically. It is one of the critical aspects of the scheduling algorithm to decide when to make a node public, allowing other workers to

share work at it. In the majority of schedulers, the worker will make his *sentry* node, i.e. his topmost private node, public when all nodes above it have become dead, i.e. have no more alternatives to explore. This means that each worker tries to keep a piece of work on its branch available to other workers.

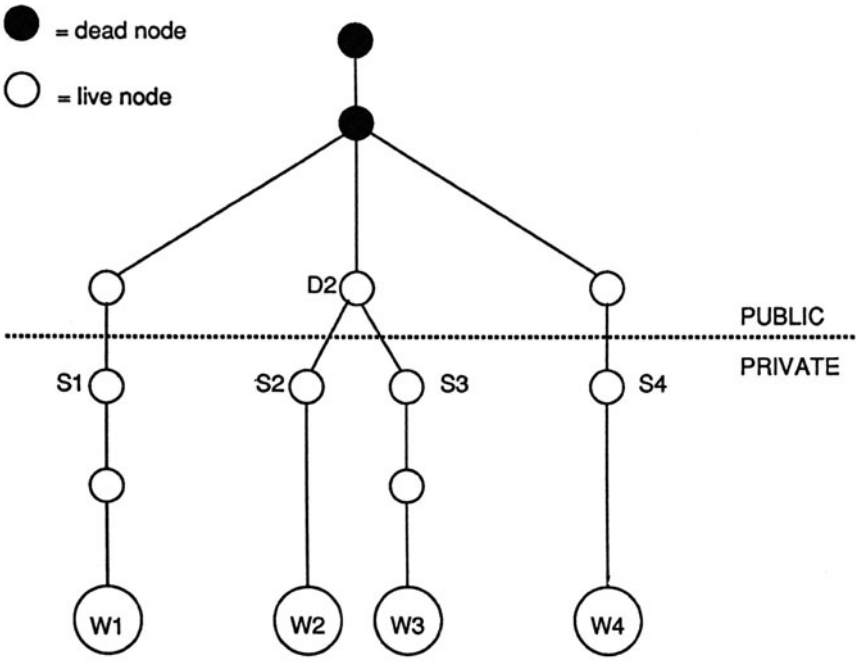


Figure 1: THE OR-TREE OF THE SRI MODEL

For example, in Figure 1, an or-tree being explored by four workers (W1–W4) is shown. The workers' sentry nodes are denoted S1–S4. Assume that there is an unexplored alternative at node D2. Now if the branch being explored by worker W1 dies back and W1 takes the alternative at D2, the node D2 will become dead, and the scheduler will normally extend the public region to include nodes S2–S3 so as to keep a piece of work available on every branch.

The exploration by a worker of its private region constitutes that worker's *assignment*, which normally terminates if the worker backtracks into the public part. The assignment terminates prematurely if the branch is *suspended*, or if it is *pruned* by some other worker.

There are three *pruning operators* currently supported by Aurora: the conventional Prolog *cut*, which prunes all branches to its right and a symmetric version of cut called *commit*, which prunes branches both to its left and right. A cut or a commit must not, and will not, go ahead if there is a chance of being pruned by a cut with a smaller scope. The third type of pruning operator is the *cavalier commit* which is executed immediately, even if endangered by a smaller cut. The cavalier commit is provided for experimental purposes only, it is expected to be used in exceptional circumstances, for operations similar to `abort` in Prolog. Work done in the scope of a pruning operator is said to be *speculative*.

Suspension is used to preserve the observable semantics of Prolog programs executed by Aurora: when a built-in predicate with some side-effect is reached on a non-leftmost branch of the search tree, or when a pruning operator is reached on a branch which could be pruned by a cut with a smaller scope, the execution must be suspended. Furthermore the scheduler may decide to suspend the current

branch when less speculative work can be done somewhere else in the tree.

Four separate schedulers are currently being developed for Aurora. The Argonne scheduler [6] relies on data stored in the tree itself to implement a local strategy according to which live nodes "attract" workers without work. When several workers are idle they will compete to get to a given piece of work and the fastest one will win. The Manchester scheduler [8] tries to select the nearest worker in advance, without moving over the tree. It uses global data structures to store some of the information on available work and workers. The wavefront scheduler [5] uses a special distributed data structure, the *wavefront*, to facilitate allocation of work to workers. The Bristol scheduler [3] tries to minimise scheduler overhead by extending the public region eagerly: sequences of nodes are made public instead of single nodes, and work is taken from the bottommost live node of a branch.

3 The Top Level View of the Interface

The principal duty of the scheduler is to provide the engine with work. The thread of control thus alternates between the two components: the engine executes a piece of Prolog code, then the scheduler finds the next assignment, passes control back to the engine, etc. A possible way of implementing this interaction is to put the scheduler *above* the engine: the scheduler *calls* the engine when it finds a suitable piece of work to be executed and the engine *returns* when such an assignment has been finished. In fact this scheme was the basis of earlier interfaces in Aurora [7].

We use a different approach in the current version of Aurora. The execution is governed by the engine: whenever it finishes an assignment, it calls an appropriate scheduler function to provide a new piece of work. The advantage of this scheme is that the environment for Prolog execution (e.g. the set of WAM-registers) is not destroyed when an assignment is terminated and need not be rebuilt upon returning to work. This is of special importance for Prolog programs with fine granularity (i.e. small assignment size), where switching between engine and scheduler code is very frequent [14].

Figure 2 shows the top view of the current interface. This is centered around the engine doing work. All the other boxes in the picture represent scheduler functions called by the engine. Note the convention that the names of all scheduler functions are prefixed with '`Sched_`'.

The functions shown in Figure 2 are arranged in three groups:

- finding work (left side of Figure 2);
- communication with other workers during work (lower part of Figure 2), e.g. when cuts or side effect predicates are to be executed;
- certain events during work that may be of interest to the scheduler (right side of Figure 2), e.g. creation and destruction of nodes.

The four boxes on the left of Figure 2 represent the so called *functions for finding work*:

`Sched_Start_Work` is used to acquire work for the first time, immediately after the initialisation of the worker;

`Sched_Die_Back` is called when the engine backtracks to a public node;

`Sched_Be_Pruned` is invoked when the worker's current branch is pruned off by another worker;

`Sched_Suspend` is called when the worker has to suspend its current branch.

These functions differ in their initial activities, but normally continue with a common algorithm for "looking for work" (see Section 5). This algorithm has two possible outcomes: either work is found, or the whole system is halted. Correspondingly each of the functions for finding work has two exits: the normal one (shown on the right side of the function boxes in Figure 2) leads back to work, while the other exit (left hand side) leads to the termination of the whole Aurora invocation.

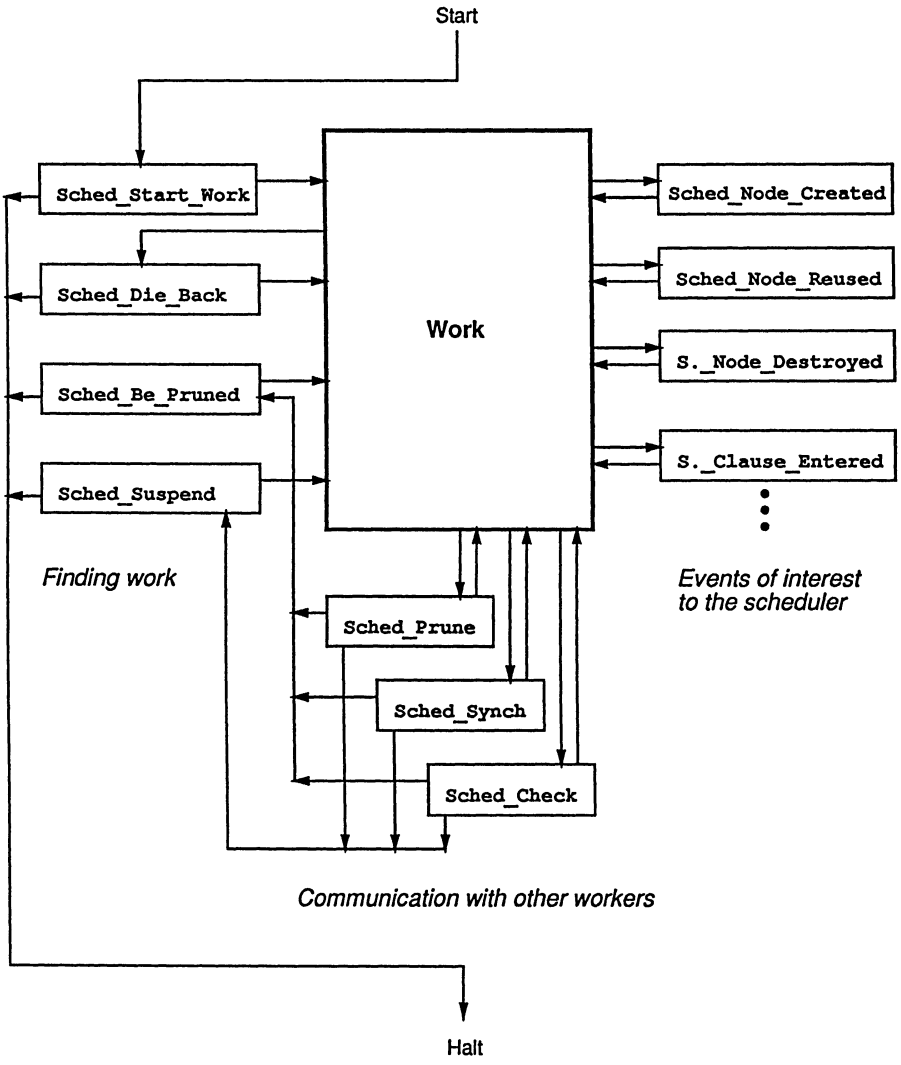


Figure 2: THE TOP LEVEL VIEW OF THE INTERFACE

The next group of interface functions provided by the scheduler is depicted at the bottom of Figure 2. These functions are called during work, when the engine may require some assistance from the scheduler (mainly in order to communicate with other workers):

`Sched_Prune` — when a cut or commit is executed;

`Sched_Synch` — when a predicate with side effects is encountered;

`Sched_Check` — at every Prolog procedure call (to check for interrupts).

The above functions have three exits. The normal exit (depicted by upwards arrows in Figure 2) leads back to work. The other two exits correspond to premature termination of the current assignment, when the current branch has been pruned or has to suspend (leftward and downward arrows). In both cases the engine will do the housekeeping operations necessary for the given type of assignment termination, and proceed to call the scheduler to find the next assignment. See Section 6 for a more detailed description of the functions for communication with other workers.

The third group of functions shown in Figure 2 (right hand side) corresponds to some events during work that may be of interest to the scheduler. A common property of this group is that the interface does not prescribe any specific activity to be done by these functions: the scheduler is merely given an opportunity to do whatever is needed for maintaining its data structures. As an example, `Sched_Node_Created` (and the corresponding `Sched_Node_Destroyed`) can be used to keep track of the presence of parallel nodes in the private region—as a prospective source of work for other workers. Similarly `Sched_Clause_Entered` can be utilised for maintaining information about the presence of pruning operators in the current branch (see Section 7.2).

There are further groups of scheduler functions, not shown in Figure 2. These are used in the initialisation of the whole system, in handling keyboard interrupts , and in the implementation of certain optimisations (Section 7.1).

The engine side of the interface consists of several groups of functions that support the scheduler algorithm:

- providing access to certain data structures (nodes and alternatives) maintained by the engine,
- extending the public region on the current branch of execution,
- positioning the engine (i.e. the binding array) in the search tree, while looking for work,
- notifying the engine of certain events, e.g. work being found.

The data structure aspects of the engine interface are presented in Section 4. Other interface functions provided by the engine will be described in Sections 5 and 6.

4 Common Data Structures

The engine is responsible for maintaining the *node stack*, a principal data area of major importance to the scheduler. The engine defines the *node* data type, but the scheduler is expected to supply a number of fields to be included in this structure for its own purposes.

Among the node fields defined by the engine, some are of interest to the scheduler. Access functions for these fields are provided in the interface:

`Node_Level` — the distance of the node from the root of the search tree,

`Node_Parent` — a pointer to the parent node in the tree,

`Node_Alternatives` — a pointer to the next unexplored alternative of the node.

The scheduler-specific fields of the node data structure normally include pointers describing the topology of the tree. For example, most schedulers will have fields storing a pointer to the first child and the next sibling of a node.

An additional common static data structure, the *alternative*, is introduced to allow the schedulers to keep static data related to clauses. This data structure is used in the Aurora engine to replace the `'try'`, `'retry'` and `'trust'` instructions of WAM [9]. Each clause of the user program is represented by an alternative, which stores a pointer to the code of the clause and a pointer to the successor alternative, if any. If a predicate is subject to indexing, the compiler may create several chains of alternatives to cater for different values in the indexing argument position. This means that several alternatives can refer to the same clause.

The scheduler may supply a number of fields to be included in the alternative structure, to accommodate any (static) information to be associated with clauses. The scheduler can derive this data from the information supplied by the engine when alternatives are created (`Sched_Alternative_Created`). There are two types of static data supplied by the engine:

- information about sequential predicates—this information is normally stored in each alternative of the predicate.
- pruning information—data on the number of pruning operators (cuts, commits and conditional expressions) contained in the clause or the predicate (see Section 7.2).

The only engine field in the alternative structure that is of interest to the scheduler is the one pointing to the successor alternative (`Alternative_Next`). This field is used, for example, when the scheduler starts a new branch from a public node and needs to advance the next alternative pointer of the node.

5 Finding Work

Figure 3 shows the engine functions used by the scheduler while it is looking for work. The actual algorithms of the four functions for finding work will normally differ, but they all use the same set of engine support functions.

Functions `Move_Engine_Up` and `Move_Engine_Down`, shown on the right hand side of Figure 3, instruct the engine to move the binding array up or down the current branch. Initially, the binding array is positioned at or below the youngest public node on the branch. Before returning, the scheduler has to position the binding array above the new sentry node.

Different schedulers employ different strategies in moving over the tree. The Argonne scheduler moves node-by-node, when approaching the potential work node. Other schedulers locate a piece of work from a distance and move the engine to the appropriate place in a few big jumps.

There is no need to move the engine if work is taken from the parent of the old sentry node. An additional entry point to the scheduler, `Sched_Get_Work_At_Parent` (see Section 7.1), has been provided for this special case.

The left hand side of Figure 3 shows the engine functions for memory management of the node stack. A worker may have to remove some dead nodes from the tree as it moves upwards. This involves deleting these nodes from the scheduler data structures (normally the sibling chain) and invoking the `Mark_Node_Reclaimable` engine function. As a special case, the old sentry node will have to be deleted from the tree at the beginning of `Sched_Die_Back` and `Sched_Be_Pruned`.

When the scheduler decides to reserve a new piece of work from a live public node (work node), it has to create a sentry node for the new branch. This involves calling the `Allocate_Node` function, which first removes all the nodes that have been marked as reclaimable from the top of the worker's stack and then allocates a new sentry node. The related `Allocate_Foreign_Node` function is used if *another* worker allocates a node on the stack of the worker looking for work. This is used in the Manchester scheduler to implement handing work to an idle worker.

