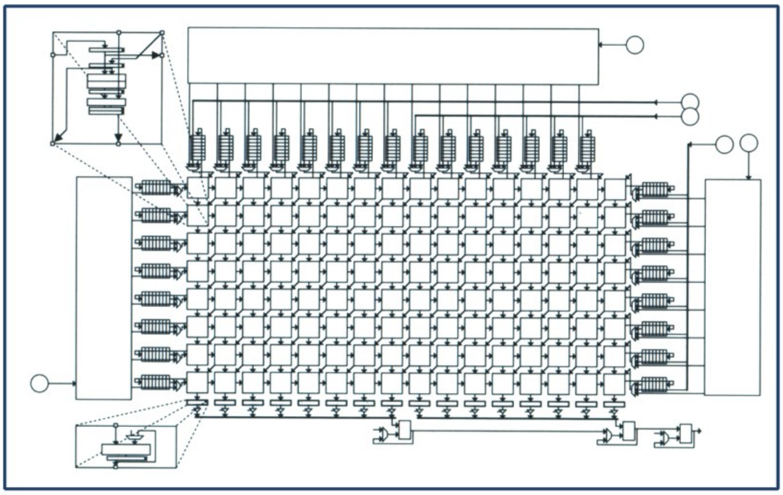
# Application-Driven Architecture Synthesis

edited by

Francky Catthoor
Lars Svensson



Springer-Science+Business Media, LLC

# APPLICATION-DRIVEN ARCHITECTURE SYNTHESIS

# APPLICATION-DRIVEN ARCHITECTURE SYNTHESIS

*edited by*

**Francky Catthoor**
**Lars Svensson**
*IMEC, Leuven, Belgium*

SPRINGER-SCIENCE+BUSINESS MEDIA, LLC

**Library of Congress Cataloging-in-Publication Data**

A C.I.P. Catalogue record for this book is available from
the Library of Congress.

ISBN 978-1-4613-6425-2    ISBN 978-1-4615-3242-2 (eBook)
DOI 10.1007/978-1-4615-3242-2

# CONTENTS

# PREFACE

The main intention of this book is to give an impression of the way current research in high-level and behavioral synthesis for real-time architectures is going. The focus lies especially on domains where application-specific VLSI solutions are attractive, such as significant parts of audio, telecom, instrumentation, speech, robotics, medical and automotive processing, image and video processing, TV, multimedia, radar, and sonar processing.

The material in this book is based on work in the context of two research projects, ASCIS (Architecture Synthesis for Complex Integrated Systems) and NANA (Novel parallel Algorithms for New real-time Architectures), both sponsored by the ESPRIT program of Directorate XIII of the European Commission. The chapters are partly based on material presented at the final project workshops, which took place at IMEC in Leuven, Belgium, on the 29th and 30th of April, 1992, marking the completion of three successful years of project cooperation.

ASCIS and NANA were among the first of the ESPRIT Basic Research Actions. These are different from other ESPRIT projects in two ways: the foreseen industrial application of the results may be as far as five to seven years in the future, and some of the financial arrangements make it easier for universities to participate. For a project to gain ESPRIT sponsorship, it must have partners from at least two different European countries. ASCIS and NANA both met this requirement with ease. The partners of the NANA project were: Delft University of Technology, the Netherlands; École Normale Supérieure de Lyon (ENSL), France; Inter-university Micro-Electronic Center (IMEC), Leuven, Belgium; IRISA, Rennes, France; and Katholieke Universiteit Leuven, Belgium. The ASCIS partners were: Eindhoven University of Technology, Eindhoven, The Netherlands; IMEC, Leuven, Belgium; INPG/TIM3, Grenoble, France; Technical University of Darmstadt, Darmstadt, Germany; Technical University of Denmark, Lyngby, Denmark; Patras University, Patras, Greece; and Lund University, Lund, Sweden. With seven partners in seven countries, ASCIS in particular was a good example of a pan-European research project.

The goal of the hardware synthesis work within these projects has been to contribute design methodologies and synthesis techniques which address the design trajectory from *real behavior* down to the RT-level *structural specification* of the system. In order to provide complete support for this synthesis trajectory, many design problems must be tackled. We do not claim to cover the complete path, but we do believe we have contributed to the solution of a number of the most crucial problems in the domains of specification and synthesis. We therefore expect this book to be of interest in academia; not for detailed descriptions of the research results—these have been published elsewhere—but for the overview of the field and a view on the many important but less widely known issues which must be addressed to arrive at industrially relevant results.

The Ascis and Nana projects have also been application-driven from the start, and the book is intended to reflect this fact. The real-life applications that have driven the research are described, and the impact of their characteristics on the methodologies is assessed. We therefore believe that the book will be of interest to senior design engineers and CAD managers in industry, who wish either to anticipate the evolution of commercially available design tools over the next few years, or to make use of the concepts in their own research and development.

The projects' emphasis on basic research notwithstanding, it must not be forgotten that Esprit is a program which has the goal to support industry through research. It is therefore important to note that some of the Ascis and Nana results already have generated interest from European CAD and systems industry. The continued research by the partners, some of it in the context of other Esprit projects, obviously also benefits from the results described in this book. In addition, a follow-up project for Nana (Nana-2) has been running for more than a year, and a proposal for an Ascis follow-up is being prepared.

It has been a pleasure for us to work within the projects. The coordination of the work has meant many hours on the phone, in airplanes and airports, and behind computer keyboards. However, we consider ourselves amply rewarded: in addition to learning many new things about behavioral synthesis and related issues, we have also developed close connections with excellent people at each of the partner sites. Moreover, the pan-European aspect has allowed us to come in closer contact with research groups with a different background and "research culture," which has led to very enriching cross-fertilization.

We would like to use this opportunity to thank the many people who have helped make these projects successful, and to express our appreciation of their contributions:

- The main authors of the chapters of this book, who were among the most active participants in the synthesis-related technical work within the projects.

- The members of the Scientific Advisory Boards for the projects, whose direction and high-level steering has been an important contribution to their success and indirectly to the results in this book: Prof. Bernard Courtois, Prof. Hugo De Man, Prof. Patrick Dewilde, Prof. Jochen Jess, Prof. Manfred Glesner, Prof. Costas Goutis, Prof. Ole Olesen, Prof. Lars Philipson, Prof. Patrice Quinton, Prof. Yves Robert, and Prof. Joos Vandewalle.

- Our Project Officer at the European Commission, Dr. Klaus Wölcken, for his support and enthusiasm.

- Our technical and administrative coordination staff at IMEC, for their support with the financial and practical details of project management that researchers are typically hopelessly inept at: Patrick Pype, Joost Deseure, and Annemie Stas.

- Research associates in many countries, who contributed to the progress of these projects; in particular, we wish to mention Per Andersson, Florin Balasa, Abdelhamid Benaini, Henri-Pierre Charles, Gjalt De Jong, Ed Deprettere, Michael Held, Jürgen Herpel, Thomas Hollstein, Holger Jürgs, Jian-Jin Li, Shen Li-Sheng, Fang Longsen, Christophe Mauras, Serge Miguet, Henrik Pallisgaard, Wim Philipsen, Yannick Saouter, Stéphane Ubéda, Alle-Jan van der Veen, Sabine Van Huffel, Steven Van Leemput, Ingrid Verbauwhede, and Claus Vielhauer.

- Last but not least: Jan Rosseel, whose LaTeX expertise proved invaluable in the production of this book.

We finally hope that the reader will find the book useful and enjoyable, and that the results presented will contribute to the continued progress of the field of high-level and behavioral synthesis.

*Leuven, Belgium* Francky Catthoor

*Santa Monica, California* Lars Svensson

# CONTRIBUTORS

**Michael Birbas**
VLSI Design Lab.
Dept. of Electrical Engineering
University of Patras
Patras 26110, Greece

**Jens P. Brage**
Dept. of Computer Science
Technical University of Denmark
Building 344
DK2800 Lyngby, Denmark

**Francky Catthoor**
VLSI Systems Design Methodology Dept.
IMEC
Kapeldreef 75
B3001 Leuven, Belgium

**Bernard Courtois**
TIM3/INPG
46 av. Felix Viallet
38031 Grenoble Cédex, France

**Alain Darte**
Laboratoire de l'Informatique
du Parallélisme—TIM3
École Normale Supérieure de Lyon
46, Allée d'Italie
69364 Lyon Cédex 07, France

**Hugo De Man**
VLSI Systems Design Methodology Dept.
IMEC
Kapeldreef 75
B3001 Leuven, Belgium

**Ed Deprettere**
Delft University of Technology
Mekelweg 4
2600 GA Delft, The Netherlands

**Patrick Dewilde**
Delft University of Technology
Mekelweg 4
2600 GA Delft, The Netherlands

**Frank Franssen**
VLSI Systems Design Methodology Dept.
IMEC
Kapeldreef 75
B3001 Leuven, Belgium

**Werner Geurts**
VLSI Systems Design Methodology Dept.
IMEC
Kapeldreef 75
B3001 Leuven, Belgium

**Manfred Glesner**
Technische Hochschule Darmstadt
FG Mikroelektronische Systeme
Karlstr. 15
D-6100 Darmstadt
Germany

**Costas Goutis**
VLSI Design Lab.
Dept. of Electrical Engineering
University of Patras
Patras 26110, Greece

**Peter Held**
Delft University of Technology
Mekelweg 4
2600 GA Delft, The Netherlands

**Geert Janssen**
Dept. of Electrical Engineering
Eindhoven Univ. of Technology
P.O. Box 513
5600 MB Eindhoven, The Netherlands

**Ahmed Amine Jerraya**
TIM3/INPG
46 av. Felix Viallet
38031 Grenoble Cédex, France

**Jochen Jess**
Dept. of Electrical Engineering
Eindhoven Univ. of Technology
P.O. Box 513
5600 MB Eindhoven, The Netherlands

**Efstathios Kyriakis-Bitzaros**
VLSI Design Lab.
Dept. of Electrical Engineering
University of Patras
Patras 26110, Greece

**Hervé Le Verge**
IRISA-CNRS-INRIA
Campus de Beaulieu
35042 Rennes Cédex, France

**Jan Madsen**
Dept. of Computer Science
Technical University of Denmark
Building 344
DK2800 Lyngby, Denmark

**Kevin O'Brien**
TIM3/INPG
46 av. Felix Viallet
38031 Grenoble Cédex, France

**Ole Olesen**
Dept. of Computer Science
Technical University of Denmark
Building 344
DK2800 Lyngby, Denmark

**Vasilis Paliouras**
VLSI Design Lab.
Dept. of Electrical Engineering
University of Patras
Patras 26110, Greece

**Inhag Park**
TIM3/INPG
46 av. Felix Viallet
38031 Grenoble Cédex, France

**Lars Philipson**
Dept. of Comp. Engineering
Lund University
P.O. Box 118
S-221 00 Lund, Sweden

**Peter Pöchmüller**
Technische Hochschule Darmstadt
FG Mikroelektronische Systeme
Karlstr. 15
D-6100 Darmstadt
Germany

**Patrice Quinton**
IRISA-CNRS-INRIA
Campus de Beaulieu
35042 Rennes Cédex, France

**Kenny Ranerup**
Dept. of Comp. Engineering
Lund University
P.O. Box 118
S-221 00 Lund, Sweden

**Tanguy Risset**
Laboratoire de l'Informatique
du Parallélisme—TIM3
École Normale Supérieure de Lyon
46, Allée d'Italie
69364 Lyon Cédex 07, France

**Yves Robert**
Laboratoire de l'Informatique
du Parallélisme—TIM3
École Normale Supérieure de Lyon
46, Allée d'Italie
69364 Lyon Cédex 07, France

**Jan Rosseel**
VLSI Systems Design Methodology Dept.
IMEC
Kapeldreef 75
B3001 Leuven, Belgium

**Dimitris Soudris**
VLSI Design Lab.
Dept. of Electrical Engineering
University of Patras
Patras 26110, Greece

**Thanos Stouraitis**
VLSI Design Lab.
Dept. of Electrical Engineering
University of Patras
Patras 26110, Greece

**Lars Svensson**
VLSI Systems Design Methodology Dept.
IMEC
Kapeldreef 75
B3001 Leuven, Belgium

**Jos van Eijndhoven**
Dept. of Electrical Engineering
Eindhoven Univ. of Technology
P.O. Box 513
5600 MB Eindhoven, The Netherlands

**Michaël van Swaaij**
VLSI Systems Design Methodology Dept.
IMEC
Kapeldreef 75
B3001 Leuven, Belgium

**Paul Wielage**
Delft University of Technology
Mekelweg 4
2600 GA Delft, The Netherlands

**Norbert Wehn**
Technische Hochschule Darmstadt
FG Mikroelektronische Systeme
Karlstr. 15
D-6100 Darmstadt
Germany

**Klaus Wölcken**
Commission of the E.C.
Rue de la Loi 200
B1049 Brussels, Belgium

# APPLICATION-DRIVEN ARCHITECTURE SYNTHESIS

# APPLICATION-DRIVEN SYNTHESIS METHODOLOGIES FOR REAL-TIME PROCESSOR ARCHITECTURES

**Francky Catthoor**[1], **Lars Svensson**[1]
**Klaus Wölcken**[2]

[1] *IMEC, Leuven*
[2] *Commission of the E.C., Brussels*

## ABSTRACT

In this chapter, we present an overview of the objectives and key achievements of the work described further in the book. Our topic is the synthesis of application-specific architectures and their realizations, targeted to microelectronic products involving real-time processing. Engineers who design real-time systems experience two classes of difficulties. One is methodical: there is no good design strategy to bring a concept to an effective realization. The other is practical: it is hard to produce a cost-effective, customized architecture for a given throughput or latency. In the course of the last four years, the contributing authors of this book have cooperatively addressed these bottlenecks. We propose a set of solutions that form part of a complete methodology for the target domain of dedicated processors for real-time systems. All the work has been driven by real-life case studies. To demonstrate our methods, a number of these realistic examples are treated in the subsequent chapters. The work has been performed in the context of two highly successful ESPRIT basic research actions: ASCIS ("Architecture Synthesis for Complex Integrated Systems") and NANA ("Novel parallel Algorithms for New real-time Architectures").