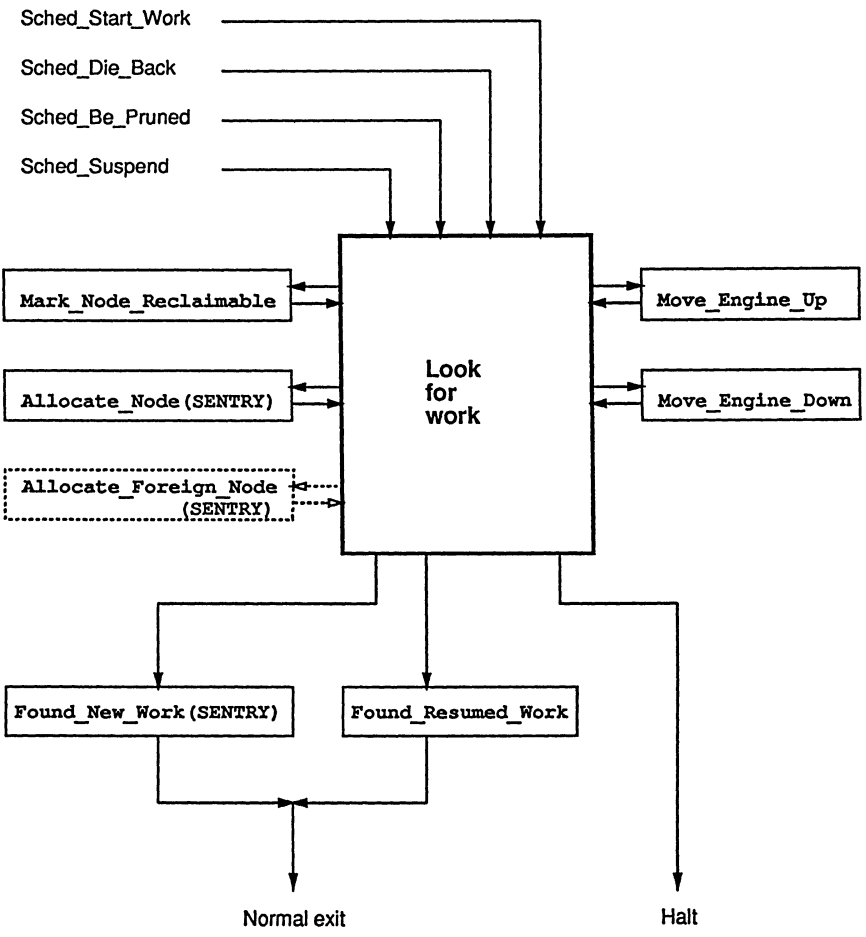


Figure 3: ENGINE FUNCTIONS IN LOOKING FOR WORK

The new sentry node serves as a placeholder for the new assignment. The scheduler inserts the sentry into the search tree and simultaneously reserves an alternative to be explored by the new branch (by reading and advancing the `Node_Alternatives` field of the work node).

The bottom part of Figure 3 shows the possible exit paths from the functions for finding work. The actual work found can correspond either to a new branch or to a branch which was hitherto suspended and can be resumed now. Functions `Found_New_Work` and `Found_Resumed_Work` are used to notify the engine about the type of the work found, and to supply the new sentry node. The box for `Found_New_Work` in Figure 3 shows the `SENTRY` argument to highlight the fact that this argument should be the same as the one returned in `Allocate_..Node`.

6 Communication with Other Workers

The need for communication with other workers arises when a pruning operator or a built-in predicate with side effects is to be executed. In addition, a periodic check is needed to examine if there are communication requests from other workers.

The `Sched_Prune` function is invoked when a pruning operator is encountered. At this moment the engine has already executed the private part of the pruning. The scheduler receives a pointer to the cut node (showing the scope of pruning) and an argument indicating the type of the pruning operator (cut, commit or cavalier commit). It has to check if the preconditions for pruning are satisfied: the current branch should not be pruned itself, and, except for the cavalier commit, it should not be endangered by cuts with a smaller scope, as discussed in [10]. The latter condition can be replaced by a requirement for the branch to be leftmost in the subtree rooted at the child of the cut node, if the scheduler does not maintain specific pruning information.

If the preconditions of pruning are not satisfied, `Sched_Prune` uses one of the abnormal exits (cf. Figure 2) to indicate that the branch has been killed or that it has to suspend (waiting to become leftmost). If the pruning operation can go ahead, the scheduler has to locate the workers that are in the pruned subtree and interrupt them. There may be branches in this subtree which have previously been suspended. A special engine function, `Mark_Suspended_Branch_Reclaimable`, is used for cleaning up such branches.

The `Sched_Synch` function is invoked when a call to a built-in predicate with side-effects is encountered. Normally such calls are executed only when their branch becomes leftmost in the whole tree. There are, however, some special predicates (e.g. those used to assert solutions in a `setof`), for which the order of invocation is not significant: their execution can go ahead if not endangered by a cut within a specific subtree. The `Sched_Synch` function receives an argument encoding the type of the check needed, and a pointer to the root of the subtree concerned.

The third communication function, `Sched_Check`, is called at every Prolog procedure call. Frequent invocation of this function is necessary so that the scheduler can answer requests (e.g. interrupts) from other workers without too much delay. Note, however, that a scheduler may choose to do the checks only after a certain number of `Sched_Check` invocations (as is the case for the Manchester and Argonne schedulers).

The nature of requests to be handled by `Sched_Check` varies from scheduler to scheduler. There are, however, two common sets of circumstances: the worker may be requested to kill its assignment or to make some of its private nodes public (to make work available to other workers). The latter activity needs assistance from the engine: the function `Make_Public` extends the public region on the current branch down to a specified node.

7 Extensions of the Basic Interface

7.1 Simplified Backtracking

When a worker backtracks to a live public node and is able to take a new branch from there, several administrative activities can be avoided. The sentry node can be re-used, rather than being marked

as reclaimable and re-allocated. There is scope for a related optimisation in the scheduler: instead of deleting the old sentry from the sibling chain and then installing it as the last sibling, the scheduler can move the sentry node to the end of the sibling chain (or do nothing if the old sentry was the last child). The interface supports this important optimisation by a function `Sched_Get_Work_At_Parent`, called when the engine backtracks to a live public node. If the scheduler, following the necessary synchronisation operations, still finds the node to be live, it can reserve an alternative from that node. If the scheduler cannot take work from the node in question, it returns to the engine, which will subsequently invoke `Sched_Die_Back` to acquire a new piece of work.

The `Sched_Get_Work_At_Parent` function also supports the *contraction* operation of the SRI model [16]. This operation removes a dead nonfork node after the last alternative has been taken from it. The node in question can be physically removed only if it is on the top of the stack of the worker executing the given branch.

7.2 Pruning Information

Information about the presence of pruning operators in a clause may be needed by the scheduler to perform pruning more efficiently or to distinguish between speculative and non-speculative work. Various algorithms related to pruning have been developed and discussed in [10]. When designing the interface, we tried to generalise and extend the format of pruning data as described in [10], so that other possible approaches (e.g. [13]) can be supported as well.

If one disregards disjunctions, the information needed about pruning is quite simple. A scheduler may wish to know whether a clause contains cuts or commits[1]. For more exact pruning algorithms the number of occurrences of each pruning operator may be needed. The fact that a clause must fail, may also be of interest: when such a clause is entered, the pruning operators in the current continuation (i.e. in the previous resolvent) become inaccessible. The simple set of pruning data would thus consist of three items for each clause: the number of cuts, the number of commits and the Boolean value indicating whether the clause ends in a failing call (i.e. `fail`, but in the future, global compile time analysis might discover this property for other calls).

The presence of disjunctions and conditionals makes the situation more complicated. In [15] we present a set of pruning data consisting of seven items, to describe the pruning properties of a general clause (one that may contain disjunctions and conditionals).

8 Implementation of the Interface in the Aurora Engine

The Aurora emulator [9] was produced by modifying the SICStus emulator to support the SRI model and by converting it from a stand-alone program to an Aurora worker component connected by an algorithmic interface to a scheduler component. The total performance degradation resulting from these changes has been found to be around 25%. In an earlier paper [11] we gave an overview of the changes imposed by the SRI model. In this section we concentrate on the impacts of the interface on the engine and on changes introduced in the new design.

8.1 Boundaries

The engine needs to maintain the boundary between the public and private regions. Within the private region, it must distinguish between *local* nodes, i.e. nodes adjacent to the top of the worker's own stack, and remote nodes. This is achieved by storing a pointer to the respective boundary nodes in certain registers. These registers are initialised when an assignment is started (`Found...Work`). They are updated when the public region is extended (`Make_Public`) or contracted (`Sched_Get_Work_At_Parent`), and when backtracking in the private region winds back to the worker's own stack. They are consulted to distinguish different cases of backtracking and pruning operations.

[1] Note that data on cavalier commits is not included in the pruning information, as this operation is expected to be used only for handling exceptional circumstances.

8.2 Backtracking

From the engine's point of view, the main complication of or-parallel execution is its impact on the backtracking routine. This routine has to check whether it is about to backtrack into the public region, in which case the scheduler must be invoked to perform public backtracking (`Sched_Die_Back` or `Sched_Get_Work_At_Parent`). Private backtracking has to face the complication that the private region may extend to other workers' stacks, and possibly wind back to the worker's own back again. As explained earlier, remote nodes cannot be reclaimed when they are trusted; instead, `Mark_Node_Reclaimable` is invoked when dying back over a remote node.

Shallow backtracking is optimised in the private region, but only if the current node is on the top of the worker's own stack.

8.3 Memory Management

As stated earlier, the stack memory management relies on the node stack. While finding work, each worker maintains a pointer to the youngest node that has to be kept for the benefit of other workers. Such pointers are used and updated by the `Allocate...Node` functions. When an assignment is started (`Found...Work`) the top of stack pointers for the other WAM stacks are initialised from relevant fields of the node physically preceding the embryonic node of the new assignment, as these fields define how much of the other stacks has to be kept.

8.4 Pruning Operators

Pruning operations must distinguish between (i) pruning local nodes only, (ii) pruning remote nodes, and (iii) pruning public nodes. In cases (i) and (ii), the node can be pruned right away, but the memory occupied by the pruned node can only be reclaimed in case (i). The trail must be tidied in all three cases, as explained in [11]. In case (iii), the scheduler is responsible for pruning the public nodes, but may decide to suspend or abort the current assignment instead, forcing the engine to invoke `Sched_Suspend` or `Sched_Be_Pruned`, respectively. Note that `Sched_Prune` is invoked in all three cases, to give the scheduler an opportunity to keep pruning information up to date.

To support suspension of cuts and commits, the compiler provides extra information about what temporary variables need to be saved until the suspended task is resumed. This extra information also encodes the type of the pruning operator.

8.5 Premature Termination

To suspend the current assignment when the scheduler uses the "suspend" exit in `Sched_Prune`, `Sched_Synch`, or `Sched_Check`, the engine creates an auxiliary node which stores the current state of computation and calls `Sched Suspend`. It is up to the scheduler to decide when the suspended work may be resumed.

To abort the current assignment when the scheduler uses the "be_pruned" exit in the above functions, the engine deinstalls all conditional bindings made by the current assignment, marks all remote nodes as reclaimable except the sentry node, and calls `Sched_Be_Pruned`.

8.6 Movement

While executing Prolog code, the binding array is kept in phase with the trail stack: whenever a binding is added to or removed from the trail, the bound value is also stored or erased in the binding array. While finding work, the engine maintains a pointer to a node in the tree corresponding to the current contents of the binding array. When the scheduler asks the engine to "move" the binding array up to a new position (`Move_Engine_Up`), bindings which were recorded on the trail path between the current and the new position are deinstalled from the binding array, and the current position is updated. Similarly, `Move_Engine_Down` installs a number of trailed binding in the binding array and updates the current position.

When an assignment is started (`Found...Work`), the engine positions its binding array at the tip node of the new or resumed branch in order to get ready to start executing the Prolog code.

9 Applying the Interface to Andorra-I

The engine-scheduler interface has been originally designed for the Aurora or-parallel Prolog system. Its primary purpose has been to support exchangeable use of several schedulers with a single engine (i.e. the Aurora engine based on Sicstus). Recently the interface has been used to link the and-parallel engine of the Andorra-I system with the Bristol scheduler developed in the context of Aurora.

In contrast with the Sicstus engine, Andorra-I performs and-parallel execution: any goals which can be reduced without making choicepoints (so called determinate goals) are executed eagerly in parallel; a team of workers work together to exploit and-parallelism. However, when no determinate goals remain, Andorra-I behaves similarly to Prolog: it uses the leftmost goal to make a choicepoint. Moreover, the backtracking routine resembles Prolog, as well: when a goal fails, the team backtracks to the nearest choicepoint, and starts to explore the next branch. Thus, despite the and-parallel execution phase, Andorra-I and Aurora behave in exactly the same way in exploring the or-tree. From the point of view of the interface, an Andorra-I team is exactly the same as an Aurora worker.

In the Andorra-I implementation the following data structures have been introduced to support the interface. First, in a way similar to Aurora, Andorra-I requires two additional pointers for each team: one for marking the boundary between the public and the private regions of the tree, and another for storing the current binding array position. Second, a parent pointer has to be added to each node (Andorra-I originally did not require the parent pointer because of the fixed node size). The backtracking routine is modified so that engine always calls the scheduler (`Sched_Die_Back`), if it is in the public region. To simplify the implementation, Andorra-I currently does not allow a worker to work on other workers' stacks. Therefore, when a worker resumes a suspended branch which belongs to someone else, the branch has to be made public.

The main difference between Aurora and Andorra-I arises in the handling of pruning operators. According to the interface, the engine should call the scheduler whenever it executes a pruning operator (`Sched_Prune`). If the scheduler decides that the pruning cannot go ahead, the engine is required to suspend the current branch and call `Sched_Suspend` immediately. In Andorra-I, however, the pruning operator is executed during the and-parallel phase, and there might be some other goals being executed simultaneously by fellow workers in the team. When a worker needs to suspend because of the pruning operator, it has to take care of its team, i.e. inform all other workers to stop and then find new work together. In fact, even if there is only one worker in the team, it is not easy to stop the and-parallel execution phase prematurely, without slowing down the whole execution process. Therefore, we have decided to let the team carry on the and-parallel phase and suspend later, if necessary. As a special case it may happen that the computation fails after `Sched_Prune` is called. In this case, the Andorra-I engine marks the suspended node as a *cut_fail* node. Later on, when the scheduler resumes the given branch, the engine will backtrack immediately.

Preliminary performance results of the Andorra-I system are very promising [2], showing that Andorra-I is capable of exploiting or-parallelism with similar efficiency as in Aurora. The overall experience of using the interface in the Andorra-I implementation is very positive: the interface proved to be well designed and of appropriate abstraction level.

10 Performance Results

No detailed performance analysis work has been done for the new Aurora implementation yet. Preliminary measurements have been performed with the Manchester scheduler, on the benchmark suite introduced in the performance analysis of the earlier Aurora version [14]. The benchmarks are divided into three groups according to granularity: course granularity (top section in the tables), medium granularity (middle section), and fine granularity (bottom section).

Table 1 shows the running times for that benchmark suite on the first generation of Aurora (using the old interface and an engine based on Sicstus Prolog 0.3). Table 2 shows the running times for the same benchmarks in the second generation of Aurora. There is an overall improvement of up to 60% in terms of absolute speed, mostly due to the new, much faster engine. For some of the fine granularity benchmarks the relative speedups have deteriorated; this is because the increase in engine speed implies a relative increase in scheduler overheads. For benchmarks with coarse granularity, and especially for the ones with frequent suspension and resumption (e.g. tina), the relative speedups have improved, showing the advantages of the new interface.

Goals * repetitions	Aurora Workers 1	4	8	11	Sicstus 0.3
8-queens1	10.11	2.54*(3.98)*	1.29*(7.84)*	0.97*(10.4)*	8.19*(1.23)*
8-queens2	29.37	7.32*(4.01)*	3.73*(7.87)*	2.76*(10.6)*	23.60*(1.24)*
tina	21.30	5.57*(3.83)*	3.02*(7.06)*	2.37*(8.98)*	17.29*(1.23)*
salt-mustard	11.71	3.03*(3.87)*	1.63*(7.18)*	1.27*(9.24)*	9.50*(1.23)*
AVERAGE		*(3.92)*	*(7.49)*	*(9.80)*	*(1.23)*
parse2 *20	9.24	2.92*(3.17)*	2.08*(4.44)*	1.96*(4.72)*	7.54*(1.23)*
parse4 *5	8.54	2.50*(3.42)*	1.67*(5.11)*	1.40*(6.10)*	6.91*(1.24)*
parse5	6.02	1.74*(3.46)*	1.17*(5.15)*	0.98*(6.14)*	4.89*(1.23)*
db4 *10	3.12	0.87*(3.60)*	0.53*(5.87)*	0.45*(6.96)*	2.69*(1.16)*
db5 *10	3.80	1.04*(3.66)*	0.64*(5.93)*	0.55*(6.92)*	3.28*(1.16)*
house *20	8.13	2.26*(3.60)*	1.40*(5.81)*	1.19*(6.84)*	6.51*(1.25)*
AVERAGE		*(3.48)*	*(5.38)*	*(6.28)*	*(1.21)*
parse1 *20	2.49	0.90*(2.77)*	0.81*(3.08)*	0.87*(2.87)*	2.02*(1.23)*
parse3 *20	2.13	0.84*(2.54)*	0.80*(2.66)*	0.83*(2.57)*	1.72*(1.24)*
farmer *100	4.83	2.34*(2.06)*	2.41*(2.00)*	2.49*(1.94)*	3.80*(1.27)*
AVERAGE		*(2.46)*	*(2.58)*	*(2.46)*	*(1.25)*

Table 1: RUN TIMES, FIRST GENERATION OF AURORA

11 Conclusions and Future Work

We have described the engine-scheduler interface used in the second generation of the Aurora or-parallel Prolog system. We have defined a simple set of functions to cover the two basic areas of engine-scheduler interaction: finding work and communication between workers. We have identified those events during Prolog execution that may be of potential interest to schedulers, e.g. creation of nodes, entering clauses, etc. We have also developed a general characterisation of pruning properties of Prolog clauses that can be used both for scheduling speculative work and for improving the implementation of pruning operators.