## 1 PROBLEM DESCRIPTION

State-of-the-art real-time signal and data processing applications typically involve a rapidly increasing arithmetic complexity. In many cases, this is combined with a need for flexible and powerful decision-making. Furthermore, they not only involve word-oriented processing but also employ advanced vector and matrix operations [1]. Loops are abundant and sometimes non-parallelizable, which leads to computational bottlenecks. This large application domain ranges from audio and speech processing, user-end (portable) telecommunication, and automotive applications, all of which exhibit low to medium throughputs, up to HDTV, image, video, and radar processing, which require much higher sample rates. In addition to such "hard" real-time applications, other data-intensive tasks that require fast execution also increase in importance. Some examples are real-time graphics and medical and other types of data acquisition.

All of these applications have in common that they are difficult to realize on general-purpose hardware [7], when an efficient solution—in terms of power or in terms of area—is required. Such realizations result in considerable overhead in terms of memory requirement, I/O bandwidth, I/O interfaces, and controller cost. Furthermore, there is typically a waste of cycles on the general-purpose rigid data-paths. Similar arguments apply largely for MIMD assemblies of general-purpose processors, even for the current generations of processors oriented towards parallel processing, such as Transputers or the TMS320C40 series. In some cases, the design time for these implementation alternatives can be small. For complex algorithms, though, the mapping stage may require many iterations and time-consuming manual tuning, just like when custom hardware is used. The major advantage is obviously the flexibility to make last-minute changes. This is especially interesting during algorithm development and prototyping, when the algorithm is subject to continuous refinement. However, when the system has reached the production stage and especially in the more mature product generations, we believe special-purpose or application-specific system realizations can be heavily motivated. In particular, this is the case if the application requires that the implementation exhibits low power consumption or a small physical size, or if the production volume is reasonably high, as is the case for the important markets of portable communication equipment and consumer electronics products like HDTV and videophone.

It has to be stressed that the term *application-specific* in this context implies few assumptions about the target implementation technology. The main requirement is that the technology permits the adaptation of the global architecture, in terms of primitive building blocks like adders, register files, and boolean operators, to the specific needs of the application. Hence, the complete range from full-custom design over macro-cell and/or standard-cell based approaches, sea-of-gates technologies, and field-programmable devices all the way to off-the-shelf building blocks can be the target.

## 2 STATE-OF-THE-ART AND BEYOND

The development cost of a complex application-specific solution can be very high, especially due to the typically long design cycle. Production cost will depend largely on the target technology chosen: small volumes will motivate the use of FPGAs, whereas large volumes favor the use of predefined macro-cells, and intermediate situations may require intermediate solutions. Therefore, much more effort should be spent on reducing the design-time-related cost. Moreover, as a consequence of this long design time, the time-to-market can be unacceptable. This is of major importance in the rapidly changing markets of today. In almost all cases, the long design cycle is heavily dominated by the time-consuming iterations due either to changes in the specifications which affect many design stages, or to late detection of incompatibilities between subcomponent designs.

State-of-the-art industrial CAD tools typically support only the lower end of the design path, i.e., structural synthesis from a structural specification down to the actual implementation [13]. In addition, the register-transfer and functional (combinatorial) levels in the design trajectory have become increasingly mature in the past five years, both in academia and independent research labs (e.g., IBM, U.C. Berkeley, University of Colorado at Boulder, Stanford, and INP Grenoble) and recently also in industry (e.g., Synopsys, Cadence, Mentor, Racal-Redac, Siemens, and Philips). Support is mainly restricted to the register-transfer-level scheduling and allocation of scalar signals to individual registers, from simple operations to basic RT-level operators and from individual transfers to multiplexers and connections.

The dominant part of the design trajectory above register-transfer level is still largely unsupported. One of the reasons for this is the huge amount of dedicated architectural options available to the designer [1, 32, 40, 23, 6]. Moreover, for automatic synthesis on the higher abstraction levels to be a viable alternative to hand design, especially in the consumer or domestic markets, a sufficiently large design efficiency has to be achieved. The results should approach those of manual designs, in terms of throughput, physical size, power consumption, and packaging. Therefore, currently emerging solutions are largely restricted to specification and co-simulation environments based on standards like VHDL or other widely distributed hardware description languages like the Verilog language. Usually, the description level applied is still very close to the RT-level structure. Moreover, *all* design decisions must be taken manually, even those that are less critical for meeting the design constraints, very error-prone, or very time-consuming.

In order to really solve the design-time bottleneck for real-time system design, this is not sufficient. There is a clear need for *higher level design support,* including *synthesis tools at the architectural or behavioral (system) level* which include *power- and area-efficient automated techniques.* The tools should still be fully controllable by the designer, so *user interaction* is another key issue. We believe that in order to achieve this ambitious goal, we need [9]:

- A range of efficient *target architecture styles* architecture style underlying the synthesis strategies. This is the only way to guarantee a sufficient coverage of the design space without losing efficiency. Dedicated regular arrays and application-specific multiplexed processors are two key target styles that are addressed in this book.
- Domain-specific *synthesis tools* which fully exploit the characteristics of the target architecture style and the corresponding application domain. The tools will frequently differ for the alternative target styles. For instance, a scheduler that is efficient for one style may be unsuitable or inefficient for another.

This has not been widely realized until about 1991, even if some of the issues had been addressed earlier. A detailed comparison with these approaches is provided below in sections 5 and 6 and in the subsequent chapters.

## 3 CONTRIBUTION OF THIS BOOK

The goal of the work on architecture synthesis within the ASCIS and NANA projects has been to contribute design methodologies and synthesis techniques which address the design trajectory from *real behavior* down to the RT-level *structural specification* of the system. Our view of this synthesis process is illustrated in figure 1.

The work described in this book specifically addresses the following topics, which we believe will have a substantial impact on future application-specific design methodologies:

- Refined system specification models that have embedded in them the practical requirements of architecture synthesis and verification methods. This is opposed to the simulation-oriented semantics of most of the widely used specification formalisms, such as VHDL; still, support for a VHDL subset has been investigated.

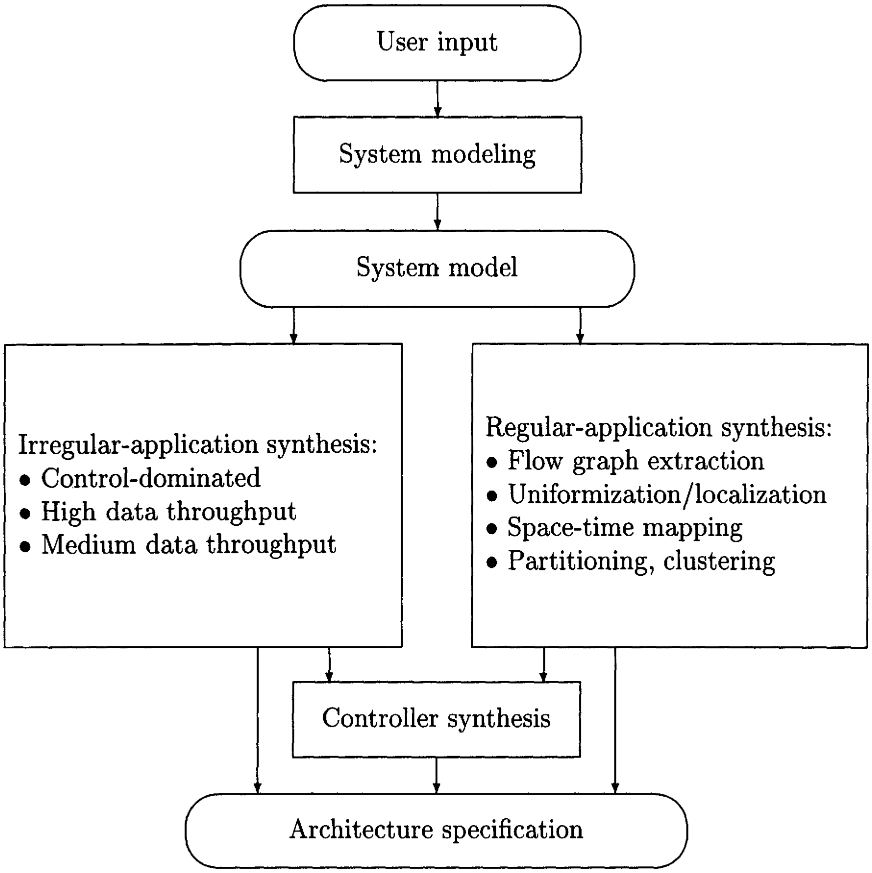


**Figure 1** Architecture synthesis as viewed within the ASCIS and NANA projects. The input from the user is converted to a formal system model which can also be written out in a readable specification format. Different design trajectories must be followed, depending on the characteristics of the application. The figure only shows the two main branches we have addressed. The final high-level performance-driven controller synthesis is similar for both branches. In the next design stages, which have not been addressed by the projects, the detailed synthesis of the data-paths and controllers on the RT and logical levels still has to take place. This is then followed by the physical design stage.

- Novel design methodologies and synthesis techniques for regular array processor architectures intended for regular, high-throughput data-flow applications.
- Novel design methodologies and synthesis techniques for multiplexed processor architectures intended for irregular high- and medium-throughput applications. Both control-flow- and data-flow-dominated target domains are treated.
- Performance-driven controller synthesis which makes high throughput possible by overcoming part of the control bottleneck.

In order to provide complete support for the high-level synthesis trajectory, many design problems must be addressed. We do not claim that the material presented in this book covers the complete path, and certainly not all target application domains or architecture styles. However, we do believe we have contributed to the solution of a number of the most crucial problems in the domains of architecture synthesis and the related issue of behavioral specification.

Moreover, the research effort of the ASCIS and NANA partners has been planned with real applications in mind. Real-life drivers have been selected to steer the development of the required methodologies and techniques. Some of the main drivers will also be used as real-life demonstrator applications in the following chapters.

The extent of each contribution by the partners and how they fit in a more global design context will be summarized in somewhat more detail in the subsequent sections. Most of this work is also reflected in the subsequent chapters. The structure of the book and the links between the other chapters will be described in section 7.

## 4 SYSTEM SPECIFICATION MODEL

Any synthesis method has to start from an initial specification, described in terms of a model which has to meet a number of requirements. In particular, there is clear need for:

- A specification model that explicitly contains the information necessary to extract the desired functional behavior *and* the timing requirements in our real-time application domain.
- The means to establish relations between the different elements or kernels of the model while synthesis progresses. For example, flow-graph nodes (operations) eventually have to be associated with structure nodes (operators).

A global reduction of design time is only possible if the aspect of verification is addressed. Leaving errors in a design until production is, of course, a disaster: it will cause large extra costs to locate the error, correct it, and restart the production process. Avoiding these design iterations by simulation alone has the well-known problems of an incomplete check, a prohibitively large computational load, and a difficult interpretation of the huge amount of simulation responses. For these reasons, *formal verification* has been addressed by several research institutes. Here, the goal is to formally prove that the system will behave well in all circumstances.

To make formal verification possible, there are again several prerequisites on the specification model:

- The model has to consist of formally defined elements. This allows formal reasoning about its behavior. The models commonly used in simulation programs are not strong enough for this.
- A partial or full description of the desired behavior of the system must be available, again expressed using mathematical formalisms. This mathematical specification should be extracted from a more user-friendly input, so that the designer does not have to learn complex formalisms.

Most of the current design models are not suitable for the real-time signal processing domain and do not deal with the above requirements. At the start of the ASCIS project, there was a clear need to fill this gap. This has led to the ASCIS data flow-graph (DFG) model described in chapter 2. It differs from most other data-flow approaches by a uniform embedding of conditional and loop constructs, allowing, for instance, the specification of conditional or repeated I/O and giving maximal opportunity for system optimizations.

A number of other design models have been proposed recently: the SIL model in the context of the SPRITE (ESPRIT 2260) project, involving Philips and its partners [21]; the DSFG/LIB models in the CATHEDRAL project at IMEC [25]; and the RLEXT design model from the University of Illinois [20] come to mind. However, the RLEXT model addresses mostly RT-level issues and not the true behavioral synthesis models. The other models mentioned have been established in close cooperation with the ASCIS work.

The resulting refined DFG format has been adopted as a common specification format by all the ASCIS and NANA partners. Moreover, initiatives have been taken to come to a standardization in this domain together with several other European partners.

In addition, effort has been spent on formal verification itself, especially regarding timing issues for control-flow-dominated systems. A study of propositional temporal logic has shown its feasibility to verify correctness of sequential logic circuits, CMOS transistor circuits, and finite state machine diagrams against each other [16]. This work has been received with great interest in the world-wide research community, since verifications were done that previously seemed untractable. It has been applied in the context of performance-driven controller synthesis, as described in chapter 10.

Verification has also been performed directly on the DFG [17]. This is a highly interesting perspective, since data flow graphs are widely used as the starting point for synthesis. The verification here is a more global data-independent scope, allowing the verification of much larger systems at the cost of a reduced resolution: detailed, data-dependent functional behavior is not modeled.

It is also important to provide a conversion from existing specification languages to the DFG model. For this purpose, a VHDL modeling scheme has been developed [27]. This allows to represent the DFG as a subset of VHDL and vice versa. Also, conversions from other languages that cover important application domains have been investigated, namely HARDWAREC [22] and SILAGE [15]. More information is provided in chapter 2.

## 5 SYNTHESIS OF ARRAY PROCESSORS

As the complexity of what can be integrated on a single die or in a multichip module is increasing tremendously, it becomes more and more attractive to implement special-purpose architectures using highly modular parallel algorithms [36]. In that case, the modularity inherent in the algorithms should be reflected in a regular architecture with one- or two-dimensional (1-D or 2-D) repetitivity in the arrangement of processing elements, local communication, and distributed local storage. Examples of such regular organizations are systolic or wavefront arrays, which are typically defined as regular networks of locally interconnected elementary processors [23]. In order to avoid a long critical path spanning several processors, intensive pipelining is used between the processors, leading to the term "systolic." Reasons to consider this regular array style include:

- *High performance* due to the systematic use of parallelism and pipelining.
- *Decreased design time* obtained by implementing only a few elementary cells which are then replicated many times.
- *Design modularity,* i.e., the possibility of using the same set of cells for the various values of some size parameter of the algorithms.
- *Simplification of test and fault-tolerance issues*, as the complexity of the analysis is broken by the regularity of the design.
- *Potentially lower power consumption.* Regular parallel architectures permit trading off clock speed and parallel computation in a much more flexible way than irregular ones. This issue is important when power consumption is to be minimized.

An important driver which has influenced several of the subtasks in the array synthesis methodologies in this book is the solution of the algebraic path problem (APP) which is occurring frequently in mathematical equation solvers and in data analysis. This driver will be discussed in detail in chapter 3. Several crucial subtasks will be identified, and the results for several architectures optimized for the APP problem will be described. For many of these subtasks, a link will be made with the synthesis techniques and tools necessary to address these issues.

Research on formal methods for the design of array architectures has been going on for about 10 years [35, 29]. Briefly, these methods consist of transforming an iterative specification of the desired algorithm into the specification of an architecture, by mapping the initial iteration space onto a new "space-time" space, composed of the processor number space and the execution time. The majority of the approaches has concentrated on pure systolic arrays [2, 8, 43, 11, 23, 29, 36, 37], as is also evident from the references of chapters 6 and 4. Moreover, most of the techniques reported until now are based on either heuristic transformations applied in a user-defined sequence, or on linear or affine transformations of the index space and dependence vectors of an iterative algorithm that is already uniformized. The heuristic transformations of the first class are executed in an unformalized, ad-hoc way. This leads to an architecture mapping which is very sensitive to user input. Most of the affine transformations are only dealing with part of the mapping trajectory, namely starting from a level usually referred to as uniform recurrence equations (UREs) [18].

In addition, most array architecture design methods deal either with the design of non-real-time systems or with programmable arrays composed of rigid PEs, mostly composed with off-the-shelf components. However, the area of fully customized designs for real-time signal and data processing subsystems is important for industry. We believe this is not addressed in a sufficiently effective way by the existing techniques, as indicated in chapters 6 and 4.

Hence, we have felt the need for several extensions to the design trajectory:

- The initial specification should be at a higher level than the conventional level of uniform recurrence equations (UREs). A "front end" is needed to build the dependence graph from a more user-friendly specification of the algorithm. Several specification languages have been proposed here. Procedural models with FORTRAN-style DO loops are used in chapters 4 and 5 for data processing applications, involving complex data updates in the algorithm that are more difficult to express without the concept of "variables." In contrast, functional languages like ALPHA [3] and SILAGE [15] are introduced in chapter 6 for more regular signal processing applications that may be elegantly described in single-assignment form. For all these languages, a conversion is needed to some type of dependence graph. For the functional languages this is relatively simple, but for the procedural languages, novel techniques have been developed to achieve this goal.

- The gap between high-level behavioral descriptions and the URE level has to be bridged with formalized high-level transformations on the signal flow graphs. These involve mainly reindexing to arrive at a set of affine recurrence equations in a single domain [44] and localization for dealing with broadcast and global operations [43]. At this stage, the possibility of interactive guidance is also of crucial importance [3]. This topic will be addressed in more detail in chapter 6.

- The properties of real-time signal and data processing applications have to be exploited to arrive at fully efficient application-specific architectures. This requires extensions to the basic linear space-time mapping methods [44] as proposed in chapters 3 and 6. In addition, an alternative affine space-time transformation method based on the existence of independent subsets in the index space is introduced in chapter 5. This method has led to efficient arrays with high hardware utilization [24].

- Both word-level and bit-level parallelism should be exploited to reach the extreme throughputs which are needed in particular applications like radar processing [33]. Extensions in this direction are introduced in chapter 5. Employing the features of various arithmetic systems, such as 2's complement and residue number systems (RNS) [42], bit-level systolic arrays can now be derived automatically.

- After space-time mapping, the conditional execution of several operations on the same processing element (PE) may be needed. Regularization techniques to address this problem are presented in chapter 4.

- In order to match the required throughput or the required array size, it is necessary to partition the initial architecture with full index ranges into a set of serially executed subsets and then to cluster these onto a smaller sized array. This important task has been addressed in depth [5] and is described in chapter 4. Extensions which can result in a better preservation of the regularity, especially for latency-limited applications, are proposed in chapter 3.

The comparison of our approaches with some other techniques, which do address particular issues that we feel to be vital for this domain, will be presented in more detail in the subsequent chapters.

Some of the applications studied during the development of the array processor design methods will be described along with the methods themselves. The applications range from so-called "kernel algorithms" (matrix computation, filters in one and two dimensions, dynamic programming, singular value decomposition, etc.) to real-life applications such as image analysis, video coding, and computer graphics algorithms.

## 6 SYNTHESIS OF MULTIPLEXED PROCESSORS

A second important architecture style for real-time signal processing systems is the multiplexed processor style, characterized by a set of application-specific, time-multiplexed data-paths steered by a hierarchically organized controller. This style is tuned to irregular applications, requiring a medium to high sample rate (10 kHz – 10 MHz). Many applications at these rates require a combination of computation-intensive arithmetic and complex decision-making operations. For these applications, the array style (as described in section 5) is unsuitable.

### 1 Control-flow-dominated processors

The multiplexed processor style spans a large complexity range, both for the data-paths and for the controller. An algorithm with a small amount of data flow with mainly operations on scalar data, but with a complex mix of nested loops, data-dependent iterations (including global exceptions and unconditional loop exits), and nested conditions, is characterized as control-flow-dominated. These algorithms abound in embedded control applications and in some telecommunication systems. They require a complex controller with a relatively simple but programmable data-path. Much effort has been spent on synthesis techniques for this important target application domain [49, 30]. However, several issues have not yet been fully addressed. In particular, support for a flexible building block (BB) library that not only contains standard BBs—such as ALUs—but also complex devices with as yet unknown clock cycle delays—such as cache memories or I/O processors—is necessary for a system to be useful for real-life designs. Efficient algorithms for allocation and scheduling tuned to these extensions are needed, too [31]. Also, interactive optimization capabilities should be stressed. Research to find a solution for these problems has led to the AMICAL environment [12] which is described in chapter 9.

### 2 Data-flow-dominated processors

In domains like automotive applications and back-end speech or audio processing, the data-flow issues are more dominant than the control flow. Then, there is a need for another target style with more emphasis on the data-paths and the storage. However, due to the medium sample rates, the data-paths will still need to be highly multiplexed and thus partly programmable in order to share hardware as efficiently as possible. Synthesis techniques for related target domains have been studied in the past [41, 38, 25]. However, in many cases there is a need to map such applications, which are not time-critical, onto pre-defined architectures for prototyping purposes. In this context, solutions based on field-programmable gate arrays (FPGAs) have become especially attractive lately. This style has been addressed using applications from the automotive field [47, 48]. The memory organization, the mapping onto the fixed data-path, and the connection network and scheduling have been the focuses here. The results on this highly multiplexed processor synthesis are described in chapter 8.

When mostly data-flow computations have to be performed combined with a limited control flow in a limited sample period, this calls for highly customized, pipelined, and more complex data-paths to solve the timing-related bottlenecks. This lowly multiplexed processor style is especially suitable for irregular image and video processing, user-end telecom, and front-end audio or speech processing subsystems, where the operations must be performed partly in parallel [6]. Dealing with these subsystems involves synthesis subtasks [25] that have not been addressed so far in existing synthesis approaches [26], such as how to map multidimensional signals in an effective way on distributed memory organizations. Most of the effort has gone into reorganizing the control flow for a given data flow [45]. In addition, the definition of the dedicated pipelined data-path organization has been addressed [14]. This has led to a complete lowly multiplexed processor synthesis approach, which is further discussed and compared to the state of the art in chapter 7.

In application domains like computer graphics, the iterative constructs for the data-flow-dominated algorithms are data-dependent. As a result, analysis and architecture synthesis techniques need to be developed for iterative constructs with potentially unbounded iterator ranges. In contrast to the situation in fixed-period signal processing where the *worst-case* response time has to meet the sample period criterion, now the *average* throughput is the important performance parameter. This problem has been addressed in part within ASCIS [4], but the results are not yet incorporated in this book.

The basic techniques to perform scheduling and assignment tasks are also important items. Although good heuristics exist for many specific problems, there is also a general demand for generic global optimization techniques to efficiently support the synthesis process. Simulated annealing [19] is a good example of such a technique, which has proven to be very useful. However, the complexity of the cost function that is evaluated for every potential step in the solution space and the resulting large computation time restrict the use of this technique in efficient interactive synthesis environments.

In recent years, new optimization methods have been published, promising general applicability with less computational complexity than simulated annealing. Two such new techniques have been investigated in the context of our projects:

- Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. Promising results in the area of scheduling for a given allocation have been reached [46].
- Optimization based on neural network solvers has been investigated, too. This technique has been applied with good results to the assignment problem as it occurs during lowly multiplexed processor synthesis [34].

Both of these techniques are characterized by their potential for massive parallelism. This aspect is very promising for their future use in the increasingly parallel computing environments.

## 3 Performance-driven controller synthesis

Designing high performance processors requires not only high performance data-paths but also fast and efficient controllers. An appropriate controller architecture is needed, as well as an efficient physical implementation.

Controller synthesis has been the subject of a very large amount of effort both in academia and industry (see section 2). However, until recently, the aspect of performance-driven manipulations has been largely neglected. Therefore, within the ASCIS context, effort has been devoted to filling this gap, even though this is not strictly within the high-level architectural synthesis domain as we define it. The results are described in chapter 10.

First of all, work has been performed on the optimization of the control unit architecture. One particular architectural feature that has been investigated is the subroutine mechanism. Subroutines enable the reuse of hardware resources; this could significantly reduce the area of the controller while also providing a shorter critical delay path. An algorithm has been developed that automatically transforms a controller specification to exploit subroutines in an efficient way [39]. Comparison to hand-made designs has shown that the algorithm in many cases produces results comparable to manual design.

In addition, effort has been spent on the optimizations related to technology mapping. In contrast to the majority of the approaches in literature until recently, the focus has been on performance-driven mapping onto compiled functional cells [28].

## 7 CHAPTER OVERVIEW

In order to provide a clear overview of the work described in this book and the way in which the other chapters are linked, they will be listed below with a short description of their content. The intention in the chapters of this book has *not* been to describe the novel synthesis techniques in detail. Many results have already been published in conferences and journals which deal with these techniques. The most important publications are listed in the bibliographies of the individual chapters. Instead, most of the emphasis will be placed on the high-level methodologies and on how these are inspired by the targeted application domains. In addition, most results are illustrated with a realistic demonstrator application. This will permit easy assessment of the power and the impact of the obtained results. It will also clarify the current status of the realization of the synthesis tools and environments.

### Behavioral system modeling

- "Behavioral specification for synthesis"

  This chapter summarizes the basic model concepts that are used for the initial specification, but that are also highly tuned towards the requirements of the subsequent architecture synthesis steps. In addition, the link to behavioral specification languages used within the synthesis environments is made, with emphasis on a VHDL subset.

### Synthesis methodologies for regular applications

- "Formal methods for solving the algebraic path problem"

  This chapter concentrates on one particular application and the methods needed to effectively tackle the architecture design. An historical overview of the evolution in the design methodologies and their effectiveness permits assessment of the significant progress made in this area. From this overview, the need for support of a number of synthesis tasks is emphasized. Many of the necessary techniques and tools are described in the other chapters.

- "HiFi: from parallel algorithm to fixed-size VLSI processor array"

  This chapter concentrates on methods suitable for extracting regular data-flow-dominated applications from a procedural specification, on methods for dealing with the conditional control flow resulting from mapping different operations on the same processor element, and on mapping techniques onto regular arrays with a fixed array size. Moreover, the necessary design models at different levels in the trajectory are discussed, and a uniform solution is described. A Floyd-Steinberg algorithm for use in document manipulation for a digital copier application is used as a demonstrator.

- "On the design of two-level pipelined processor arrays"

  This chapter concentrates on methods suitable for extracting the uniform data-flow for regular applications expressed with procedural DO loops. Synthesis techniques supporting the important extension to bit-level arrays are also discussed. This makes it possible to design arrays with a very high throughput, where all the algorithmic parallelism is exploited. A 1-D convolution algorithm which is a subsystem in many real-time processing applications is used as a demonstrator in this chapter.