The interface described in this paper is fundamentally revised with respect to earlier versions. The new interface is designed to help avoid scheduling overheads, to make the set of basic concepts simpler and more uniform, to give scope for potential optimisations including better memory management, improved treatment of pruning operations, and avoidance of speculative work.

The main purpose of the interface is to enable different engines and schedulers to be used interchangeably. To date, four separate schedulers have been written and connected to two different engines by means of the interface. Perhaps more importantly, the evolution of the interface has helped clarify many issues in implementing or-parallelism in Prolog, such as contraction and handling of pruning information.

The interface has contributed to the overall improvement of Aurora performance. We also believe that the new interface has played a significant part in the good performance results of the Bristol

Goals * repetitions	Aurora Workers 1	4	8	11	Sicstus 0.6
8-queens1	8.01	2.03*(3.95)*	1.03*(7.75)*	0.76*(10.6)*	6.77*(1.18)*
8-queens2	20.63	5.25*(3.93)*	2.64*(7.81)*	1.93*(10.7)*	16.45*(1.25)*
tina	18.40	4.65*(3.96)*	2.39*(7.69)*	1.79*(10.3)*	13.78*(1.34)*
salt-mustard	10.89	2.82*(3.86)*	1.48*(7.36)*	1.11*(9.86)*	8.85*(1.23)*
AVERAGE		*(3.92)*	*(7.65)*	*(10.4)*	*(1.25)*
parse2 *20	7.16	2.40*(2.99)*	1.71*(4.18)*	1.64*(4.37)*	5.87*(1.22)*
parse4 *5	6.67	1.85*(3.60)*	1.40*(4.76)*	1.19*(5.60)*	5.40*(1.24)*
parse5	4.71	1.42*(3.33)*	0.96*(4.89)*	0.81*(5.81)*	3.82*(1.23)*
db4 *10	2.94	0.81*(3.63)*	0.46*(6.39)*	0.38*(7.82)*	2.24*(1.31)*
db5 *10	3.56	0.97*(3.67)*	0.57*(6.25)*	0.47*(7.64)*	2.73*(1.30)*
house *20	5.07	1.47*(3.46)*	0.93*(5.48)*	0.79*(6.42)*	4.22*(1.20)*
AVERAGE		*(3.45)*	*(5.32)*	*(6.28)*	*(1.25)*
parse1 *20	1.89	0.76*(2.47)*	0.73*(2.61)*	0.78*(2.42)*	1.57*(1.20)*
parse3 *20	1.62	0.72*(2.24)*	0.68*(2.37)*	0.72*(2.25)*	1.34*(1.21)*
farmer *100	3.61	1.92*(1.88)*	2.13*(1.69)*	2.19*(1.65)*	3.06*(1.18)*
AVERAGE		*(2.20)*	*(2.22)*	*(2.11)*	*(1.20)*

Table 2: RUN TIMES, SECOND GENERATION OF AURORA

scheduler. The Bristol scheduler has been designed with the new interface in mind, and, in spite of applying a very simple scheduling strategy, its performance is comparable (and sometimes better than) that of the earlier schedulers [3].

The main outstanding issue which has not been treated in the interface is garbage collection. Patrick Weemeeuw [17] has addressed the problem of garbage collection of the public parts of the tree. Since such activities involve synchronisation between workers and possibly relocation of scheduler data, it is likely that the interface will have to be extended to support garbage collection.

The interface has recently been utilised in a project based on the Muse approach to or-parallel Prolog [1]. An or-parallel version of BIM_Prolog [4] is currently being produced by modifying the BIM engine and connecting it via the interface to the Muse scheduler.

We are convinced that the applicability of the interface extends beyond or-parallel Prolog systems. The Andorra experience is powerful evidence of this fact, but it must be stressed that in this case, the interface was used to add or-parallelism to an already and-parallel system. Generalising the interface to cover issues of and-or-parallel scheduling could be an interesting research direction to be pursued in the future.

12 Acknowledgements

The work on engine-scheduler interfaces was initiated by David Warren. Earlier versions of the interface were developed by Ross Overbeek and Alan Calderwood. The design of the new interface benefited from several discussions with Tony Beaumont, Per Brand, Bogumił Hausman and Ewing Lusk.

The authors are indebted to Feliks Kluźniak, Ewing Lusk, and the anonymous referees for careful reading and valuable comments on drafts of this paper.

This work was supported by ESPRIT projects 2471 ("PEPMA") and 2025 ("EDS").

References

[1] Khayri A. M. Ali and Roland Karlsson. The Muse approach to or-parallel Prolog. *International*

Journal of Parallel Programming, 19(2):129–162, April 1990.

[2] Anthony Beaumont, S. Muthu Raman, Vítor Santos Costa, Péter Szeredi, David H. D. Warren, and Rong Yang. Andorra-I: An implementation of the Basic Andorra Model. Technical Report TR-90-21, University of Bristol, Computer Science Department, September 1990. Presented at the Workshop on Parallel Implementation of Languages for Symbolic Computation, July 1990, University of Oregon.

[3] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Flexible scheduling or-parallelism in Aurora: the Bristol scheduler. In *PARLE 91, Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, June 1991.

[4] BIM. BIM_Prolog release 2.4. 3078 Everberg, Belgium, March 1989.

[5] Per Brand. Wavefront scheduling. Internal Report, Gigalips Project, 1988.

[6] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens. Scheduling OR-parallelism: an Argonne perspective. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1590–1605. MIT Press, August 1988.

[7] Alan Calderwood. Aurora—description of scheduler interfaces. Internal Report, Gigalips Project, January 1988.

[8] Alan Calderwood and Péter Szeredi. Scheduling or-parallelism in Aurora—the Manchester scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435. MIT Press, June 1989.

[9] Mats Carlsson and Péter Szeredi. The Aurora abstract machine and its emulator. SICS Research Report R90005, Swedish Institute of Computer Science, 1990.

[10] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG*. PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[11] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[12] Vítor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I: A parallel Prolog system that transparently exploits both and- and or-parallelism. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, April 1991.

[13] Raed Sindaha. Scheduling speculative work in the Aurora or-parallel Prolog system. Internal Report, Gigalips Project, March 1990.

[14] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

[15] Péter Szeredi and Mats Carlsson. The engine-scheduler interface in the Aurora or-parallel Prolog system. Technical Report TR-90-09, University of Bristol, Computer Science Department, April 1990.

[16] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

[17] Patrick Weemeeuw. Memory compaction for shared memory multiprocessors, design and specification. In *Proceedings of the North American Conference on Logic Programming*. MIT Press, October 1990.

Reduction of Code Space in Parallel Logic Programming Systems *

HWANG Zhiyi, HU Shouren, SUN Chengzheng and GAO Yaoqing

Department of Computer Science

Changsha Institute of Technology

Changsha, Hunan, P.R. China

Keywords: logic programming, parallel processing, compile, AND-parallelism, abstract interpretation, mode inference, PROLOG, computer languages.

Abstract

This paper presents the problem of the explosion of code space in methods for exploiting independent AND-parallelism in parallel logic programming systems, and proposes a scheme to solve it. The scheme can largely reduce the code space with minimal loss of parallelism. Some key algorithms in the scheme are presented in this paper. Finally, we give the experimental results and the preliminary performance analysis on the scheme.

1. Introduction

There have been many contributions to the exploitation of independent AND-parallelism in logic programs [3,4,5,6,7,8,9,14]. These publications can be divided into three approaches: dynamic, static and static-dynamic combined. Because of the higher overhead of

---

* This research was partially supported by the Fok Ying Tung Education Foundation and the Chinese National Natural Science Foundation.

the dynamic and the lower parallelism exploited by the static, the static-dynamic approaches have become the most promising and interesting ones to which a great deal of attentions have been dedicated [4,5,8,9,10,14,15]. With some run-time checks these approaches can exploit almost the maximum independent AND-parallelism. However, the deeper they go into the exploitation of parallelism and the more accurately they exploit parallelism, the more Execution Graph Expressions(EGEs) these approaches will generate, which will lead to an enormous code space. The reason is that, in order to minimize the loss of parallelism when generating a linear EGE, the best method is to generate a linear EGE for each possible execution order of subgoals in the clause body. For instance, [10] uses the expressions such as Cond -> D1;D2 to generate a linear expression for each possible condition combination. But with the exponential explosion of condition combinations, the increase in code space is intolerable. For example, consider the clause:

H(A,B,C,D,E):-P(A,B),Q(B,C),R(C,D),S(D,E),T(E,A).

According to the CDG algorithm in [10], the number of expressions generated for this clause is at least 81 without optimization. In other words, the code space is nearly 81 times the size of the sequential one.

The same problem exists in CAAP(Compiling Approach for exploiting AND_Parallelism in logic programs)[14]. CAAP consists of three phases: analysis of entry modes, derivation of exit modes and determination of EGEs. From the experimental results, this approach can exploit independent AND-parallelism effectively with trivial run-time overhead. In the approach, we generate an EGE for each possible entry mode of a clause. Therefore, the extended parallel WAM code space [11] compiled from CAAP's EGEs expands largely and is two to six times the sequential WAM code space, according to the experiment.

With the entry modes' increasing exponentially, the code space will become larger and larger. To contain this trend, we adopt a special technique for automatic inference of entry modes. This technique can infer one or two entry modes that users often use. We refer to them as User-Usually-Used entry modes (U-modes in short). For the other possibly used entry modes, we abstract a Default Entry Mode (DEM) from them. This mode can represent those possible modes. We use an algorithm to generate a default EGE for it. When a subgoal calls a clause with one of the possible entry modes (excluding the U-modes), the default EGE will be invoked and executed. In this way, we can ensure the seldom used modes will be dealt with correctly without much loss of parallelism while we can accurately exploit AND-parallelism in most cases. Above all, the idea can largely reduce the code space which will be constantly kept about two times the sequential WAM code space.

Based on the above ideas, we propose a scheme in the following section, which can overcome the drawbacks of CAAP scheme. We also give the experimental results to show the performance of the schemes.

2. Scheme of code space reduction

We proposed the concepts of entry mode and exit mode for the first time in CAAP [14]. An entry mode of a clause explains whether the arguments of the clause head are ground after a subgoal invokes it and unifies with its head; an exit mode of a clause explains whether the arguments of the clause head are ground after the subgoals of its body are solved. An entry mode of a subgoal explains whether its arguments are ground before it is about to be executed; an exit mode of a subgoal explains whether its arguments are ground after it is solved. There are three possible modes for an argument: G, means the corresponding argument is ground; N, means the argument is non-ground;

?, means that whether the argument is ground or not is unknown.

For a clause, its different entry modes will lead to different dependence relationships among subgoals in its body and different AND-parallelism in it. Therefore, we generate an EGE for every entry mode of the clause so that the maximal AND-parallelism may be obtained. To analyze the data-dependence among subgoals, we used bottom-up abstract interpretation [12] in CAAP. The abstract domain is {G,N}. We view a logic program as a directed graph in which each node represents a procedure and each node has pointers to the nodes it calls. There is a cycle if a procedure calls itself. A root node is a procedure that is not called by any other procedure except the top level query, and a terminal node is a procedure that does not call any other procedure except itself. For instance, the program in Example 2.1 can be viewed as the directed graph in Figure 2.1.

Example 2.1 : Quicksort program

```
quicksort([_h|_t],_s):-
                split(_h,_t,_x,_y),
                quicksort(_x,_x1),
                quicksort(_y,_y1),
                append(_x1,[_h|_y1],_s).
quicksort([],[]).

split(_h,[_e|_t],[_e|_x],_y):-
                _e < _h,
                split(_h,_t,_x,_y).
split(_h,[_e|_t],_x,[_e|_y]):-
                _e >= _h,
                split(_h,_t,_x,_y).
split(_h,[],[],[]).

append([],_l,_l).
append([_h|_t],_l,[_h|_l1]):-
                append(_t,_l,_l1).

?- quicksort([5,3,2,4,1,8,6,7,9,10],_l).
```

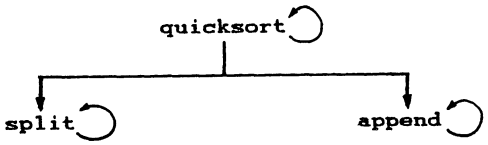


Fig.2.1 Directed graph of quicksort program

In the scheme of code space reduction, we inherit some idea of CAAP. We describe it in three phases as in CAAP: inference of entry mode; derivation of exit mode; and determination of EGE. The major difference between the CAAP and the scheme is that we generate a EGE for every correct (possible) entry mode in the former, but generate EGEs only for U-modes and DEMs in the latter. Therefore, the EGEs generated for a clause are largely reduced.

2.1 Inference of entry mode

A good deal of work has been done on automatic mode inference in recent years. The typical ones are [1,2,8]. Even though their techniques are similar, their purposes differ from each other to some extent. These techniques may be used to optimize the sequential logic programs [1,2] or to exploit parallelism in logic programs [8]. In this section we propose a technique to infer the entry modes users often use, so that the expense of code space in CAAP and other methods for exploiting independent AND-parallelism can be greatly reduced. In this paper, the mode inference with abstract interpretation is just a useful tool in our code space reduction scheme, but it is not the main idea of the paper. What we are interested in is the result of the abstract interpretation.

This technique is divided into two sub-phases: automatic inference of correct(possible) entry modes and automatic inference of U-modes. A correct entry mode is an arbitrary entry mode which may possibly succeed when it calls its procedure. A U-mode is definitely a correct entry mode, but a correct entry mode may not be a U-mode.

In the first sub-phase, we adopt the bottom-up abstract interpretation, i.e., we analyze the procedures in the directed graph from bottom up until the root node is reached. The abstract domain is {G,N}. In the sub-phase, the mode information of built-in predicates is used to infer the correct entry modes of user defined procedures,

e.g., built-ins <, > require its arguments are ground, i.e., their entry modes are G. For example, consider the clause in Example 2.1:

split(_h,[_e|_t],[_e|_x],_y):-_e < _h, split(_h,_t,_x,_y).

From the requirement of built-in '<', we can straightforwordly conclude the first argument's entry mode of split must be G. This mode requirement of split can also be used by its top procedures, e.g., quicksort. In this way, the correct entry modes of every procedure in the logic program will be obtained. The key algorithm in this sub-phase is given as below.

Algorithm I:

Input: a clause C:C+ :- C- and its entry mode

Output: an exit mode

Vars: GV: current set of ground variables

i: pointer to subgoals in the clause body

S: current subgoal

SG: sets of ground variables of S after S is executed

1. initialization.

i := 1, GV := {variables in the clause head's arguments whose entry modes are G}.

2. if C- = [], then C is a fact, go to 6.

3. set S to the ith subgoal of C-.

4. if S = nil, go to 6.

5. do the following steps for S.

i) determine the entry mode of S according to GV.

ii) check if the entry mode of S matches the correct entry modes of S. If it does not, the entry mode of C is impossible or incorrect, stop.

iii) if S invokes C recursively,

then, derive the exit mode of S according to the entry mode of S and the exit mode of facts in the

recursive procedure.

else, derive the exit mode of S according to the entry mode of S and the exit mode of procedure S.

iv) evaluate SG according to the exit mode of S.

v) set GV to GV U SG.

vii) i := i + 1, go to 3.

6. derive the exit mode of C according to GV. Stop.

This algorithm is used to test whether an entry mode is correct or not. We use it to test every combination of entry modes and then get the correct entry modes and their respective exit modes for every clause. To get the correct entry modes of a procedure, we employ two operations. For a recursive procedure, we apply intersection among correct entry modes of clauses in the procedure. For a non-recursive procedure, we apply union among correct entry modes of clauses in the procedure. The reason is that we regard a recursive procedure as a whole body and its entry modes should satisfy the requirement of each of its clauses.

Furthermore, in the same procedure, clauses may have different exit modes for the same entry mode. We will apply "*" operation to the different exit modes. The operation table is as Table 2.1. From Table 2.1, we can see the rule adopted is safe for the "producer-consumer" scheme. That means if some argument of a procedure is known, with our approach, to have entry mode N and corresponding exit mode G, then the procedure is certain to be the real "producer" of variables in the argument by the "*" operation.