- "Regular array synthesis for image and video applications"

  This chapter concentrates on space-time methods suitable for dealing with many broadcast localization alternatives and for mapping regular, data-flow-dominated applications onto regular arrays with complex processing elements. This is linked with a complete transformation environment that supports these array mapping techniques. Also, the need for extending the level of the initial specification to real behavior is motivated, including the requirements on additional synthesis techniques to fill the resulting gap in specification. A complete motion estimation subsystem from a local area network video coding application is used as a demonstrator.

## Synthesis methodologies for irregular applications

- "Memory and data-path mapping for image and video applications"

  This chapter concentrates on methods suitable for mapping high-throughput, data-flow-dominated applications onto fully application-specific, lowly multiplexed processor architectures. The emphasis lies on the techniques for the memory management needed to deal with the many multidimensional signals, and techniques for the organization of the complex customized data-paths. An updating singular-value decomposition algorithm for data acquisition and a video format conversion application are used as demonstrators.

- "Automatic synthesis in mechatronic applications"

  This chapter concentrates on methods suitable for mapping medium-rate, data-flow-dominated applications onto programmable (FPGA-based) architectures for rapid prototyping purposes. The emphasis lies on the highly multiplexed target architecture and on important steps which need to be dealt with, such as memory organization, data and control flow transformations, and scheduling/binding. A differential heat release computation subsystem in an automotive application is used as a demonstrator.

- "Synthesis for control-flow-dominated machines"

  This chapter concentrates on methods suitable for control-flow-dominated applications. A flexible library of complex building blocks, including predefined I/O interfaces and memories, is targeted. Emphasis is placed on scheduling and allocation techniques tuned to this domain and on the interactive environment in which the tools are embedded. The demonstrator of this chapter is a complete telephone answering machine controller.

## Synthesis methodologies for real-time controllers

- "Controller synthesis and verification"

  This chapter addresses the performance-driven issues in high-level controller design. In particular, effective ways to decompose the controller architecture are presented. The effect of compiled cells on performance-driven technology mapping is also discussed. Finally, it provides results on formal verification aspects in a real-time context.

## 8 CONCLUSION

This book provides an overview of state-of-the-art work in the domain of behavioral and architectural synthesis for real-time processing applications. The contributions are based on work carried out within the ASCIS and NANA research projects from mid-1989 to mid-1992. In this introductory synopsis, we have motivated that specific requirements to solve some crucial design bottlenecks are not met with existing design automation support. In particular, specific characteristics of real-time processing were not at all or not sufficiently exploited until now. In the subsequent chapters, several major contributions will be proposed to this important domain for future systems industrial needs, especially in the important markets of portable user-end telecommunication equipment, multimedia support, automotive processing, and consumer electronics such as HDTV, videophone, and digital audio broadcasting. We believe a transfer of these new technologies to development-oriented projects, including large systems companies, will help to shape the future systems and design automation industry in Europe and abroad.

To a large extent, the success of our work in this domain has been due to the excellent cooperation between the partners, both in the context of formal exchanges between the different tasks and by informal contacts at the frequent meetings and workshops. We believe that this cross-fertilization between the partners is the main added technical value of the ESPRIT type of basic research actions. The three years of cooperation has definitely led to an increase of our research productivity.

## REFERENCES

[1] J. Allen. Computer architecture for digital signal processing. *Proc. of the IEEE*, 73, number 5, pages 854–873, May 1985.

[2] J. Annevelink and P. Dewilde. HiFi: A functional design system for VLSI processing arrays. *Proc. IEEE International Conf. on Systolic Arrays*, San Diego, pages 413–452, May 1988.

[3] A. Benaini, P. Quinton, Y. Robert, Y. Saouter, and B. Tourancheau. Synthesis of a new systolic architecture for the algebraic path problem. *Science of Computer Programming*, 1989.

[4] J. P. Brage. Hardware description languages for synthesis: problems and possibilities. In *Proc. of the tenth NORCHIP Seminar '92*, Helsinki, Finland, pages 22–29. Nov 1992.

[5] J. Bu and E. Deprettere. Processor clustering for the design of optimal fixed-size systolic arrays. *Algorithms and Parallel VLSI Architectures*, Vol. A, pages 341–362. North Holland, Elsevier, Amsterdam, 1991.

[6] F. Catthoor and H. De Man. Application-specific architectural methodologies for high-throughput digital signal and image processing. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 37, number 2, pages 176–192, Feb 1990.

[7] B. Cole et al. The embedded processor breaks out of its niche. *Electronics*, pages 61–87. McGraw-Hill, Mar 1988.

[8] J.-M. Delosme and I. Ipsen. Efficient systolic arrays for the solution of Toeplitz systems. In W. Moore, A. McCabe, and R. Urquhart, editors, *Systolic arrays*, pages 37–46. Adam Hilger, Bristol, 1987.

[9] H. De Man, F. Catthoor, G. Goossens, J. van Meerbergen, J. Rabaey, and J. Huisken. Architecture-driven synthesis techniques for mapping digital signal processing algorithms into silicon. *Proc. of the IEEE*, 78, number 2, pages 58–78, Feb 1990.

[10] P. Dewilde and E. Deprettere. Architectural synthesis of large, nearly regular algorithms: design trajectory and environment. *Annales des télécommunications*, 46, number 1-2, pages 48–59, Jan–Feb 1991.

[11] J. Fortes and D. Moldovan. Parallelism detection and transformation techniques useful for VLSI algorithms. *Journal of Parallel and Distributed Computing*, 2, pages 277–301, 1985.

[12] A. Jerraya, I. Park, and K. O'Brien. AMICAL: an interactive high-level synthesis environment. *Proceedings of the European Design Automation Conf.*, Paris, France, Feb 1993.

[13] D. Gajski, editor. *Silicon Compilation.* Addison-Wesley, 1988.

[14] W. Geurts, F. Catthoor, and H. De Man. Time constrained allocation and assignment techniques for high throughput signal processing. *Proc. 29th ACM/IEEE Design Automation Conf.*, Anaheim, pages 124–127, Jun 1992.

[15] P. Hilfinger, J. Rabaey, D. Genin, C. Scheers, and H. De Man. DSP specification using the Silage language. *Proc. Int. Conf. on Acoustics, Speech and Signal Processing*, Albuquerque, NM, pages 1057–1060, Apr 1990.

[16] G. Janssen. Hardware verification using temporal logic: a practical view. In L. Claesen, editor, *Proc. IMEC-IFIP WG 10.2/10.5 International Workshop on Applied Formal Methods for Correct VLSI Design,* Houthalen, Belgium, pages 291–300. Elsevier, Amsterdam, Nov 1989.

[17] G. de Jong. Verification of data flow graphs using temporal logic. In L. Claesen, editor, *Proc. IMEC-IFIP WG 10.2/10.5 International Workshop on Applied Formal Methods for Correct VLSI Design,* Houthalen, Belgium, pages 301–310. Elsevier, Amsterdam, Nov 1989.

[18] R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14, number 3, pages 563–590, July 1967.

[19] S. Kirkpatrick, C. Gelatt Jr., and M. Vecchi. Optimization by Simulated Annealing. *Science*, No. 220, pages 671–680, 1983.

[20] D. Knapp and M. Winslett. A prescriptive formal model for data-path hardware. *IEEE Trans. on Computer-Aided Design*, CAD-11, number 2, pages 158–184, Feb 1992.

[21] T. Krol, J. van Meerbergen, C. Niessen, W. Smits, and J. Huisken. The SPRITE input language: an intermediate format for high-level synthesis. *Proc. European Conf. on Design Automation*, Brussels, Belgium, pages 193–199, March 1992.

[22] D. Ku and G. De Micheli. Synthesis of ASICs with Hercules and Hebe. In R. Camposano and W. Wolf, editors, *Trends in high-level synthesis.* Kluwer, Boston, 1991.

[23] S. Y. Kung. *VLSI Array Processors.* Prentice Hall, 1988.

[24] E. Kyriakis-Bitzaros and C. Goutis. An efficient decomposition technique for mapping nested loops with constant dependencies onto regular processor array processors. *Journal of Parallel and Distributed Computing*, 16, pages 258–264, 1992.

[25] D. Lanneer, S. Note, F. Depuydt, M. Pauwels, F. Catthoor, G. Goossens, and H. De Man. Architectural synthesis for medium and high throughput signal processing with the new CATHEDRAL environment. In R. Camposano and W. Wolf, editors, *Trends in high-level synthesis.* Kluwer, Boston, 1991.

[26] M. McFarland, A. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proc. of the IEEE*, 78, number 2, pages 301–318, Feb 1990.

[27] J. Madsen and J. Brage. Flow graph modeling using VHDL bus resolution functions. *Proc. of the First European Conference on VHDL Methods,* IMT, Marseille, Sep 1990.

[28] J. Madsen. *Layout synthesis using compiled cells.* PhD Thesis, Department of Computer Science, Technical University of Denmark, 1992.

[29] D. Moldovan. Advis: a software package for the design of systolic arrays. *Proc. IEEE Int. Conf. on Computer Design*, Port Chester NY, pages 158–164, Oct 1984.

[30] P. Michel, U. Lauther, and P. Duzy, editors. *The synthesis approach to digital system design.* Kluwer, Boston, 1992.

[31] K. O'Brien, M. Rahmouni, and A. Jerraya. DLS: a scheduling algorithm for high-level synthesis in VHDL. *Proc. Europ. Design Automation Conf.*, Paris, France, Feb 1993.

[32] R. J. Offen, editor. *VLSI Image Processing.* McGraw-Hill, 1985.

[33] V. Paliouras, D. Soudris, and T. Stouraitis. Systematic derivation of the processing element of a systolic array based on residue number system. *Proc. IEEE Int. Symp. on Circuits and Systems*, San Diego, 1992.

[34] W. Philipsen and G. de Jong. Refinement of Petri nets: the neural net approach. *Proc. of the Int. Neural Network Conf.*, Paris, France, Vol. 1, pages 266–269. Kluwer Academic Publishers, Boston, Jul 1990.

[35] P. Quinton. Automatic synthesis of systolic arrays from recurrent uniform equations. *11th Int. Symp. Computer Architecture*, Ann Arbor, pages 208–214, Jun 1984.

[36] P. Quinton and Y. Robert, editors. *Algorithms and parallel VLSI architectures II.* Elsevier, Amsterdam, 1992.

[37] S. Rao and T. Kailath. Architecture design for regular iterative algorithms. In E. E. Swartzlander, editor, *Systolic Signal Processing Systems*, pages 209–297. Dekker Inc, New York, 1987.

[38] J. Rabaey, H. De Man, J. Vanhoof, G. Goossens, and F. Catthoor. CATHEDRAL II: a synthesis system for multi-processor DSP systems. In D. Gajski, editor, *Silicon Compilation*, pages 311–360. Addison-Wesley, 1988.

[39] K. Ranerup and L. Philipson. Control architecture selection from state graph characteristics. *Proc. ASCIS Open Workshop on Controller Synthesis*, Technical University of Denmark, Lyngby, Denmark, Sep 1991.

[40] P. Ruetz, R. Jain and R. Brodersen. Comparison of parallel architectures for image processing systems. *Proc. IEEE Int. Symp. on Circuits and Systems*, San Jose, pages 732–737, April 1986.

[41] C. B. Shung, R. Jain, K. Rimey, E. Wang, M. Srivastava, B. Richards, E. Lettang, K. Azim, L. Thon, P. Hilfinger, J. Rabaey, and R. Brodersen. An integrated CAD system for algorithm-specific IC design. *IEEE Trans. on Computer-Aided Design*, CAD-10, number 4, pages 447–463, Apr 1991.

[42] F. Taylor. Residue arithmetic: a tutorial with examples. *IEEE Computer Magazine*, pages 50–62, May 1984.

[43] V. van Dongen and P. Quinton. Uniformization of linear recurrence equations: a step towards the automatic synthesis of systolic arrays. *IEEE International Conf. on Systolic Arrays*, San Diego, pages 473–482, May 1988.

[44] M. van Swaaij, J. Rosseel, F. Catthoor, and H. De Man. Synthesis of ASIC regular arrays for real-time image processing systems. *Journal of VLSI signal processing*, 3, pages 183–192, Kluwer, Boston, Sep 1991.

[45] M. van Swaaij, F. Franssen, F. Catthoor, and H. De Man. High-level modeling of data and control flow for signal processing systems. In M. Bayoumi, editor, *Design methodologies for VLSI DSP architectures and applications.* Kluwer, Boston, 1992.

[46] N. Wehn, M. Held, and M. Glesner. A novel scheduling/allocation approach for data-path synthesis based on genetic paradigms. *Proc. IFIP Working Conference on Logic and Architecture Synthesis*, Paris, France, May 1990.

[47] N. Wehn, H. Herpel, T. Hollstein, P. Pöchmüller, and M. Glesner. High-level synthesis in a rapid-prototype environment for mechatronic systems. *Proc. of EURO-DAC'92*, Hamburg, pages 188–193, Sep 1992.

[48] P. Windirsch, H. Herpel, A. Laudenbach, and M. Glesner. Application-specific microelectronics for mechatronic systems. *Proc. of EURO-DAC'92*, Hamburg, pages 194–199, Sep 1992.

[49] W. Wolf, A. Takach, and T-C. Lee. Architectural optimization methods for control-dominated machines. In R. Camposano and W. Wolf, editors, *Trends in high-level synthesis.* Kluwer, Boston, 1991.

# 2

# BEHAVIORAL SPECIFICATION FOR SYNTHESIS

**Jos T. J. van Eijndhoven**[1], **Jochen Jess**[1]
**Jens P. Brage**[2]

[1] *Eindhoven University of Technology*
[2] *Technical University of Denmark*

## ABSTRACT

This chapter describes some results of the ASCIS project on behavioral specification languages and models used as input for high-level synthesis. Three very different languages have been investigated for input specification: SILAGE, HARDWAREC, and VHDL. For VHDL, a semantic and syntactic subset suitable for high-level synthesis has been chosen; an important characteristic of this subset is asynchronous communication. The specification languages are converted into a data flow graph representation. A data flow model is presented, which supports hierarchy and special control constructs for conditional and iterative statements and maximizes the opportunities for global optimizations. Standardization at this level enables a synthesis environment which supports different synthesis trajectories starting from a common entry point. Moreover, it has facilitated exchange of examples and algorithms between the project partners.

## 1 INTRODUCTION

High-level synthesis concerns generating an architecture (a network at the register transfer level) that implements (executes) a given behavioral specification. Since the space of possible solutions is extremely large, both hard constraints and optimization criteria are applied. Due to the complexity of the problem, finding the optimal solution cannot in general be guaranteed. This results in different ways of partitioning the synthesis problem and different heuristics to solve subproblems. The partitioning in and ordering of the different subproblems, and the specific algorithms used to solve each of them, very much depend on the application domain. Signal processing, video algorithms, controllers, and microprocessors require different optimization strategies to end up with good architectures.

In the ASCIS project, different groups concentrated on architectural synthesis for different application areas, as described in the subsequent chapters of this book. To allow exchange of examples and algorithms, a common interface was needed at the level of behavioral specifications. While different research groups worked with different specification languages, partly for historical reasons but more importantly because of suitability for their application domain, it was decided to make data exchange at the data flow graph level. The main reasons were:

- Data flow graphs are a suitable starting point for architectural synthesis, since they allow maximal freedom in exploiting area/time tradeoffs and do not impose real restrictions towards different design styles.
- To start architectural synthesis, an initial data flow analysis is required. It is also this process which resolves the very different nature of current designer interface languages (VHDL, HARDWAREC, SILAGE, ...). By standardization on the result of this analysis, the input alternatives become available for all the synthesis projects.
- Unlike many designer-oriented specifications, data flow graphs are semantically clean and simple, thus forming an unambiguous behavioral definition suitable to interface to or exchange between synthesis packages, as well as to formal verification.
- Various optimization tools that perform manipulations at the data flow graph level become generally available in the synthesis projects.

As result of this approach, the data flow graphs serve as an intermediate format, and a system structure as outlined in figure 1 is obtained. The formal verification is included to check thoroughly for design errors in the initial behavioral specification, which is of utmost importance with complex ASICs: one cannot afford the time and money for a major redesign. The required type of verification at this stage is sometimes referred to as *model checking*: verifying whether the current specification guarantees certain desired properties [6, 7]. This is opposed to other types of verification, where two different specifications are checked for equivalence or where a developed implementation is checked

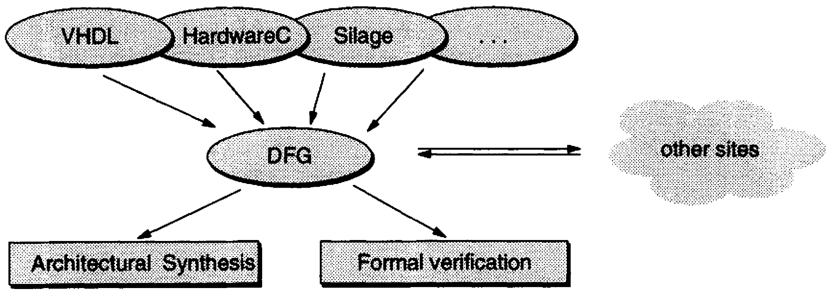


**Figure 1** The language interface.

to fulfill the initial specification, as described in section 4 of chapter 10. To allow for formal verification, the semantics of the data flow graph must be accurately and unambiguously defined. The definition of this intermediate format is therefore done in close cooperation with the development of the verification methods.

The next section will treat the data flow graph standard, as developed in the ASCIS project, with topics like design criteria, allowed graph structures, semantics, syntax, and later extensions. Section 3 discusses designer-oriented specification languages. It will focus on the suitability of VHDL and a newly developed VHDL subset for high-level synthesis.

## 2 THE ASCIS DATA FLOW GRAPH

### 1 Background

The data flow graph model for the ASCIS project [16] is based on earlier work at the Eindhoven University of Technology: both theoretical work [17] and the application in synthesis [14]. The model was designed to combine a unique set of features, which set it apart from other approaches [5, 12, 19]:

- The data flow graphs are allowed to contain conditionals as well as loop constructs. A token flow semantics is responsible for a concise behavioral model, without the need for additional external control information. Having conditionals and loops coherently represented in the data flow graph allows synthesis programs to perform several global optimizations, uninhibited by block boundaries, as are imposed by most other representations.

- A flexible and open approach to data typing is used, allowing numeric as well as bitwise operations on the same values, and supporting easy addition of new (or design-specific) datatypes.
- Dedicated nodes can be used for input and output operations, allowing the specification of sequences of reads and writes on one physical port, conditional I/O, or the sharing of one physical port with several hierarchically structured subgraphs.
- The exchange format is a textual file with a Lisp-like syntax. This makes the format easily extendible for local or future needs, while maintaining backwards compatibility with older programs that do not understand these extensions.

## 2 The data flow graph

A data flow graph is a graph where each *node* represents an operation, and the *edges* represent the transfer of values between the nodes. The edges attach at *ports* of the nodes. The ports are either input ports or output ports. The behavior of a node is defined as a behavior between its ports. A crucial property of the data flow graph is that each input port has precisely one edge attached to it, whereas the number of edges on an output port is left free.

The behavior of the graph is defined by a token flow mechanism. A token flow machine is a graph where the nodes represent operations, and the edges transport tokens from the origin node to a destination node (directed edges). A token can correspond to a new data-value—such a token is called a *data* token—or it is just a signal which can enable the destination node (a *sequence* token). A token stays on an edge until it is consumed by the node at its destination. In principle, it is allowed to have multiple tokens on an edge, in which case they maintain their order: a queue of tokens. The execution of a node can start when a token is available on each input port. The node then takes the input tokens away and starts its execution. After execution, the node places one output token (which may contain a computed data value) on each output port and the edges transport these tokens to the next nodes. If multiple edges connect to an output port, each edge obtains a copy of the token.

### *Classification of nodes and edges*

Two types of edges are distinguished by the kind of tokens they transport. *Data* edges transport tokens containing actual data values. *Sequence* edges carry tokens of which the data value is to be ignored: they are used to enforce a certain sequence in the execution of the nodes.

Several different node types are distinguished:

**Operation nodes:** These nodes represent operations like arithmetic operations ($+$, $-$, $\times$, $\div$), boolean operations ($\wedge$, $\vee$, $<$, $\geq$), or more complex operations. The complex operation nodes provide hierarchy within the graph semantics as used in description languages (procedures, functions).

**Input and output nodes:** A graph links with the outside world exclusively through its input and output nodes. Nodes of type *output* are the only nodes without output ports; nodes of type *input* are the only nodes without input ports.

**Constant nodes:** Nodes of type constant are nodes that generate a constant data value at their output port.

**Control nodes:** Such nodes are used for building control structures, such as *if-then-else* or *while-do* constructs.

**Get/put nodes:** These nodes correspond to actions performed on physical terminals of the generated network. On one terminal, a sequence of read and/or write actions can be performed.

**Delay nodes:** Nodes of this type are used to reference data values from previous executions of the graph or to explicitly indicate pipelining. At initialization time of the graph, the node causes an initial token to emerge at its input and otherwise just passes all incoming tokens to its output.

**Array nodes:** An array represents the explicit storage of values, and can be referenced with *update* and *retrieve* nodes.

### *Data flow analysis*

Variables as used in hardware description languages or ordinary programming languages attach names to values, which are inputs and outputs of expressions. In a data flow graph, values obtained from expressions are transported by tokens. Removing the explicit reference to variables in an input language, and creating a data flow graph with a single edge to each input port is called data flow analysis. When no loops are present, this is a straightforward and fast process, well known from compiler technology. See, for example, the pro-

```
Procedure swap(a,b)
begin  h = a;
       a = b;
       b = h;
end
```

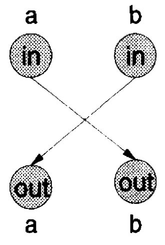


**Figure 2** The *swap* algorithm and its data flow graph.

```
x = a-b;
d++;
Z = e1+e2+e3+e4;
```

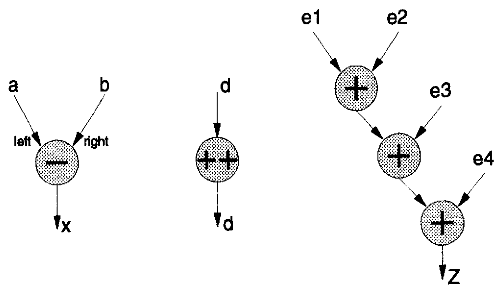


**Figure 3** Simple expressions and their data flow graphs.

cedure *swap* which exchanges its two arguments in figure 2. However, when potentially data-dependent loops including indexed variables are present, the analysis becomes much more complex. This problem is addressed in chapters 4 and 5.

Expressions are built by using operation-type nodes and data edges. An operation node contains one or more input ports and one or more output ports. The translation of a few simple expressions is given in figure 3. Note that the ports must be annotated for the inputs of the "–" operator node. Obviously, tree height reduction can optionally be applied to the resulting data flow graphs, shortening, for instance, the path length through the adders.

## *Operation and procedure nodes*

Procedures are used for a hierarchical description of a design or to break down the description into several smaller parts. A procedural description results in a graph that describes the behavior or semantics of the procedure. The instantiation (call) is done with a node whose type corresponds to the name of

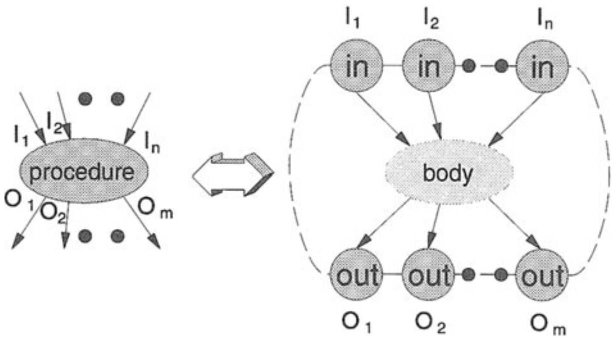


**Figure 4** a) Flow graph with a procedure call; b) Procedure definition.

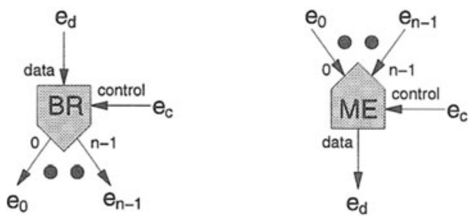


**Figure 5** a) Branch node; b) Merge node.

the graph. The behavior of the instantiation is by definition identical to the in-place expansion of the graph contents.

Each instantiation node belongs to the class of operation type nodes, or equivalently, an operation node is an instantiation of an implicitly predefined graph. The ports of the node correspond to the input and output nodes of the corresponding graph (see figure 4).

## *Conditional statements*

A graph construct for a multiway conditional *case* statement is available, which is also used for representing simple *if-then(-else)* statements. The sub-graph which implements the test expression delivers a *data* token, whose value selects one of several subgraphs to be executed. This is implemented by *branch* and *merge* nodes, which route incoming tokens to one of the subgraphs, and gather the tokens again to a common output for later use (see figure 5).

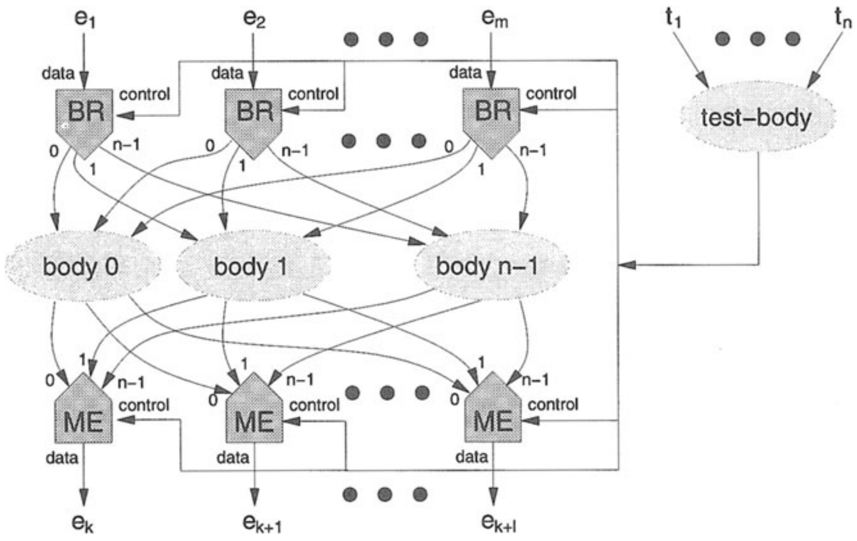


**Figure 6** Template for conditional statements.

A branch node executes when tokens arrive at the *data* input and the *control* input. According the value of the control token, one of the output ports is selected to pass the token from the data input. The output port selected is identified by a table look-up with the control value.