*	G	N
G	G	N
N	N	N

Table 2.1 An operation table

As for the program in Example 2.1, adopting above method, we can obtain the correct entry/exit mode tables in Table 2.2 to Table 2.4.

entry mode	exit mode
<G,G>	<G,G>
<G,N>	<G,G>

Table 2.2 Entry/exit mode table for "quicksort"

entry mode	exit mode
<G,G,N,N>	<G,G,G,G>
<G,G,G,N>	<G,G,G,G>
<G,G,N,G>	<G,G,G,G>
<G,G,G,G>	<G,G,G,G>
<G,N,G,G>	<G,G,G,G>

Table 2.4 Entry/exit mode table for "split"

entry mode	exit mode
<N,N,N>	<N,N,N>
<N,N,G>	<G,G,G>
<N,G,N>	<N,G,N>
<N,G,G>	<G,G,G>
<G,N,N>	<G,N,N>
<G,N,G>	<G,G,G>
<G,G,N>	<G,G,G>
<G,G,G>	<G,G,G>

Table 2.3 Entry/exit mode table for "append"

In the second sub-phase, we adopt the top-down abstract interpretation, i.e., we analyze the procedures in the directed graph from top down until every leaf node is reached. The abstract domain is {G,N,?}. At first, we regard the correct entry mode as the U-mode of the top procedure and use it to infer the entry modes of procedures the top procedure invokes. We then regard these entry modes as their U-modes and use them to continue above process. In this way, we can infer the U-mode of every procedure in the program. The reason that we regard these types of entry modes as U-modes is that that how the top procedure invokes the procedures below reflect the semantic information of the program and the tendency that users invoke the procedures. For example, we use the correct entry mode of quicksort <G?> to infer the entry modes of split and append. Their respective

entry modes are <GGNN> and <GG?>. ? may represent G or N. The key algorithm in the sub-phase is presented in the following.

Algorithm II:

Input: a clause C:C+ :- C- and its U-mode

Output: 3U entry modes of subgoals in the body

Vars: GV: current set of ground variables

NV: current set of non-ground variables

VS: current set of variables occurring in the clause

S: current subgoal

SG, SN: sets of ground and non-ground variables of S after S is executed, respectively

SV: variable set of S

i: a pointer

1. initialization.

i := 1,

GV := {variables in the clause head's arguments whose entry modes are G}.

NV := {variables in the clause head's arguments whose entry modes are N}.

VS := {variables in the clause head}.

2. set S to the ith subgoal of C-.

3. if S = nil, stop.

4. the following steps are done for S.

i) determine the entry mode of S according to GV, NV and VS.

ii) record the entry mode as a U-mode of S.

iii)find the exit mode of the entry mode according to the correct entry/exit mode table.

iv) evaluate SG,SN and SV according to the exit mode of S.

v) GV := GV U SG,

NV := NV U SN,

VS := VS U SV.

vi) i := i + 1, go to 2.

Adopting Algorithm II, we can obtain the U-modes of Example 2.1 as below.

```
U-mode(quicksort(2),[GN,GG]).
U-mode(split(4),[GGNN]).
U-mode(append(3),[GGN,GGG]).
```

We have employed above entry mode inference technique to improve the CAAP scheme. In the improved scheme, we only generate an EGE for every U-mode while we generate a default EGE for other possible entry modes. In this way, we can largely reduce the code space without much loss of parallelism. The experimental results in Section 3 can prove what we expect.

For other correct entry modes, we can infer a DEM to represent them. The DEMs for Quicksort program inferred by [15] are as below:

```
DEM(split(4),[G???]).
DEM(append(3),[???]).
```

'?' means the entry mode for the corresponding argument is may be either 'G'(ground) or 'N'(non-ground). It is not necessary to infer a DEM for quicksort(2) since it has no correct entry mode left apart from its U-modes.

2.2 Derivation of exit modes

The function in this phase is the same as the one in CAAP. The exit modes can be acquired straitforwardly from the entry/exit mode tables in Section 2.1 after using a searching algorithm. We simply give the entry/exit mode tables of the Quicksort program as below:

```
modetable(quicksort(2), [(GN,GG),(GG,GG)]).
modetable(split(4),[(GGNN,GGGG),(G???,GGGG)]).
modetable(append(3),[(GGN,GGG),(GGG,GGG),(???,???)]).
```

2.3 Determination of EGEs

The algorithm for determining EGEs of U-modes remains the same as the one presented in [14]. But for the EGEs of DEMs, we employ the following algorithm.

We have some principles to design the algorithm. Firstly, the EGEs generated should be simple, and there should be no nested condition checks in the EGEs so that the parallel WAM code compiled out from them is more efficient. Secondly, the overhead of the algorithm should be small. Thirdly, the EGEs should not change the original order of the subgoals so as to facilitate the handle of side-effects problem.

The idea of the algorithm is that we divide a clause body into some maximal sections, which satisfy the requirement that inside a section no two subgoals share any variable that occurs in the clause for the first time. Therefore we should execute the sections sequentially, and we add condition checks inside a section to exploit parallelism.

Algorithm for determining a clause's EGE of DEM

Input: a clause H:-B1,...,Bm and its DEM.

Output: EGE of the clause.

Vars: G, set of ground variables determined with DEM.
V, set of variables occurring in the previous sections.
N, set of variables, which occur in the clause for the first time, in the current section.
B, the remaining subgoals in the clause.
i,j, pointers.

1. initialization.
   G := {variables in the arguments of H, whose entry modes are 'G' according to DEM}
   V := {variables occurring in H}

N := ∅

B := B1,...,Bm.

i := 1, j := i

2. for the subgoal Bj, compute FV(Bj) := V(Bj) - V, in which V(Bj) is the set of variables occurring in Bj, and therefore FV(Bj) is the set of variables, which do not occur in the previous sections, in Bj.

3. if FV(Bj) ∩ N <> ∅ then goto 4

N := N ∪ FV(Bj),

j := j + 1,

if j <= m then goto 2

4. regard Bi,...,Bj-1 as one section, compute

S := ∪ V(Bp) ∩ V(Bq) (i =< p,q < j,p <> q)

S is the set of variables which occur in both Bp and Bq, i =< p, q < j, p <> q. The EGE for the section is as below:

GPAR(S)IPAR(V(Bi),...,V(Bj-1))(Bi,...,Bj-1).

Because there is no need to check groundness of variables in G and independence between variables in S and N, the EGE may be optimized as

GPAR(S-G)IPAR(V(Bi)-S-N,...,V(Bj-1)-S-N)(B1,...,Bj-1).

Then, update the Vars:

V := V U N,

N := ∅,

i := j,

B := Bj,...,Bm.

5. if B <> nil, goto 2.

6. use SEQ expression to contain above EGEs, which is the final EGE for the clause. Stop.

For example, we have a clause p(X,Y,Z):-a(X,T), b(X,V), c(T,W),

d(Y,R), e(W,Z), f(R,Z). The DEM of the clause is '???'. Using the algorithm we can generate the following EGE:

```
p(X,Y,Z):-
    SEQ(GPAR(X)(a(X,T),b(X,V)),
        IPAR(T,Y)(c(T,W),d(Y,R)),
        GPAR(Z)IPAR(W,R)(e(W,Z),f(R,Z))).
```

Now we give the EGEs of Quicksort program in the Improved CAAP scheme.

```
quicksort(GN,[_h|_t],_s):-
        SEQ(split(GGNN,_h,_t,_x,_y),
            PAR(quicksort(GN,_x,_x1),
                quicksort(GN,_y,_y1)),
            append(GGN,_x1,[_h|_y1],_s)).
quicksort(GG,[_h|_t],_s):-
        SEQ(split(GGNN,_h,_t,_x,_y),
            PAR(quicksort(GN,_x,_x1),
                quicksort(GN,_y,_y1)),
            append(GGG,_x1,[_h|_y1],_s)).

split(GGNN,_h,[_e|_t],[_e|_x],_y):-
            PAR((GG,_e < _h),
                split(GGNN,_h,_t,_x,_y)).
split(G???,_h,[_e|_t],[_e|_x],_y):-
            PAR((GG,_e < _h),
                split(G???,_h,_t,_x,_y))

split(GGNN,_h,[_e|_t],_x,[_e|_y]):-
            PAR((GG,_e >= _h),
                split(GGNN,_h,_t,_x,_y)).
split(G???,_h,[_e|_t],_x,[_e|_y]):-
            PAR((GG,_e >= _h),
                split(G???,_h,_t,_x,_y)).

append(GGN,[_h|_t],_l,[_h|_l1]):-
        append(GGN,_t,_l,_l1).
append(GGG,[_h|_t],_l,[_h|_l1]):-
        append(GGG,_t,_l,_l1).
append(???,[_h|_t],_l,[_h|_l1]):-
        append(???,_t,_l,_l1).
```

Above EGEs are more optimized forms than the algorithm generates. From the result, we know the EGEs generated are significantly less than those in CAAP.

3. Experimental results and performance analysis

Based on the improved scheme, we have implemented a precompiler in SES-PIM system [13]. The comparisons among CAAP, the above scheme(called Improved CAAP) and the dynamic approach [17], which can

detect the maximum AND-parallelism at run-time, are given in Table 3.1, 3.2 and 3.3 after running some typical benchmarks, such as quicksort, n-queen problem, matrix multiplication, Hanoi tower, maze problem, factorial, in SES-PIM system.

From Table 3.1 we know there is no loss of parallelism in the Improved CAAP. By the degree of parallelism, we mean the number of processes in one inference step of logic program. The degree of parallelism in Table 3.1 implies the average parallelism.

From Table 3.2 we know the run-time checks do not increase in the Improved CAAP.

	ICAAP	CAAP	the dynamic
quicksort	3.82	3.82	3.82
nqueen	19.21	19.21	19.21
matrix multiplication	6.08	6.08	6.08
Hanoi tower	5.21	5.21	5.21
maze problem	10.59	10.59	10.59
factorial	6.14	6.14	6.14
on average	16	16	16

Table 3.1 Degree of parallelism exploited in CAAP, the Improved CAAP (ICAAP) and the dynamic

In Table 3.3, the first column is the number of clauses (excluding facts) in the corresponding program. The second column is the number of EGEs generated by CAAP. The third column is the number of EGEs generated by the Improved CAAP. Even though we do not compile the EGEs into parallel WAM code (the code-size of a compiled EGE is nearly the same as that of an optimized WAM-compiled clause), we can give an estimation of the code space from Table 3.3: the code space in CAAP is nearly four times the sequential WAM code space while the code space in the improved CAAP is approximately two times the sequential one, on

		CAAP	ICAAP
quicksort	IPAR	0	0
	GPAR	0	0
nqueen	IPAR	0	0
	GPAR	0	0
matrix multiplica-tion	IPAR	12	12
	GPAR	0	0
Hanoi tower	IPAR	0	0
	GPAR	0	0
maze problem	IPAR	0	0
	GPAR	0	0
factorial	IPAR	0	0
	GPAR	0	0
in total	IPAR	12	12
	GPAR	0	0

Table 3.2 Number of checks used during execution

	# clauses	# EGEs	# EGEs/redu.
quicksort	4	20	9
nqueen	5	14	8
matrix multiplication	6	18	12
Hanoi tower	3	17	5
maze problem	4	24	16
factorial	2	4	4
in total	24	97	54

Table 3.3 Number of EGEs generated in CAAP and the Improved CAAP

an average. The result is satisfying. Most importantly, we can predict that the code space in the Improved CAAP is constantly about two times the sequential code space with trivial loss of parallelism.

The loss of parallelism in the Improved CAAP may occur when a default EGE of a clause has loss of parallelism according to [10] and the clause is invoked with the DEM during execution. The number of run-time checks may also increase if the EGE of a DEM is invoked. But if the precise U-modes are inferred the loss of parallelism and the increase of checks are trivial. From the experiment, we know our U-mode inference technique is very effective and the U-modes are very accurate.

Summary

In this paper, we present the problem of the explosion of code space in CAAP and other methods for exploiting independent AND-parallelism, and propose an Improved CAAP scheme to solve it. The scheme can largely reduce the code space with minimal loss of parallelism. The idea of this paper can also be useful to Hermenegildo's CDG algorithm. Only the linear expressions corresponding to conditions with high probability of success are generated while the rest are contained in one expression with a default condition. In this way, the code space generated by CDG algorithm can also be reduced effectively.

Acknowledgements

We would like to thank Dr. Doug DeGroot for many helpful comments and suggestions on this work. We would also thank Cao Peng and Shan Rui for valuable discussions.

References

[1] C.S. Melish, The automatic generation of mode declarations for Prolog programs, DAI Research paper 163, Dept. of Artificial Intelligence, Univ. of Edinburgh (August 1981), U.K.

[2] S.K. Debray and D.S. Warren, Automatic Mode Inference for Prolog Programs, Journal of Logic Programming, 207-229,Sept. 1988.

[3] J.S. Conery, The AND/OR model for parallel interpretation of logic programs, Ph.D. Th., Dept. of Infor. and Computer Sci., Univ. of California, Irvine, 1983.

[4] D. DeGroot, Restricted And-parallelism, Proc. of the Int'l Conf. on Fifth Generation Computer System, Tokyo, (Nov. 1984) pp. 471-478.

[5] D. DeGroot, A technique for compiling execution graph expressions for restricted And-parallelism in logic programs, Proc. of the 1987 Int'l Supercomputing Conf., Athens, Greece, (June 1987).

[6] J.-H. Chang, A. M. Despain and D. DeGroot, And-parallelism of logic programs based on a static data dependency analysis, COMPCON 85, San Francisco, Feb., 1985, pp. 218-225.

[7] Yu-Wen Tung and Dan I. Moldovan, Detection of And-parallelism in logic programming, Proc. of the 1986 Int'l Conf. on Parallel Processing, IEEE, Pennsylvania, 1986, pp. 984-991.

[8] H. Xia and W.K. Giloi, A Hybrid Scheme for Detecting AND-Parallelism in Prolog Programs, Proc. of ACM 1988 Int. Conf. on Supercomputing, France, July, 1988.

[9] K. Muthukumar and M. Hermenegildo, Methods for Automatic Compile-time Parallelization of Logic Programs using Independent/Restricted And-parallelism, Technical Report ACA-ST-233-89, MCC, Austin, TX 78759, March 1989.

[10] K. Muthukumar and M. Hermenegildo, The CDG, UDG and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism, Technical Report ACA-ST-023-90, MCC, Austin, TX 78759, 1990.

[11] Gao Yaoqing and Hu Shouren, A RAP/LOP-WAM Abstract Instruction Set, Technical Report, Dept. of Computer Sci., Changsha Inst. of Tech., 1989.

[12] C.S. Melish, Abstract Interpretation of Prolog Programs, In 3rd International Conf. on Logic Programming, pp 463-475, Imperial College, Springer-Verlag, July 1986.

[13] Sun Chengzheng and Ci Yungui, SES-PIM: a simulation and experiment system for PIM-PSOF, the 2nd National Conf. on Logic Programming, China, 1986.

[14] Hwang Zhiyi and Hu Shouren, A compiling approach for exploiting And-parallelism in parallel logic programming systems, Proc. of Parallel Architecture and Language Europe, Netherlands, 1989.

[15] Hwang Zhiyi and Hu Shouren, An Improved CAAP Scheme, Technical Report, Dept. of Computer Sci., Changsha Inst. of Tech., 1989.

[16] Hwang Zhiyi and Hu Shouren, Compilation techniques of parallel inference machines, Journal of Computer Science, 1990(3), China.

[17] Sun Chengzheng and Ci Yungui, An automatic partition algorithm for And-parallel execution in the framework of OR-forest, Proc. of the 2nd Int'l Conf. on Computers and Applications, Beijing, 1987.

Search Level Parallel Processing of Production Systems

Satoshi Fujita, Masafumi Yamashita, and Tadashi Ae
Faculty of Engineering, Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 724 Japan
Tel. +81-824-22-7111 (Ext. 3459), Fax. +81-824-22-7195
Email: ae@csl.hiroshima-u.ac.jp

Abstract

This paper examines parallel matching for PS's. The matching operation, which consists of the template matching and the join of matched data sets, is believed to be the most time consuming operation of PS's. Thus to realize a real-time PS, we should clarify the limitation on the speedup by the parallel processing and should find a mechanism to support it effectively.

The matching operation includes two level parallelisms, say, the constraint satisfaction level and the DB search level. Many conventional approaches have focused on the former so far, however, the parallelism is not so large in existent PS programs [4]. Hence we should focus on the another level parallelism as well to achieve a limitation on the speedup. The search level parallelism can be effectively supported by two kinds of parallel processing schemes based on (1) shared bus connected multiprocessor, and (2) multiprocessor with special memory devices. In this paper, we will provide the concrete specification of the schemes and verify the availability using few benchmark programs.