A merge node, on the contrary, executes when a token has arrived on its control input and a token has arrived on the port identified by the value of the control token. After the execution of the *merge* node the token on the selected input is passed to the output.

For the construction of a conditional structure, all bodies are investigated and for each needed input value, a branch node is created. For each computed value that is used later outside the conditional construct, a merge node is created. Then, all control inputs of both the branch and merge nodes are connected to the result of the test expression (see figure 6).

### *Loops*

For the implementation of loop constructs, the *entry* and *exit* control nodes are used, which are similar to *merge* and *branch* nodes. Figure 7 shows the graph structure of a *while-do*-loop. A *do-while* loop (where the body is always exe-

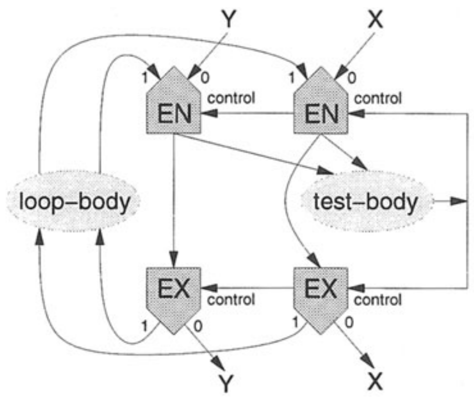


**Figure 7** Example of a while-do loop.

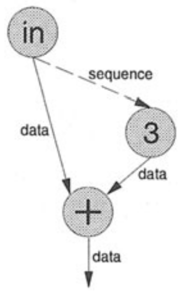


**Figure 8** A constant node.

cuted at least once) is easily made by moving the loop body into the downward edges. To obtain the proper executional semantics, the semantics of the entry nodes define that an initial token, choosing the external entry in the loop, is placed at their control input at graph initialization time.

### *Constants*

For the generation of constant values in the algorithmic description, *constant*-nodes are defined. These nodes deliver the (specified) constant value to their output port when the nodes are executed, and can be regarded as unary operators. A *sequence* edge is connected to deliver the enabling token; see figure 8.

### *Input/Output*

For the data flow graph, a provision for communication with the external (non-DFG) world is made by *get* and *put* nodes. These nodes respectively read from and write to physical terminals. During synthesis, these nodes are mapped on hardware modules, whose implementation can depend upon the semantics of the external world (straightforward pass-through, handshake, bus resolution). In general, several *get* and *put* nodes operate on a single physical terminal. To group these nodes together and fix the sequential ordering of the I/O operations, they are serially linked with *sequence* edges. This path of sequence edges can be continued through conditional constructs, loops, and procedure instantiations. Therefore, the path always starts and ends at an *input* and *out*-

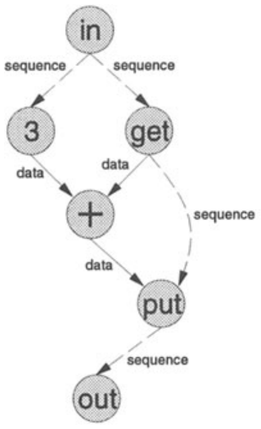


**Figure 9** I/O operations.

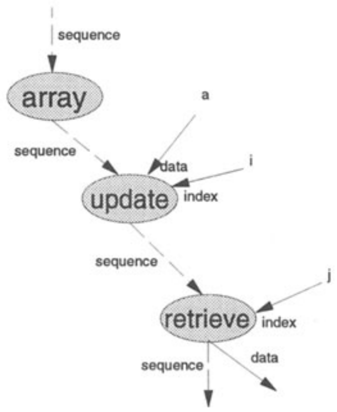


**Figure 10** Array operations.

*put* node, respectively. See figure 9 for a graph corresponding to the following statements:

```
Terminal p;
  x = Get(p);
  y = x + 3;
  Put(p, y);
```

## *Arrays*

An *array* node establishes a (possibly multidimensional) array of values. It is activated by an incoming sequence edge, like a *constant* node. The array values can be read or written by subsequent *retrieve* or *update* nodes, respectively. Initially, these nodes will probably occur in a serial chain of sequence edges, starting at the array node; see figure 10. If it is possible to determine a (partial) independence between the update and retrieve nodes by analyzing the applied index expressions, the chain can be transformed into a rooted directed acyclic graph, allowing more freedom for the synthesis process.

## 3 DFG semantics

As explained in the previous section, the data flow graph has an executional semantics. The semantical definitions were chosen to obtain a set of desirable properties. Assume that a DFG satisfies the following rules:

- All input ports of all nodes have exactly one incoming edge.
- The conditional and loop constructs partition the graph in separate bodies, as outlined in figures 6 and 7.
- The graph becomes acyclic by detaching all edges into *delay* nodes and *entry* nodes (except for leaving the edge at the *entry* port 0 which externally feeds the loop; see figure 7).
- If an operation node is executed, it fetches precisely one token from each input, and generates precisely one token at each output port.

Then, the following properties can be formally proven [7]:

- If a graph is provided with one token at each input node, nodes in the graph can be executed until finally one token appears at each output node. (As a consequence, the graph can be instantiated elsewhere as operation node.)
- If so desired, the execution order can be chosen in such a way that at most one token is at any edge at any time.
- The result of the graph execution (the data values in the final tokens) does not depend upon the chosen order in which the nodes are executed.
- After execution of the graph, no tokens remain in the graph, except replacements for the tokens that were inserted at initialization time (i.e., the tokens at the control inputs of the entry nodes, and at the inputs of the delay nodes).
- If a sequence (queue) of tokens is provided for each input node, again nodes can be executed in any chosen order, resulting in a unique queue for each output node. (The different sets of input tokens can never intermix or influence each other.)

This last property is useful for studying pipelined or multithreaded architectures. Together with the more flexible I/O, it compares favorably with the otherwise resembling approach of SIL [11].

The defined executional semantics is based on presence and passing of tokens (discrete data values), and thus fixes an algorithmic behavior. On purpose, nothing is said about *time*. This leaves maximal freedom to optimize timing aspects during synthesis, without disturbing the algorithmic behavior. For example:

- Although the execution of loops seems to be sequential by nature, it is perfectly legal to have all iterations of the body executed in a single clock cycle. Also with real sequential executions, for instance one variable of the loop can cycle with twice the iteration speed of another variable in the same loop.
- The *time* needed to execute an operation is not a property of the data flow node, but instead a property of the hardware module that is assigned to execute that node. As a result, different nodes of the same operation type can be assigned to different hardware modules, which behave differently over time.
- If an operation with three inputs and two outputs executes, it consumes three input tokens and produces two output tokens. Seen in *time* it might first consume two tokens, then produce one output token, then eat the third token, and finally produce the last output token.

Besides as a property of the hardware modules, time can come in as designer-specified constraints. These are added into the graph as *sequence* edges, labeled with the time constraints. A detailed coverage falls outside the scope of this chapter.

## 4 DFG textual format

To store and exchange the data flow graphs, a text-based format is used [16]. This permits an easy interface to various programming languages and transfer between different machines. The brace-oriented syntax style using a pair of braces for each keyword (like Lisp and EDIF) ensures simple parsing: any LL-1 parser, such as a recursive descent parser, is strong enough. It furthermore permits local and future extensions to the format, without disturbing already existing software (both *upwards* and *downwards* compatibility), and does not require a set of reserved words forbidden as identifiers.

The basic format is very simple: every statement forms a list. Any list starts with an opening brace and a keyword on which the application determines its interest in the list. The items of the list are names, numbers, and other lists, and the list is terminated with a closing brace. If an application is not interested in the information attached to the keyword—or does not recognize the keyword—it can skip this list, without knowing anything about its (structured) contents, by just counting braces. Hence, every tool or site is free to add more data for its own purpose. This property was considered highly important. The following fragment gives an impression of the textual format:

```
(dfg-view
   (graph fdct
      (node N-10 (type +)
         (in-edges E-9 E-8) (out-edges E-34 E-28))
      (edge E-34 (type data) (varname X0)
         (origin N-10) (destination N-23 (port left)))
      (edge E-28 (type data) (varname X0)
         (origin N-10) (destination N-20))
      (node N-11 (type +)
         (in-edges E-11 E-10) (out-edges E-32 E-30))
      ...
```

## 5 Recent developments

During the last year of ASCIS, the DFG format was extended to support intermediate or full synthesis results, and a standard way to include and describe libraries was introduced. Such an extension greatly enhances possible cooperation between the partners, by allowing the comparison or use of each other's algorithms for individual synthesis steps (scheduling, allocation, binding, and network generation). The extension is basically made by adding two new types of graphs, a control graph (CTG) and a network graph (NWG), next to the existing data flow graph. Whereas the DFG defines the algorithmic behavior, the CTG fixes the timing behavior and hints on the controller design, and the NWG defines the hardware on which the algorithm executes: the final synthesis result. All these extensions have not yet been incorporated into the systems of the ASCIS partners.

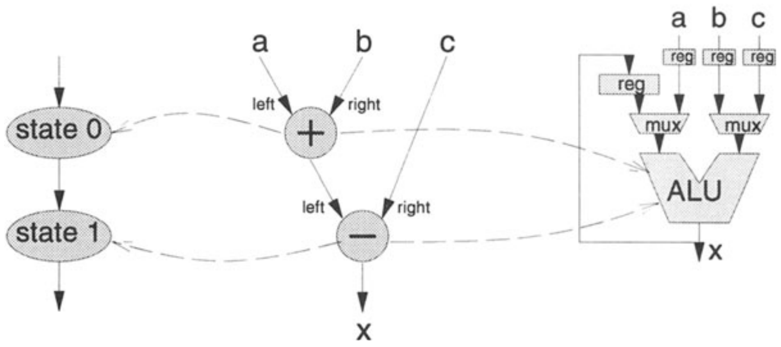


**Figure 11** The control, data flow, and network graphs.

The control graph basically corresponds to the finite state diagram, as commonly drawn for controllers. However, we extended its semantics to allow concurrent multiple active states, and hierarchical structuring of such graphs. This allows multithreaded operation, as used, for instance, in chapter 9. The result of scheduling can now be expressed as links between DFG and CTG nodes.

The network graph describes the final network that results from the architectural synthesis. The nodes in the graph correspond to physical modules to be used in the final architecture. Initially, the graph might contain a set of nodes only (no edges), indicating the set of hardware modules on which the algorithm must be executed: the result of module allocation. Later, links between DFG nodes and NWG nodes indicate which operations are mapped onto which modules: the result of binding. Similar notions hold for register files and busses. Finally, the fully interconnected network follows as result. These node links are shown in a tiny example in figure 11.

Besides links between nodes, links between graphs are supported. These express relations such as "this DFG is controlled by this CTG," or "this NWG is a possible implementation of this DFG." These graph links allow, together with the hierarchy concepts, the description of the synthesis library (operations, modules, and their relations) in the same terms, by adding a DFG without body for each operation and a NWG without body for each module.

A programming interface has been developed to manipulate these sets of graphs, suitable to be used in all tools, with the following features:

- The interface provides functions to access and manipulate graphs in the three domains (data flow, control, network), and the links between them. The consistency of the data structures is enforced by the interface.
- No assumptions are made on the actual high-level synthesis method used or on the order of solving different subproblems. The basic functionality is sufficient to represent any partial synthesis result.
- Each application can extend the provided data structure for its own needs by means of object inheritance. This does not affect the functionality of the library. In particular, the writer and parser accept the extension without the need for recompiling the interface library.
- The interface has a parser and a writer to exchange data between tools as textual files. Data added by one application remain hidden for other applications that are not prepared to use them.

## 3 INPUT SPECIFICATION LANGUAGES

Ideally, the same language should be used at all points in the synthesis process. Unfortunately, different purposes in language usage tends to result in language requirements which cannot be reconciled. In particular, for high-level synthesis the following two purposes are important:

- For human specification of the input to high-level synthesis, the important language features are conciseness and portability. Standard languages, such as VHDL, are well suited for this purpose.
- For internal use in the synthesis tools, simple and well-defined languages are highly desirable. The token flow model described above is particularly well suited for this purpose.

Because of this dichotomy in the language requirements, several languages *must* be used. This imposes the condition that either all the languages support the same interface semantics, or the language definitions provide an external conversion methodology between the interface protocols. For both the behavior defined on the interface and the internal behavior of a block, specification of the desired behavior must consider both functionality, sequencing, and timing.

Three hardware description languages were of particular interest as input languages in the ASCIS project: SILAGE, HARDWAREC, and VHDL.

## 1 Silage

SILAGE has relatively old roots—it was conceived around 1983—and was specifically designed to drive synthesis systems [10]. The language specifies behavior in a functional style, where each variable is assigned only once, and is geared towards real-time digital signal processing applications, for which it permits a very compact specification. The language implicitly assumes an outer loop which infinitely repeats over time, presenting a new set of input data values for each execution. A compact and elegant "delay operator" is used to reference values from previous execution phases. As basic data types, the language supports 2's-complement integers, fixed-point numbers of any specified precision, and bitvectors. Powerful constructs are available to handle arrays, and the language has conditional statements, loops, and functions. Loops are expected to be manifest, i.e., the loop bounds can be evaluated at compile time and do not depend on run-time data values.

## 2 HardwareC

HARDWAREC is designed to drive architectural synthesis over a large spectrum of application areas [8]. It features both a procedural part to describe algorithmic behavior and a declarative part to describe a network of interconnected components. For communication between concurrently active processes, it furthermore explicitly supports message passing channels. In the algorithmic part, the language basically supports only bitvectors for data; these may be interpreted as 2's-complement integers. Loops and bitvector indices must be resolved at compile time, and the language lacks an array construct. Support is provided for explicit specification of timing constraints.

To overcome some of the limitations, the ASCIS group at Darmstadt has developed an enhanced HARDWAREC with more advanced data types (e.g., arrays are allowed). However, the support for network structures was dropped. See chapter 8 for a description of the Darmstadt synthesis system.

## 3 VHDL

VHDL was approved as an IEEE standard in 1987 and has gained considerable momentum in the last few years [18, 1]. The language model can be described as a network of interconnected components, each of which has an algorithmically described behavior. The expressive power of the language is very large: all basic data types, including subranges, records, and arrays, are supported;

overloading permits operators and functions to be redefined for different data types; and powerful configuration control statements are provided. Even constructs difficult to realize in hardware, such as file access, unconstrained arrays, and dynamic memory allocation, are provided.

The expressiveness makes the language attractive for many applications, and allows, for instance, its use for both the synthesis input (the algorithm) and the output (the synthesized architecture). Using commercial simulators, it is then possible to simulate both the specification and the implementation within the same environment.

At the time VHDL was designed, the main objective of the language was to describe and simulate the input/output behavior of an existing hardware module. As a consequence, the semantics of VHDL was based on the concept of the event-driven hardware simulator:

> A process is activated when an event (typically a signal edge) occurs on an input signal. The process then executes its algorithm in zero time, possibly changing some output signals. The effect of the output signal changes may be delayed by a specified time.

Unfortunately, this choice of semantics makes VHDL ill-suited as a language for high-level synthesis: a rigorous implementation of a VHDL design must conform to the behavior of the VHDL source code as defined by the VHDL Language Reference Manual, down to each delta-time unit. This would effectively require the implementation of a VHDL simulator kernel in the hardware, which is obviously neither feasible in practice nor the purpose of hardware synthesis. An additional problem is the large size of VHDL, which makes the design of synthesis tools unnecessarily difficult.

Consequently, VHDL is not a viable choice for a high-level synthesis input language. Rather than choosing an entirely different language, however, there is an alternative: to select a subset of VHDL. Due to VHDL's status as a standard and its broad acceptance, this option received special attention in the ASCIS project.

## 4 To subset or not to subset

There are several major advantages in creating an embedded language with VHDL as the base language:

- It becomes possible to perform mixed-level simulations on partly synthesized descriptions.
- Choosing an industry standard language makes it easier to overcome the university/industry barrier.
- It is easier to steal a language than to design one, provided that one can avoid conflicts in the semantics.

The main consequence of selecting a subset of VHDL's syntactics is the possibility of applying a different interpretation to the subset, i.e., it is possible to choose semantics that are usable for synthesis. The main disadvantage of subsetting VHDL is, of course, a loss of portability since different tools may utilize different subsets.

When a new language is created, it is obviously necessary to ensure that the semantics of the language are well defined. Less obvious, perhaps, is that this is true also for embedded languages: it is extremely important to define exactly which parts of the original language are part of the new language, and what extensions are introduced. Failure to do so results in ambiguities, which lead to interpretation errors and descriptions that cannot be carried between tools.

A particularly important requirement is to preserve the input/output behavior of the original language (for the chosen subset), with respect to some abstraction of the interface to the external world. L. Berrojo et al. [2] suggest a subset somewhat similar to that presented in the following subsection, but the interpretation of their subset violates this cardinal rule.

## 5 ProcVHDL: semantics for synthesis

PROCVHDL [4] is a subset of VHDL intended to be used as an input language for high-level synthesis. As mentioned in the previous subsection, it is possible to choose a new interpretation for a syntactical subset. In the case of PROCVHDL, the new semantics are based on the following hardware model:

> The design and its environment are modeled as procedural functional units in a hierarchical network, communicating by level-sensitive, asynchronous protocols.

In order to ensure preservation of the input/output behavior between the VHDL and the PROCVHDL interpretation, the following abstraction is imposed on the interface:

> Only the sequencing of input/output events is considered as important (i.e., the exact timing is ignored).

This abstraction is acceptable for the kind of systems PROCVHDL was designed to specify: systems employing only asynchronous protocols.[1] Despite the fact that the semantics of VHDL and PROCVHDL differ considerably, it turns out that PROCVHDL actually matches quite well with a subset of VHDL:

- The hierarchical network of PROCVHDL matches the component instantiation concept for structural descriptions in VHDL, with communication carried out by simple VHDL signals. This match would in fact be possible for most simulator-based languages, and is mainly a consequence of the choice of asynchronous communication protocols in the hardware model: this model places very few restrictions on the actual low-level signal behavior.
- The procedural model for functional units used by PROCVHDL are well matched by the behavioral descriptions in VHDL, whereas the functional descriptions of many older hardware definition languages have insufficient expressive power.

It should be noted that even as the asynchronous communication of PROCVHDL may be embedded in VHDL's event-based semantics, it may also be built on top of synchronous hardware. In fact, a likely implementation of a PROCVHDL functional unit is a synchronous finite state machine with an attached datapath.

[1]Several other subsets have been suggested for synchronous systems [13, 15].

## 6 The ProcVHDL language definition

This section describes the ProcVHDL subset. The description is far from complete, but a more detailed description, including a full syntax specification, is given elsewhere [3]. The two main subsetting restrictions in ProcVHDL is on the **WAIT** and signal assignment statements. These restrictions are described below.

The ProcVHDL **WAIT** statement is strongly restricted compared to the VHDL counterpart. Only two forms are permitted:

1. **WAIT UNTIL** BooleanSignal [**OR** BooleanSignal...];
2. **WAIT FOR** Time;

The first construct is used to synchronize with external events. The execution is suspended until one or more external signals becomes active. The second form is used to sequence output signals. A **WAIT** for any length of time indicates that a given set of output signals must change before any other signal changes. Note that the actual time specified is without importance, though it may be useful to pace simulation of the circuit.

Notice that **WAIT ON** (sensitivity lists) are eliminated. Also, **WAIT UNTIL** with more than a single signal is further restricted: this statement *must* be bracketed by an **IF** statement with the negated condition. This is caused by the fact that VHDL (and ProcVHDL) requires a level change to break a **WAIT**. This is not, in general, compatible with level-sensitive interface protocols.

The other restriction on the process statements of VHDL is on the signal assignment statement. ProcVHDL does not allow the VHDL **AFTER** clause that is used for timing specifications—the focus of ProcVHDL is sequencing, not timing. Also, the following restriction is imposed on signal assignments in ProcVHDL: it is illegal to assign twice to the same signal without an intervening **WAIT** statement. (The first assignment could, of course, be ignored, as it would be in VHDL, but permitting the construct would make analysis difficult.)

```
WHILE Count < Max AND (Z.R * Z.R + Z.I * Z.I)/Unit < 4*Unit LOOP
  Temp := (Z.R * Z.R - Z.I * Z.I)/Unit + C.R;
  Z.I := 2 * Z.R * Z.I/Unit + C.I;
  Z.R := Temp;
  Count := Count +1;
END LOOP;
Iterations <= Count -1;
WAIT for 1 ns;
Strobe <= TRUE;
IF NOT Acknowledge THEN WAIT UNTIL Acknowledge; END IF;
```

**Figure 12** A ProcVhdl fragment.

## 7 A ProcVHDL example

Figure 12 shows an excerpt from a ProcVhdl specification. While this is a small toy example, a baseline JPEG encoder/decoder has been specified in about 2000 lines of code [9], demonstrating the feasibility of ProcVhdl for reasonably complex specifications. The example displays a typical property of ProcVhdl code: the input/output specification is intertwined with the general control flow, similarly to the way data and control flow are intertwined in the DFGs described in section 2. This is different from the typical synchronous modeling style in Vhdl, which tends to separate the control flow from the input/output operations, thus cluttering up the model (from a human point of view) and increasing the risk of specification errors.

## 4 CONCLUSION

In the Ascis project, a data flow graph standard has been developed, targeted towards high-level synthesis. Its coherent representation of both data and control flow and its independence of timing aspects provide extreme flexibility for transformations and optimizations during synthesis. Furthermore, the specification is accurate enough to allow formal reasoning about its behavior. These data flow graphs are appropriate over a large range of application domains, can be generated from different designer specification languages, and are therefore suitable as interchange medium. The concept of a DFG will recur frequently in the subsequent chapters.

VHDL is not immediately suitable as a designer specification language, due to its event-driven semantics and inherent overspecification of timing. A subset which adheres to the original semantics of VHDL has been developed to overcome this problem. This is achieved by restricting the interprocess communication to asynchronous protocols, thus defining only the sequencing of input/output events.

## REFERENCES

[1] J.-M. Bergé, A. Fonkoua, S. Maginot, and J. Rouillard. *VHDL Designer's Reference.* Kluwer Academic Publishers, Dordrecht, Netherlands, 1992.

[2] L. Berrojo, P. Sanchez, and E. Villar. High-level synthesis and simulation with VHDL. *Proc. of Second European Conference on VHDL Methods,* Stockholm, Sweden, pages 62–69, Sep 1991.

[3] J. P. Brage. *ProcVHDL: A VHDL subset for high-level synthesis.* Technical report, Dep. of Comp. Sc., Technical University of Denmark, Lyngby, Denmark, Jun 1991.

[4] J. P. Brage. Hardware description languages for synthesis: problems and possibilities. *Proc. of Tenth NORCHIP Seminar,* Helsinki, Finland, pages 22–29, Nov 1992.

[5] R. Camposano and W. Rosenstiel. Synthesizing circuits from behavioural specifications. *IEEE Trans. on Comp. Aided Design*, CAD-8, number 2, pages 171–180, Feb 1989.

[6] G. G. de Jong. Verification of data flow graphs using temporal logic. In L. J. M. Claessen, editor, *Formal VLSI Correctness Verification, VLSI Design Methods-II: proc. of the IMEC-IFIP WG10.2 WG10.5 Int. Workshop on Appl. Formal Methods for Correct VLSI Design*, pages 169–178, North-Holland, 1990.

[7] G. G. de Jong. *Generalized data flow graphs: theory and applications.* To appear as PhD thesis. Eindhoven Univ. of Tech., Eindhoven, The Netherlands, 1993.

[8] G. De Micheli and D. C. Ku. HERCULES—a system for high-level synthesis. *Proc. of the 25th Design Autom. Conf.*, Anaheim, CA, Jun 1988.

[9] K. Djigande. *Image compression and decompression, system architecture.* Master's thesis, Dep. of Comp. Sc., Technical University of Denmark, Lyngby, Denmark, Jul 1992.

[10] P. Hilfinger, J. Rabaey, D. Genin, C. Scheers, and H. De Man. DSP specification using the SILAGE language. *IEEE Int. conf. on Acoustics, Speech and Signal Processing*, pages 1057–1060, Apr 1990.

[11] Th. Krol, J. van Meerbergen, C. Niessen, W. Smits, and J. Huisken. The Sprite Input Language, an intermediate format for high level synthesis. *Proc. of Eur. Conf. on Design Automation (EDAC)*, Brussels, Belgium, pages 186–192, Mar 1992.

[12] J. S. Lis and D. D. Gajski. Synthesis from VHDL. *Proc. of the Int. Conf. on Comp. Design*, pages 378–381, 1988.

[13] A. Postula. VHDL specific issues in high level synthesis. *Proc. of Second European Conference on VHDL Methods,* Stockholm, Sweden, pages 70–77, Sep 1991.

[14] L. Stok. *Architectural Synthesis and Optimization of Digital Systems.* PhD Thesis, Eindhoven Univ. of Tech., Eindhoven, The Netherlands, 1991.

[15] A. Stoll, J. Biesenack, and S. Rumler. Flexible timing specification in a VHDL synthesis system. *Proc. of EURO-DAC '92,* Hamburg, Germany, pages 610–615, Sep 1992.

[16] J. T. J. van Eijndhoven, G. G. de Jong, and L. Stok. *The ASCIS data flow graph: semantics and textual format.* Technical report 91-E-251, Eindhoven University of Technology, Jun 1991.

[17] A. H. Veen. *The misconstrued semicolon: reconciling imperative languages and dataflow machines.* PhD Thesis, Eindhoven University of Technology, The Netherlands, 1985.

[18] *IEEE Standard VHDL Language Reference Manual.* IEEE Std. 1076–1987, The Institute of Electrical and Electronics Engineers, Inc., New York, USA, 1988.

[19] R. A. Walker and D. E. Thomas. Design representation and transformation in the system architect's workbench. *Proc. of the Int. Conf. on Comp. Aided Design*, pages 166–169, 1987.

# 3

# FORMAL METHODS FOR SOLVING THE ALGEBRAIC PATH PROBLEM

**Alain Darte, Tanguy Risset, Yves Robert**

*École Normale Supérieure de Lyon*

## ABSTRACT

This chapter deals with the interplay between algorithm design and synthesis methodologies. The algebraic path problem is used throughout the text as a target computational kernel. First, we present pioneering and state-of-art systolic implementations; then, we describe how synthesis methodologies have been extended (space-time optimality, partitioning techniques) to cope with advances in the algorithmic field.