1 Introduction

A production system (PS) is one of execution models for declarative programming languages. Application programs on the PS model (PS programs) are composed of three parts: the database (DB), the rulebase (RB), and the conflict resolution strategy (CRS). DB is a collection of facts, and a datum in DB is a tuple of name and attributes. RB is a collection of rules. Each rule is a pair of condition part and action part which implies if the condition part holds, then the action part can be executed. Action parts of rules contain DB manipulate instructions. CRS determines the direction of the computation.

The computation of the PS model is a repetition of

1. the matching phase to search rule instances whose condition parts hold,
2. the conflict resolution phase to select the most adequate instance from the instances found in the matching phase, and
3. the action phase to execute the action part of the instance selected in the conflict resolution phase.

Since the computation time is generally large and is dominated by the matching phase, it is an essential problem for the speedup of PS's to find a mechanism for the quick matching phase execution.

Many researchers initially considered matching algorithms on conventional machines. Forgy proposed a matching algorithm Rete [2] which is an extension of McDermott's filter [8]. Rete is designed focusing on the following two features of the matching operation, i.e., (1) locality of the DB modification, and (2) similarity among condition parts of rules. For the first feature, he proposed to save the state of the matching phase besides DB, which consists of result of the template matching (i.e., constant test), that of the join operation (i.e., variable test) and the final results (i.e., instances of rules). For the second feature, on Rete, sharing of instructions among analogous matching sequences is examined. This idea has been extended by many researchers after Rete [11][6].

Most of above minor changes of Rete, however, do not contribute to the drastic speedups (i.e., they have an obvious limitation[1]) because they have to be executed sequentially. Hence to obtain further speedup on the computation of PS's, we should apply a sort of parallel processing to the operation.

PSM project at Carnegie-Mellon University has examined the parallel execution of Rete algorithm. They directly implemented Rete on a commercial multiprocessor system Encore Multimax, and achieved 11 folds speedup using 16 processors [5]. PESA-1 [12] and MANJI [9] are classified in this category. For general expert systems with 200-3,000 rules, these architectures work so effectively, since they support only the higher level parallelism, and in general, the magnitude of the parallelism is not too large [4]. Alternative idea applied so far is to extract the search level parallelism using significantly large number of processing elements. Tree-structured massively parallel processors DADO [14] and NON-VON [13] are classified in this category. Although they can perform several DB queries quickly, the effect is not drastic because of the fine granularity of the operation.

Our target in this paper is to clarify the limitation on the speedup of the matching operation, and to provide parallel processing schemes to support it effectively. To achieve the first objective, we should review the nature of PS programs in detail. The parallelism of the matching operation is classified into following two levels : the constraint satisfaction level (CE level) and the DB search level (search level). The former was supported by PSM and the latter was supported by DADO, respectively. In the following chapters, we will show the importance of the latter parallelism to achieve the limitation. For the second objective, we will propose two parallel processing schemes to support the search level parallelism. To implement them effectively, we assume two kinds of hardwares, i.e., (a) a shared bus connected multiprocessor, and (b) multiprocessor with special purpose memory devices. The availability of the proposed schemes will be estimated using benchmark programs.

[1]In fact, even if we use 5 MIPS 32 bit microprocessor, each DB modification takes few hundred nano second as long as it applies sequential processing [4]

2 State-Saving Matching Algorithm MASS

This chapter introduces a simple state-saving matching algorithm MASS (Matching Algorithm using State-Saving method) for PS to clarify the discussion in later chapters.

Previous to the concrete description of MASS, we should define the matching operation more precisely. It is essentially a pattern matching among patterns (i.e., CE's of rules) and objects (i.e., data in DB). Each datum has a form $(C_i, attr_1, attr_2, ..., attr_k)$ where C_i is the name and each attribute is a pair of attribute name and attribute value. On the other hand, each CE of rules has a form $(C_i, field_1, field_2, ..., field_h)$ where C_i is the *name* and each field is a *predicate* on the attribute. Remember that condition part of a rule is a conjunction of several CE's where any data in DB are instances of CE's.

As an example, consider the following rule consisting of three CE's :

$$
\begin{aligned}
rule\ r1: \ \text{if} \quad & (station, name =< x >, flag = 1) \cdots CE_1 \\
\& \quad & (adjacent, pre =< x >, suc =< y >) \cdots CE_2 \\
\& \quad & (station, name =< y >, flag = 0) \cdots CE_3 \\
\text{then} \ remove \quad & (station, name =< y >, flag = 0) \\
make \quad & (station, name =< y >, flag = 1).
\end{aligned}
$$

Rule r1 is a rule to find stations reachable from one station by train, which means that "if there is a reachable station, then the adjacent station is also reachable." Reachable stations are marked by "flag" attribute (stations with flag=1 are reachable and those with flag=0 have not yet been examined the reachability). DB is referred to find reachable but not marked station, which will be marked in the action part.

On the PS model introduced in Chapter 1, condition part of a rule may contain several variables in the predicate (e.g., r1 contains two variables $< x >$ and $< y >$). Furthermore, they may be bound among CE's so that to have the same value (e.g., in r1, $< x >$ is bound between first attribute of CE_1 and first attribute of CE_2). Thus for example, a tuple of data ((station, name=tokyo, flag=1), (adjacent, pre=tokyo, suc=kanda), (station, name=kanda, flag=0)) satisfies the constraint on $< x >$ and $< y >$, while ((station, name=tokyo, flag=1), (adjacent, pre=tokyo, suc=kanda), (station, name=hiroshima, flag=0)) does not satisfy it since $< y >$ takes different value in CE_2 (=kanda) and in CE_3 (=hiroshima). Precisely speaking, a given condition part of a rule and a tuple of data are said to be matched when both of following two conditions are held :

1. For a given datum (in DB) and the corresponding CE, the datum has to be an instance of the CE (intra CE condition), and

2. for given instances of CE_1 and CE_2, they have to satisfy the inter CE's constraint between CE_1 and CE_2 specified by bound variables. □

The target of MASS and other state-saving matching algorithms is to check above conditions effectively by saving the state of the matching operation. Intuitively, when the algorithm is applied, the inspection of whole DB and the check for all rules can be replaced by the check for the data newly added or deleted in the last action phase and the inspection of clusters related to the data.

Figure 1 shows the outline of a concrete state-saving matching algorithm MASS. Algorithm MASS is composed of following three parts: (1) CT (Constant Test) processing (steps 1-7) (2) VT (Variable Test) processing (steps 8-19) (3) Generation of new conflict set (steps 20-24).

3 Models for Parallel Matching

In this chapter, execution models for the parallel matching are examined. Firstly, three kinds of parallelisms are extracted from MASS. Each of them will be represented as a parallel algorithm. Next in Section 3.2, three typical machine models for the parallel matching will be introduced. Each of them will be used as a simulation model in Chapter 4.

3.1 Parallelism in MASS

As was pointed out by Gupta, matching algorithms based on the state-saving method (e.g., MASS or Rete) also contains several kinds of parallelisms [4]. They are classified into following three types according to the level of the operation. (a) **rule level OR** parallelism in CT, (b) **CE level AND/OR** parallelism in VT, and (c) **search level OR** parallelism in VT.

Figure 2 shows a parallel algorithm to support (a). Steps 3-6 of MASS can be executed by every CE's. For parallelism (b), we may merge two matching sequences for different rules, if they have an identical structure to save the processing time. Figure 3 shows a parallel algorithm to support (b). Parallelism (c) is positioned at the most primitive level of the matching operation. We could save the time for the operation by using a complicated data structure such as hash tables or B-trees. Figure 4 shows a parallel algorithm to support parallelism (c). Note that several techniques to save the processing time tend to reduce the degree of the parallelism and to increase the overhead due to the complicated structure. Hence they would not be immediately applicable to the parallel execution. We have to carefully examine the trade-off on the effectiveness if we would improve the efficiency of the parallel processing applying these techniques.

3.2 Machine Model

3.2.1 The Number of Processors

Gupta and Forgy provided several results about measurements on existent PS programs [3]. Their measurements are based on Rete algorithm and six existent PS programs with 100-2,000 rules. The results concerned with parallel matching are summarized below [3]:

(A1) About hundred attribute checks are made in CT per action in average, and the success rate is 5-15%.

(A2) About ten clusters (α-nodes) are modified by an action in average.

(A3) Dozens of CE's (20-40) are visited for VT processing (i.e., parallelism (b)) per action in average.

(A4) The sharing on (b) reduces the number of clusters to 20-50% and the number of AND evaluation among CE's to 80-100%.

(A5) The sharing on (a) reduces the number of attribute check to 10-25%.

Note that (A4) and (A5) are static properties of PS programs. (A2)-(A4) indicate that dozens of parallelism (32-64) is sufficient to support VT processing even if the sharing on (b) is not applied. Such locality of the state-saving matching operation is independent to the number of rules. For CT processing, 32-64 processors are sufficient as well. Consequently, we should adopt the machine model with at most 32-64 processors to support parallelism (a) and (b) effectively for general expert systems.

3.2.2 Interconnection among Processors

The parallel matching has the following properties :

(B1) Broadcasting of data and the collecting of rule instances are performed by every matching phase.

(B2) The matching operation would have a dense and irregular structure.

Hence, we should adopt an interconnection scheme with high connectivity and short diameter. Particularly, shared bus or multiple bus [10] would be reasonable because of the locality of activated clusters in VT processing (see Observation (A3)).

3.2.3 Advantages of CAM

CAM has become a practicable device for general computer systems [1]. The search level parallelism includes a primitive operation suitable to CAM device, which is the search of data satisfying the predicates in a condition element. Although CAM device is available for the matching operation by itself [7], the effect of CAM increases in the following situations :

(C1) CAM is available if each predicate in CE's is simple or a composition of simple predicates.

(C2) The number of data per cluster is sufficiently large and there exist no complicated data structures to support the data search effectively.

In consequence, in following chapters, we will adopt the machine model with 32-64 processors on which CAM device may be used as the local memory. And we assert that those processors are connected to each other by shared bus or multiple bus.

4 Simulation

This chapter provides simulation results on the parallel matching. We firstly clarify the simulation model in Section 4.1. Simulation in later sections is classified according to the level of the parallelism, i.e., (1) the CE level parallelism, (2) the search level parallelism, and (3) the integration of them.

4.1 Simulation Model

We consider a shared bus (or multiple bus) connected multiprocessor system with at most 64 processors in the following simulations. Firstly, we make several assumptions for simulation on the computation and the communication.

Assume that every pattern matching between a datum and a CE takes a constant time t_{comp}. Thus in CT processing, test for each CE (i.e., a test-node in Rete) takes t_{comp} time, while test for each CE in VT processing requires $t_{comp} \times n$ time where n means the number of data in the corresponding cluster. If CAM device is used for VT, the time reduces to $t_{comp} \times m$ where m is the number of successful data on the test.

On the simulation in Section 4.2 [4], time for each node is weighted to reflect the effect of the use of a hash table, which is derived from the trace of simulation programs. Note that the weight is an average cost.

Assumptions on communication among processors are summarized below.

1. *We consider contention free memory and shared bus, and each processor has an ideal (i.e. long enough) buffer for the communication.*
2. *Each token is transmitted from one processor to others in t_comm time.*
3. *We consider the communication overhead at the start of every data transmission. We assume that it takes $t_{commovh}$ time per transmission. (e.g., if n token are transmitted continuously, it takes $t_{comm} \times n + t_{commovh}$ time).*

The image of 1-3 is illustrated in Figure 5.

The simulation in Section 4.2 assumes that the amount of $t_{commovh}$ during the program execution (i.e., $\sum t_{commovh}$) reduces the total execution time to 60 % [4]. The simulation in Section 4.3 neglects $t_{commovh}$, however, according to the experiment on a multiprocessor, $t_{commovh}$ would be negligible. In Section 4.4 we let $t_{comp} = t_{comm} + t_{commovh}$ to clarify the difference of two memory devices CAM and RAM (random access memory).

4.2 CE Level Parallelism

This section is a summary of [4]. Gupta further classified the CE level parallelism as follows.

(1) Production parallelism

Partition RB into several parts and perform VT for each parts independently. Note that it contains a redundancy, since matching sequences among different parts are not shared (the loss factor against sequential Rete is estimated about 1.64).

(2) Node parallelism

It is a direct parallel implementation of Rete on which VT is performed in parallel by CE as in Figure 3. To extract the parallelism, the sharing ratio should be reduced. The loss factor is about 1.22.

(3) Intra-node parallelism

Focusing on a CE to be examined, perform VT in parallel by token for the CE. The loss factor is the same as (2).

(4) Action parallelism

This parallelism includes CT besides VT, namely, for each datum in $\Delta DB(t-1)$, they are performed in parallel. Note that it can be applied together with (1)-(3).

Gupta estimated the effect of 1-4 using eight existent PS programs. Programs consist of 100-2,000 rules, and during the execution, 1,000-3,000 data are manipulated by rules. Matching processes are allocated onto processors as follows : (a) it allocates each cluster onto (plural) processors without the decomposition, and (b) the allocation is performed dynamically. Table 1 summarizes the results on which the average of eight programs is represented.

In this table, two kinds of speedup measurements are represented. Nominal means the speedup against the sequential execution including some overhead due to the parallel execution (e.g., synchronization or scheduling), while true means the speedup against that without any overhead (i.e., the fastest sequential execution).

From Table 1, we can assert the following facts.

1. The speedup ratio is less than 10 without (4). It is caused by the irregularity of load in the parallel execution besides the essential parallelism indicated in Section 3.2 (A).

2. The speedup ratio does not exceed 20 even if (4) is applied. It suggests that the degree of the parallelism on (4) is rather low on most PS programs.

3. The total loss factor due to merge, scheduling, and synchronization reduces the effect of the parallel execution to one half or one third (e.g., we could obtain only twice speedup with (1)).

Above examination implies that we can not achieve a significant (e.g. 100-1,000 folds) speedup only by the CE level parallelism, since the effect would be at most 10 folds. From another viewpoint, to support the parallelism effectively, a shared bus connected multiprocessor system consisting of at most 20 powerful RISC processors is sufficient as is pointed out by [5] as long as it applies only the CE level parallelism.

4.3 Search Level Parallelism

To estimate the effect of search level parallel processing of VT, we implemented procedure of Figure 4 on prototype multiprocessor UNIP. UNIP is a bus-connected multiprocessor with 31 processors (1 master and 30 slave processors).

Matching processes are allocated onto the multiprocessor as follows : (a) each cluster is partitioned into p subclusters where p means the number of available processors, and (b) a subclusters is allocated to a processor. The partition and the allocation of clusters are performed dynamically. UNIP provides two kinds of communication schemes at the hardware level, which are (1) broadcasting from master to all slaves and (2) point-to-point communication between master and each of slaves. Thus the broadcasting from arbitrary slaves has to be simulated using above facilities. To reduce the communication overhead $\sum t_{commovh}$, the data transmission at step 5 of Exam is suspended until current Buff(i) becomes empty (thus the communication occurs synchronously).

Simulation was performed in two steps, i.e., (1) preliminary software simulation to estimate the degree of the search level parallelism in the matching operation, and (2) simulation on the multiprocessor to evaluate the availability and the feasibility. We used two programs (Search1 and Search2) for our experiments.

Figure 6 shows results of the preliminary experiment. For the comparison, the effect of the CE level parallelism is also illustrated. The search level parallelism is more significant in Search1, while it is less significant in Search2. It means that if the DB size is large and the CE level parallelism is small, the availability of the search level parallelism for the speedup would be increased.

Next we implemented problem Search1 on UNIP. Figure 7 shows a result of the scaling. On the scaling, we assume 10 MIPS CPU (e.g., Transputer [15]) and 100 Mbyte/sec communication rate [5]. Figure 8 shows an ideal speedup based on preliminary experiments on which no scheduling time (i.e., time for partition and allocation of clusters) is considered. Because the curve in Figure 7 including scheduling time is close to the ideal curve in Figure 8, we can assert that the scheduling time on our scheme is sufficiently small to be negligible. It is contrastive to that the CE level parallelism requires a hardware scheduler to support the dynamic cluster allocation effectively [4].

4.4 Multiprocessor with CAM Local Memory

The use of CAM device would increase the speed for the matching operation. To verify it by the simulation, we used following two programs.

(EFarmer) An extension of "farmer's dilemma" consisting of 4 rules and 4 clusters. It is extended as follows : (1) the number of objects (foxes, geese, and corn) is n, which is used as a parameter, and (2) geese (corn) are eaten by foxes (geese) when the number of geese (corn) is equal to the number of foxes (geese) minus 2 or less.

(EMonkey) An extension of monkey and banana's problem consisting of 20 rules and 20 clusters. It is extended so that n redundant objects, which will be used as a parameter, are scattered in the room at the initial state.

Figure 9 shows the effect of CAM on EFarmer on which single processor is assumed. The horizontal axis indicates the times of rule execution, and the vertical axis indicates the times of memory access during the program execution. Obviously, the frequency of the memory access can be reduced by using CAM (note that similar speedup could be obtained using a more complicated data structure such as hash tables).

The performance of multiprocessor with CAM is evaluated by problem EMonkey. Matching processes are allocated onto processors as follows : each cluster is allocated onto a processor in static without the decomposition. The CE level parallelism is supported by the multiprocessor, and the search level parallelism is supported by CAM adopted to each of processors. When the number of redundant objects n is smallm the speedup ratio is not so significant. However, as the number increases, the speedup ratio becomes remarkable. Furthermore, the difference between RAM and CAM will be expanded if we use an interconnection network with broad bandwidth. In the simulation, we assumed the use of a multiport memory (denoted by 3-D CM).

5 Discussion

From results in the last chapter, we have obtained following observations.

Firstly, the CE level parallelism is not so significant for actual PS programs. It is due to the rule level synchronization at each conflict resolution phase and the locality of the DB modification. The first reason will be removed if we can apply parallel rule execution discussed in Section 2.2, however, the second reason is essential problem for the speedup. It means that the CE level parallelism contains an obvious limitation for the speedup. The search level parallelism, on the other hand, seems to have more possibility for the speedup.

The originalities of our research are described as follows. Firstly, we applied two kinds of multiprocessor architectures different from tree-structured parallel processors [14][13], i.e. (a) a shared bus connected multiprocessor system, and (b) the use of advanced VLSI technology on the architecture. Both of them will be used to implement the search level parallelism effectively. We proposed a new parallel matching scheme based on (a) on which load balancing during the matching operation can be effectively achieved with low scheduling overhead. It is expected to

break the limitation of the CE level parallelism. In fact, if there are a large number of data to be searched (e.g., several pages per cluster) and if any other techniques to accelerate the DB search (e.g., hash tables or CAM) are disable, the availability of the scheme based on (a) would be increased.

The latter architecture, on the other hand, is assumed to integrate both of above two parallelisms. Although CAM is powerful to support the search level parallelism by itself as shown in Figure 9, it can be further accelerated by coupling it with a high speed communication network. It is obvious that the proposed scheme based on (b) illustrates the essential limitation on the speedup of the matching operation by the parallel processing, since there are no more parallelisms to be extracted.

In this paper, we considered no trade-off due to the implementation cost or other practical factors, since the target of the paper is to illustrate the possibility and the limitation on the speedup by the parallel processing of the matching operation. In practice, therefore, although the scheme based on (b) is preferable to speedup the matching operation for building a real-time PS, we should select the scheme based on (a) if we consider the current technologies. The VLSI technology to realize an ideal multiport memory assumed in (b) is just at an experimental level, though it will be at an practical level within the next decade.

6 Remarks

Results obtained in this paper are summarized below.

1. The parallel matching to speed up the computation of PS programs is experimentally examined. The matching operation contains two level parallelisms, i.e., the CE level and the search level. The former has an obvious limitation for the speedup caused by the nature of PS programs. Thus to reduce the computation time as much as possible, we should focus on the latter parallelism as well.

2. To extract the search level parallelism, two kinds of parallel processing architectures are assumed. Based on the architectures, two parallel matching schemes to support the search level parallelism are newly proposed. The allocation of the schemes onto the parallel processing architectures are also illustrated.

3. The availability of the proposed schemes are verified using benchmark programs. The simulation results show us the possibility of the search level parallelism for the further speedup of the matching operation.

References

[1] L.Chisvin and R.J.Duckworth, "Content-Addressable and Associative Memory: Alternatives to the Ubiquitous RAM," *IEEE Computer*, 22, 7 (July 1989) 51-64.

[2] C.L.Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artif. Intel.*, 19 (1982) 17-37.

[3] A.Gupta and C.L.Forgy: *Mesurements on Production Systems*, Technical Report CMU-CS-83-167, Crnegie-Mellon Univ. (1983).

[4] A.Gupta: *Parallelism in Production Systems*, Pitman Publishing (1987).

[5] A.Gupta, et al.: *Results of Parallel Implementation of OPS5 on the Encore Multiprocessor*, Techinical Report CMU-CS-87-146, Carnegie-Mellon Univ. (1987).

[6] T.Ishida, "Optimizing Condition Parts of Production System Programs," *Trans. IPS Japan*, 29, 12 (Dec.1988) 1158-1169, in Japanese.

[7] P.M.Kogge, et al., "VLSI and Rule-Based Systems," *Proc. of Int'l Workshop on VLSI for Artif. Intel.* (July 1988) D3.

[8] J.McDermott, et al., "The Efficiency of Certain Production System Implementations," *Pattern Directed Inference Systems* Academic Press (1978) 155-176.

[9] J.Miyazaki, et al., "A Shared Memory Architecture for MANJI Production System", *Database Machines and Knowledge Base Machines*, Kluwer Academic Pblishers (1988) 517-531.

[10] T.N.Mudge,et al., "Multiple Bus Architectures," *IEEE Computer*, 20, 6 (June 1987) 42-48.

[11] L.L.Schor,et al., "Advances in Rete Pattern Matching," *Proc. of AAAI-86*, IEEE (1987) 166-169.

[12] F.Schreiner and G.Zimmermann, "PESA 1 - A Parallel Architecture for Production Systems," *Proc. of 1987 ICPP*, IEEE (1987) 166-169.

[13] D.E.Shaw, "On the Range of Applicability of an Artificial Intelligence Machine," *Artif. Intel.*, 32 (1987) 151-172.

[14] S.J.Stolfo, "Initial Performance of the DADO-2 Prototype," *IEEE Computer*, 20, 1 (Jan.1987) 75-83.

[15] C.Whitby-Strevens, "The Transputer," *Proc. of 12th International Symposium on Computer Architecture* (June 1985) 292-300.

procedure MASS

```
    input:   CS(t-1)   /* the conflict set at (t-1)th cycle */
             Δ DB(t-1)   /* differential of DB at (t-1)th cycle */
             all CE'_i s   /* the clustered database */
    output:  CS(t)   /* the conflict set at t-th cycle */
      begin
 1:   Buff := φ;                              /* CT Processing */
 2:   for all condition element CE_i
 3:     for all data in ΔDB(t − 1)
 4:       if data and CE_i satisfy condition (1) then {
 5:         τ := <id of CE_i, tag of data, value of data >;
 6:         Buff := Buff ∪ {τ}};
 7:   ΔCS := φ;                               /* VT processing */
 8:   repeat {
 9:     pick up a token from Buff;
10:     if token is a rule instance
11:       then ΔCS := ΔCS ∪ {token}
12:       else for each CE_i to be examined {          /* database search */
13:         sign := ( tag of token)×(tag of CE_i );
14:         for all data in CE_i
15:           if token and data satisfy condition (2) then {
16:             id := id determined by token and CE_i;
17:             τ :=<id, sign, token ∪ data >;
18:             Buff := Buff ∪{τ}}};
19:   until Buff=φ};
20:   CS(t) := CS(t-1);                  /* Generation of CS(t) */
21:   for all token in ΔCS
22:     if tag of token is plus
23:       then CS(t) := CS(t) ∪ {token}
24:       else CS(t) := CS(t) - {token}
      end.
```

Figure 1 State-saving matching algorithm MASS

```
/* parallelized CT processing */

1:    Buff := φ ;
2:      for all condition element CE_i do in parallel
3:      for all data in ΔDB(t − 1)
4:        steps 4-6 of MASS;
```

Figure 2 CT processing to support CE level parallelism.

```
/* stream-or VT processing ( on processor i ) */

1:    ΔCS(i) := φ;
2:    repeat {
3:      pick up a token from Buff(i);
4:      if the token is a rule instance
5:        then ΔCS(i) := ΔCS(i) ∪ {token}
6:        else if the token is concerned with processor i then
7:          for each CE_i to be examined {
8:            sign := ( tag of token )×( tag of CE_i );
9:            for all data in CE_i Exam(data, token, sign)};
10:   until Buff(i)'s for all i are empty };

procedure Exam(data, token, sign)
      begin
1:    if token and data satisfy condition (2) then {
2:      id := id determined by token and CE_i;
3:      τ := <id, sign, token ∪ data >;
4:      for all i
5:        Buff(i) := Buff(i)∪{τ}}              /* broadcast of data */
      end.
```

Figure 3 VT processing to support CE level parallelism.

```
/* search-or VT processing ( on processor i ) */

1:    ΔCS(i) := φ;
2:    repeat {
3:      pick up a token from Buff(i);
4:      if token is a rule instance
5:        then ΔCS(i) := ΔCS(i) ∪ {token}
6:        else for each CE_i to be examined {          /* database search */
7:          sign := (tag of token)×(tag of CE_i);
8:          for all data in CE_i do in parallel Exam(data, token, sign)};
9:    until Buff(i)'s for all i are empty };
```

Figure 4 VT processing to support search level parallelism.

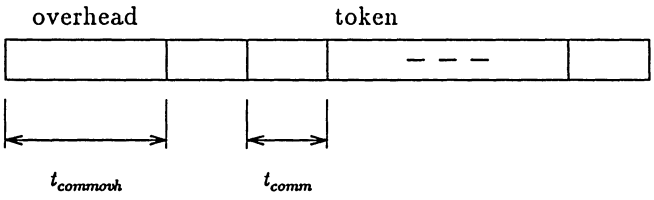


Figure 5 Assumptions on communication.

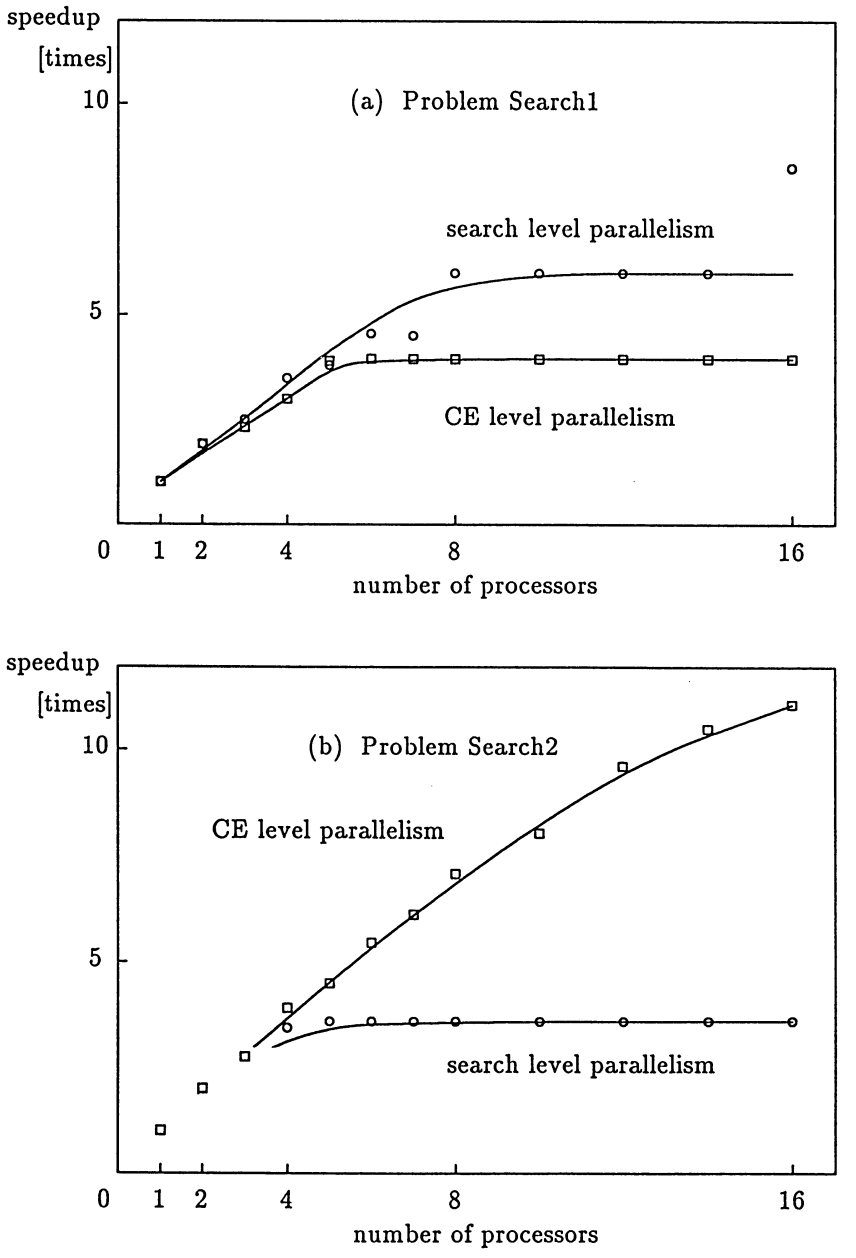


Figure 6 Preliminary experiments on the search level parallelism.

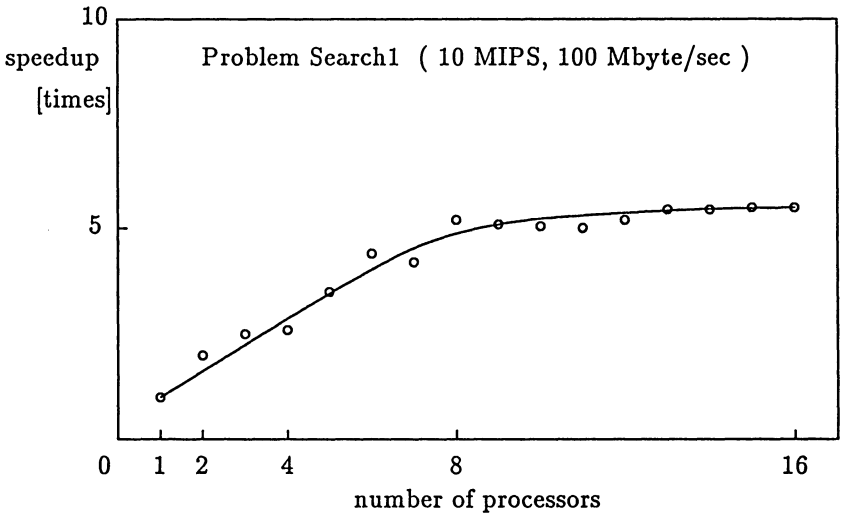


Figure 7 Speedup on a high speed multiprocessor
(including $t_{commovh}$).

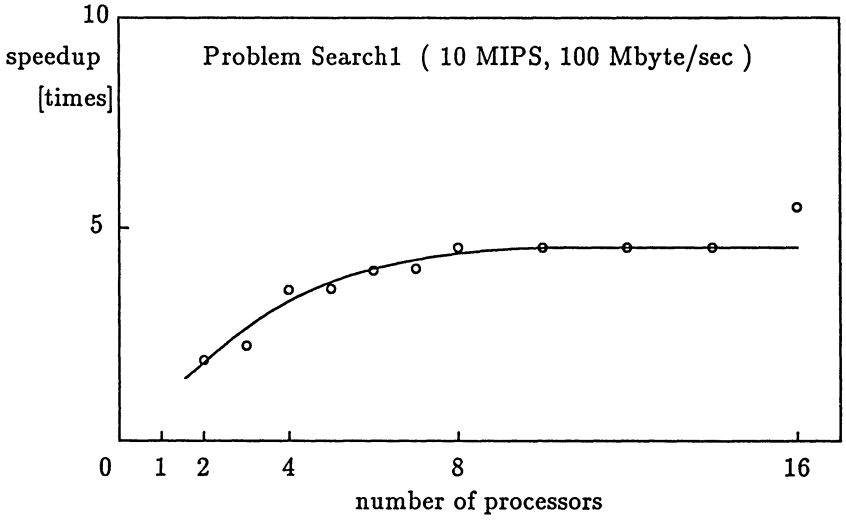


Figure 8 Speedup on a high speed multiprocessor
(without $t_{commovh}$).

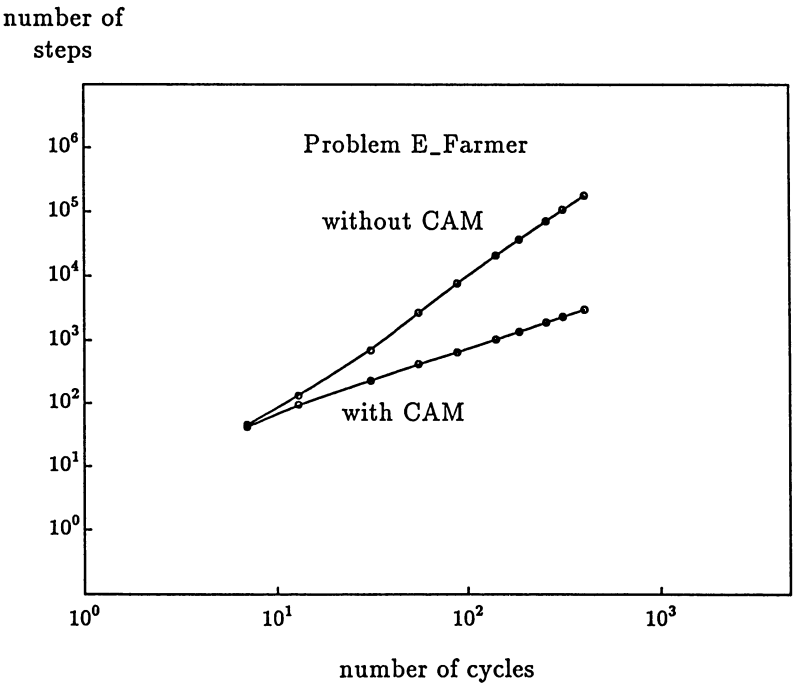


Figure 9 Effect of CAM on a single processor.

Table 1 Simulation results on CE level parallelism [4].

Type of Parallelism	Speedup ratio		Saturation	Overhead	
	nominal	true	No. of Proc.	merge	total
(1) (+(4))	5.1 (7.6)	1.9 (1.5)	16~32	1.64	2.65
(2) (+(4))	5.8 (10.7)	2.9 (5.4)	16~32	1.22	1.98
(3) (+(4))	7.6 (19.3)	3.9 (10.2)	32~64	1.22	1.98

(1) Production parallelism (2) Node parallelism
(3) Intra-node parallelism (4) Action parallelism

Notes :

(a) Speedup ratio is the average of 8 programs.
Nominal is the speedup for sequential execution assuming overhead, and *true* is the speedup for that assuming no overhead.

(b) Overhead factor due to "synchronization and scheduling" is 1.62.
(i.e., 2.65=1.64×1.62, and 1.98=1.22×1.62)

Author Index Volume I

Lecture Notes in Computer Science

For information about Vols. 1–411
please contact your bookseller or Springer-Verlag

Vol. 412: L.B. Almeida, C.J. Wellekens (Eds.), Neural Networks. Proceedings, 1990. IX, 276 pages. 1990.

Vol. 413: R. Lenz, Group Theoretical Methods in Image Processing. VIII, 139 pages. 1990.

Vol. 414: A. Kreczmar, A. Salwicki, M. Warpechowski, LOGLAN '88 – Report on the Programming Language. X, 133 pages. 1990.

Vol. 415: C. Choffrut, T. Lengauer (Eds.), STACS 90. Proceedings, 1990. VI, 312 pages. 1990.

Vol. 416: F. Bancilhon, C. Thanos, D. Tsichritzis (Eds.), Advances in Database Technology – EDBT '90. Proceedings, 1990. IX, 452 pages. 1990.

Vol. 417: P. Martin-Löf, G. Mints (Eds.), COLOG-88. International Conference on Computer Logic. Proceedings, 1988. VI, 338 pages. 1990.

Vol. 418: K.H. Bläsius, U. Hedtstück, C.-R. Rollinger (Eds.), Sorts and Types in Artificial Intelligence. Proceedings, 1989. VIII, 307 pages. 1990. (Subseries LNAI).

Vol. 419: K. Weichselberger, S. Pöhlmann, A Methodology for Uncertainty in Knowledge-Based Systems. VIII, 136 pages. 1990 (Subseries LNAI).

Vol. 420: Z. Michalewicz (Ed.), Statistical and Scientific Database Management, V SSDBM. Proceedings, 1990 V, 256 pages. 1990.

Vol. 421: T. Onodera, S. Kawai, A Formal Model of Visualization in Computer Graphics Systems. X, 100 pages. 1990.

Vol. 422: B. Nebel, Reasoning and Revision in Hybrid Representation Systems. XII, 270 pages. 1990 (Subseries LNAI).

Vol. 423: L.E. Deimel (Ed.), Software Engineering Education. Proceedings, 1990. VI, 164 pages. 1990.

Vol. 424: G. Rozenberg (Ed.), Advances in Petri Nets 1989. VI, 524 pages. 1990.

Vol. 425: C.H. Bergman, R.D. Maddux, D.L. Pigozzi (Eds.), Algebraic Logic and Universal Algebra in Computer Science. Proceedings, 1988. XI, 292 pages. 1990.

Vol. 426: N. Houbak, SIL – a Simulation Language. VII, 192 pages. 1990.

Vol. 427: O. Faugeras (Ed.), Computer Vision – ECCV 90. Proceedings, 1990. XII, 619 pages. 1990.

Vol. 428: D. Bjørner, C.A.R. Hoare, H. Langmaack (Eds.), VDM '90. VDM and Z – Formal Methods in Software Development. Proceedings, 1990. XVII, 580 pages. 1990.

Vol. 429: A. Miola (Ed.), Design and Implementation of Symbolic Computation Systems. Proceedings, 1990. XII, 284 pages. 1990.

Vol. 430: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (Eds.), Stepwise Refinement of Distributed Systems. Models, Formalisms, Correctness. Proceedings, 1989. X, 808 pages. 1990.

Vol. 431: A. Arnold (Ed.), CAAP '90. Proceedings, 1990. VI, 285 pages. 1990.

Vol. 432: N. Jones (Ed.), ESOP '90. Proceedings, 1990. IX, 436 pages. 1990.

Vol. 433: W. Schröder-Preikschat, W. Zimmer (Eds.), Progress in Distributed Operating Systems and Distributed Systems Management. Proceedings, 1989. V, 206 pages. 1990.

Vol. 434: J.-J. Quisquater, J. Vandewalle (Eds.), Advances in Cryptology – EUROCRYPT '89. Proceedings, 1989. X, 710 pages. 1990.

Vol. 435: G. Brassard (Ed.), Advances in Cryptology – CRYPTO '89. Proceedings, 1989. XIII, 634 pages. 1990.

Vol. 436: B. Steinholtz, A. Sølvberg, L. Bergman (Eds.), Advanced Information Systems Engineering. Proceedings, 1990. X, 392 pages. 1990.

Vol. 437: D. Kumar (Ed.), Current Trends in SNePS – Semantic Network Processing System. Proceedings, 1989. VII, 162 pages. 1990. (Subseries LNAI).

Vol. 438: D.H. Norrie, H.W. Six (Eds.), Computer Assisted Learning – ICCAL '90. Proceedings, 1990. VII, 467 pages. 1990.

Vol. 439: P. Gorny, M. Tauber (Eds.), Visualization in Human-Computer Interaction. Proceedings, 1988. VI, 274 pages. 1990.

Vol. 440: E. Börger, H. Kleine Büning, M.M. Richter (Eds.), CSL '89. Proceedings, 1989. VI, 437 pages. 1990.

Vol. 441: T. Ito, R.H. Halstead, Jr. (Eds.), Parallel Lisp: Languages and Systems. Proceedings, 1989. XII, 364 pages. 1990.

Vol. 442: M. Main, A. Melton, M. Mislove, D. Schmidt (Eds.), Mathematical Foundations of Programming Semantics. Proceedings, 1989. VI, 439 pages. 1990.

Vol. 443: M.S. Paterson (Ed.), Automata, Languages and Programming. Proceedings, 1990. IX, 781 pages. 1990.

Vol. 444: S. Ramani, R. Chandrasekar, K.S.R. Anjaneyulu (Eds.), Knowledge Based Computer Systems. Proceedings, 1989. X, 546 pages. 1990. (Subseries LNAI).

Vol. 445: A.J.M. van Gasteren, On the Shape of Mathematical Arguments. VIII, 181 pages. 1990.

Vol. 446: L. Plümer, Termination Proofs for Logic Programs. VIII ,142 pages. 1990. (Subseries LNAI).

Vol. 447: J.R. Gilbert, R. Karlsson (Eds.), SWAT '90. 2nd Scandinavian Workshop on Algorithm Theory. Proceedings, 1990. VI, 417 pages. 1990.

Vol. 448: B. Simons, A. Spector (Eds.), Fault Tolerant Distributed Computing. VI, 298 pages. 1990.

Vol. 449: M.E. Stickel (Ed.), 10th International Conference on Automated Deduction. Proceedings, 1990. XVI, 688 pages. 1990. (Subseries LNAI).

Vol. 450: T. Asano, T. Ibaraki, H. Imai, T. Nishizeki (Eds.), Algorithms. Proceedings, 1990. VIII, 479 pages. 1990.

Vol. 451: V. Marík, O. Stepánková, Z. Zdráhal (Eds.), Artificial Intelligence in Higher Education. Proceedings, 1989. IX, 247 pages. 1990. (Subseries LNAI).

Vol. 452: B. Rovan (Ed.), Mathematical Foundations of Computer Science 1990. Proceedings, 1990. VIII, 544 pages. 1990.

Vol. 453: J. Seberry, J. Pieprzyk (Eds.), Advances in Cryptology – AUSCRYPT ´90 Proceedings, 1990. IX. 462 pages. 1990.

Vol. 454: V. Diekert, Combinatorics on Traces. Xll, 165 pages. 1990.

Vol. 455: C.A. Floudas, P.M. Pardalos, A Collection of Test Problems for Constrained Global Optimization Algorithms. XIV, 180 pages. 1990.

Vol. 456: P. Deransart, J. Maluszyn´ski (Eds.), Programming Language Implementation and Logic Programming. Proceedings, 1990. Vlll, 401 pages. 1990.

Vol. 457: H. Burkhart (Ed.), CONPAR ´90 – VAPP IV. Proceedings, 1990. XIV, 900 pages. 1990.

Vol. 458: J.C.M. Baeten, J.W. Klop (Eds.), CONCUR ´90. Proceedings, 1990. Vll, 537 pages. 1990.

Vol. 459: R. Studer (Ed.), Natural Language and Logic. Proceedings, 1989. Vll, 252 pages. 1990. (Subseries LNAI).

Vol. 460: J. Uhl, H.A. Schmid, A Systematic Catalogue of Reusable Abstract Data Types. Xll, 344 pages. 1990.

Vol. 461: P. Deransart, M. Jourdan (Eds.), Attribute Grammars and their Applications. Proceedings, 1990. Vlll, 358 pages. 1990.

Vol. 462: G. Gottlob, W. Nejdl (Eds.), Expert Systems in Engineering. Proceedings, 1990. IX, 260 pages. 1990. (Subseries LNAI).

Vol. 463: H. Kirchner, W. Wechler (Eds.), Algebraic and Logic Programming. Proceedings, 1990. Vll, 386 pages. 1990.

Vol. 464: J. Dassow, J. Kelemen (Eds.), Aspects and Prospects of Theoretical Computer Science. Proceedings, 1990. Vl, 298 pages. 1990.

Vol. 465: A. Fuhrmann, M. Morreau (Eds.), The Logic of Theory Change. Proceedings, 1989. X, 334 pages. 1991. (Subseries LNAI).

Vol. 466: A. Blaser (Ed.), Database Systems of the 90s. Proceedings, 1990. Vlll, 334 pages. 1990.

Vol. 467: F. Long (Ed.), Software Engineering Environments. Proceedings, 1969. Vl, 313 pages. 1990.

Vol. 468: S.G. Akl, F. Fiala, W.W. Koczkodaj (Eds.), Advances in Computing and lnformation – lCCI ´90. Proceedings, 1990. VII, 529 pages. 1990.

Vol. 469: I. Guessarian (Ed.), Semantics of Systeme of Concurrent Processes. Proceedings, 1990. V, 456 pages. 1990.

Vol. 470: S. Abiteboul, P.C. Kanellakis (Eds.), ICDT ´90. Proceedings, 1990. Vll, 528 pages. 1990.

Vol. 471: B.C. Ooi, Efficient Query Processing in Geographic Information Systems. Vlll, 208 pages. 1990.

Vol. 472: K.V. Nori, C.E. Veni Madhavan (Eds.), Foundations of Software Technology and Theoretical Computer Science. Proceedings, 1990. X, 420 pages. 1990.

Vol. 473: I.B. Damgård (Ed.), Advances in Cryptology – EUROCRYPT ´90. Proceedings, 1990. Vlll, 500 pages. 1991.

Vol. 474: D. Karagiannis (Ed.), Information Syetems and Artificial Intelligence: Integration Aspects. Proceedings, 1990. X, 293 pages. 1991. (Subseries LNAI).

Vol. 475: P. Schroeder-Heister (Ed.), Extensions of Logic Programming. Proceedings, 1989. Vlll, 364 pages. 1991. (Subseries LNAI).

Vol. 476: M. Filgueiras, L. Damas, N. Moreira, A.P. Tomás (Eds.), Natural Language Processing. Proceedings, 1990. Vll, 253 pages. 1991. (Subseries LNAI).

Vol. 477: D. Hammer (Ed.), Compiler Compilers. Proceedings, 1990. Vl, 227 pages. 1991.

Vol. 478: J. van Eijck (Ed.), Logics in Al. Proceedings, 1990. IX, 562 pages. 1991. (Subseries in LNAI).

Vol. 480: C. Choffrut, M. Jantzen (Eds.), STACS 91. Proceedings, 1991. X, 549 pages. 1991.

Vol. 481: E. Lang, K.-U. Carstensen, G. Simmons, Modelling Spatial Knowledge on a Linguistic Basis. IX, 138 pages. 1991. (Subseries LNAI).

Vol. 482: Y. Kodratoff (Ed.), Machine Learning – EWSL-91. Proceedings, 1991. XI, 537 pages. 1991. (Subseries LNAI).

Vol. 483: G. Rozenberg (Ed.), Advances In Petri Nets 1990. VI, 515 pages. 1991.

Vol. 484: R. H. Möhring (Ed.). Graph-Theoretic Concepts In Computer Science. Proceedings, 1990. IX, 360 pages. 1991.

Vol. 485: K. Furukawa, H. Tanaka. T. Fullsaki (Eds.), Logic Programming '89. Proceedings, 1989. IX, 183 pages. 1991. (Subseries LNAI)

Vol. 486: J. van Leeuwen. N. Santoro (Eds.). Distributed Algorithms. Proceedings, 1990. VI, 433 pages. 1991.

Vol. 487: A. Bode (Ed.). Dlstributed Memory Computing. Proceedings, 1991. XI, 506 pages. 1991

Vol. 488: R. V. Book (Ed.). Rewriting Techniques and Applications. Proceedings, 1991. VII, 458 pages. 1991.

Vol. 489: J. W. de Bakker, W. P. de Roever, G. Rozenberg (Eds.), Foundations of Object-Oriented Languages. Proceedings, 1990. VIII, 442 pages. 1991.

Vol. 490: J. A. Bergstra, L. M. G. Feljs (Eds .), Algebraic Methods 11: Theory, Tools and Applicatlons. VI, 434 pages. 1991.

Vol. 491: A. Yonezawa, T. Ito (Eds.). Concurrency: Theory, Language, and Architecture. Proceedings, 1989. VIII, 339 pages. 1991.

Vol. 492: D. Sriram, R. Logcher, S. Fukuda (Eds.). Computer-Aided Cooperative Product Development. Proceedings, 1989 VII, 630 pages. 1991.

Vol. 493: S. Abramsky, T. S. E. Maibaum (Eds.). TAPSOFT '91. Volume 1. Proceedings, 1991. VIII, 455 pages. 1991.

Vol. 494: S. Abramsky, T. S. E. Maibaum (Eds.). TAPSOFT '91. Volume 2. Proceedings, 1991. VIII, 482 pages. 1991

Vol. 495: 9. Thalheim, J. Demetrovics, H.-D. Gerhardt (Eds.), MFDBS '91. Proceedings, 1991. VI, 395 pages. 1991.

Vol. 497: F. Dehne, F. Fiala. W.W. Koczkodaj (Eds.), Advances in Computing and Intormation - ICCI '91 Proceedings, 1991. VIII, 745 pages. 1991.

Vol. 498: R. Andersen, J. A. Bubenko jr., A. Sølvberg (Eds.), Advanced Information Systems Engineering. Proceedings, 1991. VI, 579 pages. 1991.

Vol. 499: D. Christodoulakis (Ed.), Ada: The Choice for '92. Proceedings, 1991. VI, 411 pages. 1991.

Vol. 500: M. Held, On the Computational Geometry of Pocket Machining. XII, 179 pages. 1991.

Vol. 501: M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, D. Sannella (Eds.). Algebraic System Specification and Development. VIII, 98 pages. 1991.

Vol. 502: J. Bārzdiņš, D. Bjørner (Eds.), Baltic Computer Science. X, 619 pages. 1991.

Vol. 503: P. America (Ed.), Parallel Database Systems. Proceedings, 1990. VIII, 433 pages. 1991.

Vol. 504: J. W. Schmidt, A. A. Stogny (Eds.), Next Generation Information System Technology. Proceedings, 1990. IX, 450 pages. 1991.

Vol. 505: E. H. L. Aarts, J. van Leeuwen, M. Rem (Eds.), PARLE '91. Parallel Architectures and Languages Europe, Volume I. Proceedings, 1991. XV, 423 pages. 1991.

Vol. 506: E. H. L. Aarts, J. van Leeuwen, M. Rem (Eds.), PARLE '91. Parallel Architectures and Languages Europe, Volume II. Proceedings, 1991. XV, 489 pages. 1991.

Author Index Volume II

Lecture Notes in Computer Science

For information about Vols. 1–411
please contact your bookseller or Springer-Verlag

Vol. 412: L.B. Almeida, C.J. Wellekens (Eds.), Neural Networks. Proceedings, 1990. IX, 276 pages. 1990.

Vol. 413: R. Lenz, Group Theoretical Methods in Image Processing. VIII, 139 pages. 1990.

Vol. 414: A. Kreczmar, A. Salwicki, M. Warpechowski, LOGLAN '88 – Report on the Programming Language. X, 133 pages. 1990.

Vol. 415: C. Choffrut, T. Lengauer (Eds.), STACS 90. Proceedings, 1990. VI, 312 pages. 1990.

Vol. 416: F. Bancilhon, C. Thanos, D. Tsichritzis (Eds.), Advances in Database Technology – EDBT '90. Proceedings, 1990. IX, 452 pages. 1990.

Vol. 417: P. Martin-Löf, G. Mints (Eds.), COLOG-88. International Conference on Computer Logic. Proceedings, 1988. VI, 338 pages. 1990.

Vol. 418: K.H. Bläsius, U. Hedtstück, C.-R. Rollinger (Eds.), Sorts and Types in Artificial Intelligence. Proceedings, 1989. VIII, 307 pages. 1990. (Subseries LNAI).

Vol. 419: K. Weichselberger, S. Pöhlmann, A Methodology for Uncertainty in Knowledge-Based Systems. VIII, 136 pages. 1990 (Subseries LNAI).

Vol. 420: Z. Michalewicz (Ed.), Statistical and Scientific Database Management, V SSDBM. Proceedings, 1990 V, 256 pages. 1990.

Vol. 421: T. Onodera, S. Kawai, A Formal Model of Visualization in Computer Graphics Systems. X, 100 pages. 1990.

Vol. 422: B. Nebel, Reasoning and Revision in Hybrid Representation Systems. XII, 270 pages. 1990 (Subseries LNAI).

Vol. 423: L.E. Deimel (Ed.), Software Engineering Education. Proceedings, 1990. VI, 164 pages. 1990.

Vol. 424: G. Rozenberg (Ed.), Advances in Petri Nets 1989. VI, 524 pages. 1990.

Vol. 425: C.H. Bergman, R.D. Maddux, D.L. Pigozzi (Eds.), Algebraic Logic and Universal Algebra in Computer Science. Proceedings, 1988. XI, 292 pages. 1990.

Vol. 426: N. Houbak, SIL – a Simulation Language. VII, 192 pages. 1990.

Vol. 427: O. Faugeras (Ed.), Computer Vision – ECCV 90. Proceedings, 1990. XII, 619 pages. 1990.

Vol. 428: D. Bjørner, C.A.R. Hoare, H. Langmaack (Eds.), VDM '90. VDM and Z – Formal Methods in Software Development. Proceedings, 1990. XVII, 580 pages. 1990.

Vol. 429: A. Miola (Ed.), Design and Implementation of Symbolic Computation Systems. Proceedings, 1990. XII, 284 pages. 1990.

Vol. 430: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (Eds.), Stepwise Refinement of Distributed Systems. Models, Formalisms, Correctness. Proceedings, 1989. X, 808 pages. 1990.

Vol. 431: A. Arnold (Ed.), CAAP '90. Proceedings, 1990. VI, 285 pages. 1990.

Vol. 432: N. Jones (Ed.), ESOP '90. Proceedings, 1990. IX, 436 pages. 1990.

Vol. 433: W. Schröder-Preikschat, W. Zimmer (Eds.), Progress in Distributed Operating Systems and Distributed Systems Management. Proceedings, 1989. V, 206 pages. 1990.

Vol. 434: J.-J. Quisquater, J. Vandewalle (Eds.), Advances in Cryptology – EUROCRYPT '89. Proceedings, 1989. X, 710 pages. 1990.

Vol. 435: G. Brassard (Ed.), Advances in Cryptology – CRYPTO '89. Proceedings, 1989. XIII, 634 pages. 1990.

Vol. 436: B. Steinholtz, A. Sølvberg, L. Bergman (Eds.), Advanced Information Systems Engineering. Proceedings, 1990. X, 392 pages. 1990.

Vol. 437: D. Kumar (Ed.), Current Trends in SNePS – Semantic Network Processing System. Proceedings, 1989. VII, 162 pages. 1990. (Subseries LNAI).

Vol. 438: D.H. Norrie, H.W. Six (Eds.), Computer Assisted Learning – ICCAL '90. Proceedings, 1990. VII, 467 pages. 1990.

Vol. 439: P. Gorny, M. Tauber (Eds.), Visualization in Human-Computer Interaction. Proceedings, 1988. VI, 274 pages. 1990.

Vol. 440: E. Börger, H. Kleine Büning, M.M. Richter (Eds.), CSL '89. Proceedings, 1989. VI, 437 pages. 1990.

Vol. 441: T. Ito, R.H. Halstead, Jr. (Eds.), Parallel Lisp: Languages and Systems. Proceedings, 1989. XII, 364 pages. 1990.

Vol. 442: M. Main, A. Melton, M. Mislove, D. Schmidt (Eds.), Mathematical Foundations of Programming Semantics. Proceedings, 1989. VI, 439 pages. 1990.

Vol. 443: M.S. Paterson (Ed.), Automata, Languages and Programming. Proceedings, 1990. IX, 781 pages. 1990.

Vol. 444: S. Ramani, R. Chandrasekar, K.S.R. Anjaneyulu (Eds.), Knowledge Based Computer Systems. Proceedings, 1989. X, 546 pages. 1990. (Subseries LNAI).

Vol. 445: A.J.M. van Gasteren, On the Shape of Mathematical Arguments. VIII, 181 pages. 1990.

Vol. 446: L. Plümer, Termination Proofs for Logic Programs. VIII ,142 pages. 1990. (Subseries LNAI).

Vol. 447: J.R. Gilbert, R. Karlsson (Eds.), SWAT '90. 2nd Scandinavian Workshop on Algorithm Theory. Proceedings, 1990. VI, 417 pages. 1990.

Vol. 448: B. Simons, A. Spector (Eds.), Fault Tolerant Distributed Computing. VI, 298 pages. 1990.

Vol. 449: M.E. Stickel (Ed.), 10th International Conference on Automated Deduction. Proceedings, 1990. XVI, 688 pages. 1990. (Subseries LNAI).

Vol. 450: T. Asano, T. Ibaraki, H. Imai, T. Nishizeki (Eds.), Algorithms. Proceedings, 1990. VIII, 479 pages. 1990.

Vol. 451: V. Marík, O. Stepánková, Z. Zdráhal (Eds.), Artificial Intelligence in Higher Education. Proceedings, 1989. IX, 247 pages. 1990. (Subseries LNAI).

Vol. 452: B. Rovan (Ed.), Mathematical Foundations of Computer Science 1990. Proceedings, 1990. VIII, 544 pages. 1990.

Vol. 453: J. Seberry, J. Pieprzyk (Eds.), Advances in Cryptology – AUSCRYPT ´90 Proceedings, 1990. IX. 462 pages. 1990.

Vol. 454: V. Diekert, Combinatorics on Traces. Xll, 165 pages. 1990.

Vol. 455: C.A. Floudas, P.M. Pardalos, A Collection of Test Problems for Constrained Global Optimization Algorithms. XIV, 180 pages. 1990.

Vol. 456: P. Deransart, J. Maluszyn´ski (Eds.), Programming Language Implementation and Logic Programming. Proceedings, 1990. Vlll, 401 pages. 1990.

Vol. 457: H. Burkhart (Ed.), CONPAR ´90 – VAPP IV. Proceedings, 1990. XIV, 900 pages. 1990.

Vol. 458: J.C.M. Baeten, J.W. Klop (Eds.), CONCUR ´90. Proceedings, 1990. Vll, 537 pages. 1990.

Vol. 459: R. Studer (Ed.), Natural Language and Logic. Proceedings, 1989. Vll, 252 pages. 1990. (Subseries LNAI).

Vol. 460: J. Uhl, H.A. Schmid, A Systematic Catalogue of Reusable Abstract Data Types. Xll, 344 pages. 1990.

Vol. 461: P. Deransart, M. Jourdan (Eds.), Attribute Grammars and their Applications. Proceedings, 1990. Vlll, 358 pages. 1990.

Vol. 462: G. Gottlob, W. Nejdl (Eds.), Expert Systems in Engineering. Proceedings, 1990. IX, 260 pages. 1990. (Subseries LNAI).

Vol. 463: H. Kirchner, W. Wechler (Eds.), Algebraic and Logic Programming. Proceedings, 1990. Vll, 386 pages. 1990.

Vol. 464: J. Dassow, J. Kelemen (Eds.), Aspects and Prospects of Theoretical Computer Science. Proceedings, 1990. Vl, 298 pages. 1990.

Vol. 465: A. Fuhrmann, M. Morreau (Eds.), The Logic of Theory Change. Proceedings, 1989. X, 334 pages. 1991. (Subseries LNAI).

Vol. 466: A. Blaser (Ed.), Database Systems of the 90s. Proceedings, 1990. Vlll, 334 pages. 1990.

Vol. 467: F. Long (Ed.), Software Engineering Environments. Proceedings, 1969. Vl, 313 pages. 1990.

Vol. 468: S.G. Akl, F. Fiala, W.W. Koczkodaj (Eds.), Advances in Computing and lnformation – lCCI ´90. Proceedings, 1990. VII, 529 pages. 1990.

Vol. 469: I. Guessarian (Ed.), Semantics of Systeme of Concurrent Processes. Proceedings, 1990. V, 456 pages. 1990.

Vol. 470: S. Abiteboul, P.C. Kanellakis (Eds.), ICDT ´90. Proceedings, 1990. Vll, 528 pages. 1990.

Vol. 471: B.C. Ooi, Efficient Query Processing in Geographic Information Systems. Vlll, 208 pages. 1990.

Vol. 472: K.V. Nori, C.E. Veni Madhavan (Eds.), Foundations of Software Technology and Theoretical Computer Science. Proceedings, 1990. X, 420 pages. 1990.

Vol. 473: I.B. Damgård (Ed.), Advances in Cryptology – EUROCRYPT ´90. Proceedings, 1990. Vlll, 500 pages. 1991.

Vol. 474: D. Karagiannis (Ed.), Information Syetems and Artificial Intelligence: Integration Aspects. Proceedings, 1990. X, 293 pages. 1991. (Subseries LNAI).

Vol. 475: P. Schroeder-Heister (Ed.), Extensions of Logic Programming. Proceedings, 1989. Vlll, 364 pages. 1991. (Subseries LNAI).

Vol. 476: M. Filgueiras, L. Damas, N. Moreira, A.P. Tomás (Eds.), Natural Language Processing. Proceedings, 1990. Vll, 253 pages. 1991. (Subseries LNAI).

Vol. 477: D. Hammer (Ed.), Compiler Compilers. Proceedings, 1990. Vl, 227 pages. 1991.

Vol. 478: J. van Eijck (Ed.), Logics in Al. Proceedings, 1990. IX, 562 pages. 1991. (Subseries in LNAI).

Vol. 480: C. Choffrut, M. Jantzen (Eds.), STACS 91. Proceedings, 1991. X, 549 pages. 1991.

Vol. 481: E. Lang, K.-U. Carstensen, G. Simmons, Modelling Spatial Knowledge on a Linguistic Basis. IX, 138 pages. 1991. (Subseries LNAI).

Vol. 482: Y. Kodratoff (Ed.), Machine Learning – EWSL-91. Proceedings, 1991. XI, 537 pages. 1991. (Subseries LNAI).

Vol. 483: G. Rozenberg (Ed.), Advances In Petri Nets 1990. VI, 515 pages. 1991.

Vol. 484: R. H. Möhring (Ed.). Graph-Theoretic Concepts In Computer Science. Proceedings, 1990. IX, 360 pages. 1991.

Vol. 485: K. Furukawa, H. Tanaka. T. Fullsaki (Eds.), Logic Programming '89. Proceedings, 1989. IX, 183 pages. 1991. (Subseries LNAI)

Vol. 486: J. van Leeuwen. N. Santoro (Eds.). Distributed Algorithms. Proceedings, 1990. VI, 433 pages. 1991.

Vol. 487: A. Bode (Ed.). Dlstributed Memory Computing. Proceedings, 1991. XI, 506 pages. 1991

Vol. 488: R. V. Book (Ed.). Rewriting Techniques and Applications. Proceedings, 1991. VII, 458 pages. 1991.

Vol. 489: J. W. de Bakker, W. P. de Roever, G. Rozenberg (Eds.), Foundations of Object-Oriented Languages. Proceedings, 1990. VIII, 442 pages. 1991.

Vol. 490: J. A. Bergstra, L. M. G. Feljs (Eds .), Algebraic Methods 11: Theory, Tools and Applicatlons. VI, 434 pages. 1991.

Vol. 491: A. Yonezawa, T. Ito (Eds.). Concurrency: Theory, Language, and Architecture. Proceedings, l989. VIII, 339 pages. 1991.

Vol. 492: D. Sriram, R. Logcher, S. Fukuda (Eds.). Computer-Aided Cooperative Product Development. Proceedings, 1989 VII, 630 pages. 1991.

Vol. 493: S. Abramsky, T. S. E. Maibaum (Eds.). TAPSOFT '91. Volume 1. Proceedings, 1991. VIII, 455 pages. 1991.

Vol. 494: S. Abramsky, T. S. E. Maibaum (Eds.). TAPSOFT '91. Volume 2. Proceedings, 1991. VIII, 482 pages. 1991

Vol. 495: 9. Thalheim, J. Demetrovics, H.-D. Gerhardt (Eds.), MFDBS '91. Proceedings, 1991. VI, 395 pages. 1991.

Vol. 497: F. Dehne, F. Fiala. W.W. Koczkodaj (Eds.), Advances in Computing and Intormation - ICCI '91 Proceedings, 1991. VIII, 745 pages. 1991.

Vol. 498: R. Andersen, J. A. Bubenko jr., A. Sølvberg (Eds.), Advanced Information Systems Engineering. Proceedings, 1991. VI, 579 pages. 1991.

Vol. 499: D. Christodoulakis (Ed.), Ada: The Choice for '92. Proceedings, l991. VI, 411 pages. 1991.

Vol. 500: M. Held, On the Computational Geometry of Pocket Machining. XII, 179 pages. 1991.

Vol. 501: M. Bidoit, H.-J. Kreowski, P. Lescanne, F. Orejas, D. Sannella (Eds.). Algebraic System Specification and Development. VIII, 98 pages. 1991.

Vol. 502: J. Bārzdiņš, D. Bjørner (Eds.), Baltic Computer Science. X, 619 pages. 1991.

Vol. 503: P. America (Ed.), Parallel Database Systems. Proceedings, 1990. VIII, 433 pages. 1991.

Vol. 504: J. W. Schmidt, A. A. Stogny (Eds.), Next Generation Information System Technology. Proceedings, 1990. IX, 450 pages. 1991.

Vol. 505: E. H. L. Aarts, J. van Leeuwen, M. Rem (Eds.), PARLE '91. Parallel Architectures and Languages Europe, Volume I. Proceedings, 1991. XV, 423 pages. 1991.

Vol. 506: E. H. L. Aarts, J. van Leeuwen, M. Rem (Eds.), PARLE '91. Parallel Architectures and Languages Europe, Volume II. Proceedings, 1991. XV, 489 pages. 1991.