## 1 INTRODUCTION

This chapter focuses on the architecture study for a basic algorithm kernel—the algebraic path problem (APP)—that we have been investigating during NANA. Our goal in this chapter is threefold:

- Introduce the need for novel design methodologies and synthesis techniques for regular array processor architectures.
- Survey the progress that has been made in the field.
- Illustrate the cross-fertilization between parallel VLSI algorithm and architecture design and synthesis methodologies.

Section 2 is devoted to a brief presentation of the APP and its applications in computer science and electrical engineering. Section 3 surveys the pioneering developments that led to the first systolic solutions of the APP. Next, section 4 presents the very efficient—and accordingly very complex—solutions that have been derived during NANA. In section 5 we first explain the problems that prevent a direct automatic synthesis of these solutions (non-uniform dependences, anti-dependences, ...), and the developments that were proposed to cope with these problems (localization, re-indexing, ...). Then, we address the space-time complexity issues that permit an evaluation of the resulting architectures, and give partial solutions to the support of the architectural mapping process, both from an algorithmic and a methodological point of view. Finally, we deal with array partitioning issues in section 6.

Rather than dealing with technical details, we present problems and give an insight to their solution. Throughout the text, we refer to the corresponding NANA results and publications.

## 2 THE ALGEBRAIC PATH PROBLEM

### 1 The APP formulation

Path problems are ubiquitous in computer science. The algebraic path problem (APP) is a general framework that unifies several algorithms arising from various fields of computer science. It is defined as follows [35]: given a weighted graph $G = (V, E, w)$, where $V$ is a finite vertex set, $E$ an arc set, and $w$ a function $w : E \to H$ with weights from a semi-ring $(H, \oplus, \otimes)$ with zero 0 and unity 1; find for all pairs of vertices $(i, j)$ the quantities

$$d_{i,j} = \bigoplus_{p \in M_{ij}} w(p)$$

where $M_{ij}$ denotes the set of all paths from $i$ to $j$. To the weighted graph $(V, E, w)$, we associate the $n \times n$ weight matrix $A = (a_{ij})$, where $a_{ij} = w(i, j)$ if $(i, j) \in E$ and $a_{ij} = 0$ otherwise. We denote by $M_{ij}^k$ the set of all paths from $i$ to $j$ which contain only vertices $x$ with $1 \leq x \leq k$ as intermediate vertices. In practice, $a_{i,j}^k = \bigoplus_{p \in M_{ij}^k} w(p)$ is equal to the successive values of $a_{ij}$ which we want to compute, starting from the initial value $a_{ij}(0) = a_{ij}$ up to $a_{ij}(n) = d_{ij}$. The solution to the APP problem can be obtained by a direct generalization of the Gauss-Jordan diagonalization algorithm to compute the inverse of a real matrix:

```
for $k := 1$ to $n$ do
    begin
    $a_{kk}^{k} := (a_{kk}^{k-1})^{\star}$;
    for $i := 1$ to $n$, $i \neq k$ do
        $a_{ik}^{k} := a_{ik}^{k-1} \otimes a_{kk}^{k}$ ;
    for $j := 1$ to $n$, $j \neq k$ do
        begin
        for $i := 1$ to $n$, $i \neq k$ do
            begin
            $a_{ij}^{k} := a_{ij}^{k-1} \oplus a_{ik}^{k} \otimes a_{kj}^{k-1}$ ;
            $a_{kj}^{k} := a_{kk}^{k} \otimes a_{kj}^{k-1}$ ;
            end
        end
    end
```

In the computational procedure we have let $c^{\star} = \bigoplus_{i \geq 0} c^{i}$ for $c \in H$.

## 2 Applications of the APP

Applications of the APP are obtained by specializing the operations $\oplus$ and $\otimes$ in the appropriate semi-rings. Let us mention three of them:

- Determination of the inverse of a real matrix:

  $A$ is a real matrix, $\oplus$ and $\otimes$ are the usual operations in $H$, and $\otimes$ is defined by: **if** $c \neq 1$ **then** $c^{\star} := 1/(1-c)$. The APP algorithm outputs the inverse matrix $(I - A)^{-1}$. Of course, straightforward modifications permit to compute $A^{-1}$ directly.

- Shortest distances in a weighted graph:

  The weights $a_{ij}$ are taken in $\mathbf{H} = H \cup (-\infty, +\infty)$, $\oplus$ is the addition in H extended to $\mathbf{H}$ (with $-\infty \oplus \infty = +\infty$), $\otimes$ is the minimum, and $^{\star}$ is defined by: **if** $c \geq 0$ **then** $c^{\star} := 0$ **else** $c^{\star} := -\infty$.

- Transitive and reflexive closure of a binary relation:

  The $a_{ij}$ are boolean, $\oplus$ and $\otimes$ are respectively the *and* and *or* operations, and $^{\star}$ is defined by: $c^{\star} := 1$ for all $c$.

This short list demonstrates the great importance of the APP in computer science, particularly in statistical data analysis and control engineering. Many practical applications, such as on-line data analysis for robust robot or vehicle control (to mention a single one), rely on shortest-path computations as a key computational kernel.

## 3 PIONEERING SYSTOLIC APP DESIGNS

### 1 Initial designs

From 1980 to 1985, several authors have presented systolic arrays for solving special instances of the APP, such as transitive closure or matrix inversion. The first systolic array introduced in the literature for solving a specialized instance of the APP is the two-dimensional (2-D) toroidal array of Guibas-Kung-Thompson [17]. It is restricted to computing the reflexive and transitive closure of a binary relation. This array has been extended to solving other graph algorithms (shortest paths, connected components) by Ullman [36]. The first solution to the general APP was presented by Rote [35]: this solution was based on a clever extension of the Kung-Leiserson array for band LU decomposition [21]; it is a hexagonally connected systolic array of $(n+1)^2$ processors that can solve any instance of size $n$ of the APP within $7n-2$ time steps. One time step corresponds here to a multiply-and-add or a star operation in the underlying algebra.

In table 1, we summarize the characteristics of several published solutions for the APP. The execution time includes loading and unloading of the matrix coefficients. The area is expressed as the number of elementary cells. The period is defined as the minimum time $P$ between the solution of two consecutive instances of the problem.

### 2 Improving Rote's solution

In table 1, we see that the price to pay for dealing with the general APP seems to be an increase in execution time from $5n$ up to $7n$. In fact, rather than dealing with the above formulation which is close to in-place matrix inversion, several authors had the idea to use an augmented matrix. We can derive an equivalent formulation by padding the matrix $A$ with two copies of the identity matrix: $A := \begin{pmatrix} A & I_n \\ I_n & 0 \end{pmatrix}$. Now operating on the new $2n \times 2n$ matrix $A$, we have the following algorithm, which is closer to the Faddeev algorithm [18]:

| Reference | Application | Area | Time | Period |
|---|---|---|---|---|
| Guibas & al. [17] | Transitive closure | $n^2$ | $6n$ | $4n$ |
| Lakhani-Dorairaj [16] | Shortest paths | $n^2$ | $5n$ | $n$ |
| Kung-Lo [24] | Shortest paths | $n^2$ | $7n$ | $3n$ |
| Kramer-Leeuwen [25] | Matrix inversion | $n^2$ | $6n$ | $5n$ |
| Nash-Hansen [18] | Matrix inversion | $1.5n^2$ | $5n$ | $n$ |
| Robert-Tchuente [33] | Matrix inversion | $n^2$ | $5n$ | $n$ |
| Rote [35] | General APP | $n^2$ | $7n$ | $5n$ |

**Table 1** Pioneering arrays solving the APP.

```
for $k := 1$ to $n$ do
    begin
    $a_{kk}^{k} := (a_{kk}^{k-1})^{\star}$;
    for $i := k + 1$ to $k + n$ do
        $a_{ik}^{k} := a_{ik}^{k-1} \otimes a_{kk}^{k}$ ;
    for $j := k + 1$ to $k + n$ do
        begin
        for $i := k + 1$ to $k + n$ do
            begin
            $a_{ij}^{k} := a_{ij}^{k-1} \oplus a_{ik}^{k} \otimes a_{kj}^{k-1}$ ;
            $a_{kj}^{k} := a_{kk}^{k} \otimes a_{kj}^{k-1}$ ;
            end
        end
    end
```

The solutions of Robert-Trystram [34] and Kung-Lo-Lewis [23] (later extended by Lewis-Kung [22] to cope with the general APP) are very similar. Both can be viewed as extensions of the rectangular Ahmed-Delosme-Morf array [1] for dense matrix triangularization. The array of Delosme [14] is quoted for the sake of completeness: Delosme uses a completely different algorithm (a path algebra extension of the Bareiss algorithm) for solving the APP.

The operation of the Robert-Trystram array is represented in figure 1. Circular cells are devoted to $\star$ operations while rectangular cells perform $\oplus$ and $\otimes$ computations.

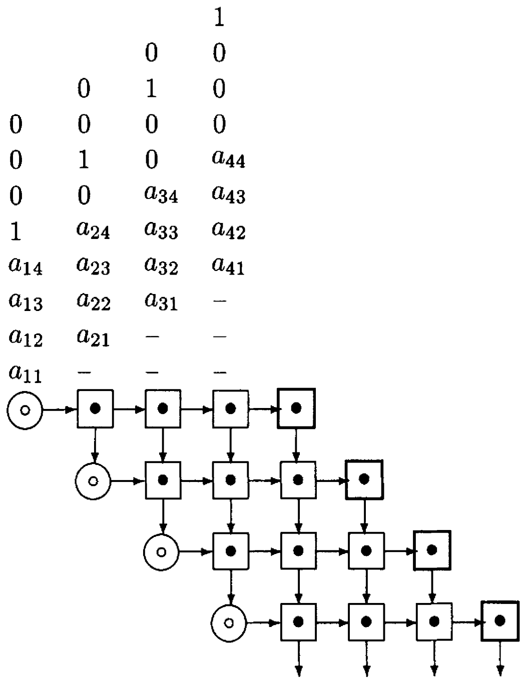

1
0 0
0 1 0
0 0 0 0
0 1 0 $a_{44}$
0 0 $a_{34}$ $a_{43}$
1 $a_{24}$ $a_{33}$ $a_{42}$
$a_{14}$ $a_{23}$ $a_{32}$ $a_{41}$
$a_{13}$ $a_{22}$ $a_{31}$ –
$a_{12}$ $a_{21}$ – –
$a_{11}$ – – –

**Figure 1** A rectangular array for the APP.

## 4 ADVANCED SYSTOLIC APP DESIGNS

### 1 Folding the array

The best area-time tradeoff for solving an instance of the APP of size $n$ seems to be $5n$ units of time and $n^2$ cells; hence the area-time product is $5n^3$. Among the architectures that achieve this tradeoff, the one of Kung-Lo-Lewis [23] has the smallest period $P = n$ (the solution of a new instance of the APP can begin every $n$ steps). Rote [35] has shown that $5n$ units of time is asymptotically optimal under the hypothesis that no coefficient $a_{ij}$ is duplicated in the array. Note that the architecture of Delosme [14] requires $4n$ time-steps and $5n^2/4$ cells; hence the area-time product is $5n^3$ for this architecture, too.

Benaini, Robert, and Tourancheau [4] showed how to decrease the area-time product by a factor of two: they introduce a 2-D toroidal array of only $n^2/2$ processors that can solve any instance of the APP in $5n$ time-steps. The main idea is the use of the torus topology that allows the data to re-enter the array after crossing it a first time, just as in the pioneering array of Guibas-Kung-Thompson [17].

Nothing comes for free, though! The price to pay for the folded array is a period of $P = 4n$ against a period $P = n$ for Kung-Lo-Lewis [23]. Also, partitioning issues are more involved, since acyclic implementations usually exhibit more favorable characteristics with respect to fault tolerance, two-level pipelining, and problem decomposition in general (Hwang and Cheng [9], Kung and Lam [20]). However, the number of cells in the folded array is half that of Kung-Lo-Lewis, and this is a considerable improvement.

## 2 Obtaining space-time minimality

In latency-limited applications, the important criterion is the execution time $T$. This is in contrast to the real-time signal and data processing applications, where the period $P$ is more important. Solutions for the latter category are the focus of chapters 5 and 6. In this chapter, the latency-limited case will be addressed. Even though a large part of the underlying theory can be shared, important distinctions will be pointed out.

Following Cappello [6], we have the following two definitions for space-time minimality:

**Definition 3.1** *A schedule is time-minimal when the number of time-steps is equal to the length of a longest path in the dependence graph (which is clearly the minimal time needed to achieve the computation).*

**Definition 3.2** *A systolic array is space-time-minimal when it is scheduled in minimal time and when it uses the minimal number of processors among all possible minimal-time solutions.*

Theoretical considerations show that any systolic array that solves the APP using the previous formulation requires an execution time $T \geq 5n - 2$ time-steps.

| Reference | Application | Area | Time | Period |
|---|---|---|---|---|
| Robert-Trystram [34] | General APP | $n^2$ | $5n$ | $2n$ |
| Kung-Lo-Lewis [23] | Transitive closure | $n^2$ | $5n$ | $n$ |
| Lewis-Kung [22] | General APP | $n^2$ | $5n$ | $n$ |
| Delosme [14] | General APP | $1.25n^2$ | $4n$ | $n$ |

**Table 2** Comparison of some systolic arrays solving the APP.

**Proposition 3.1** *There is one unique time-minimal schedule.*

The proof is given elsewhere [3].

All the arrays listed in table 2 are scheduled with the time-minimal schedule; hence their execution time $5n - 2$ is optimal. A natural question arises: what is the minimal number of processors that a time-minimal systolic solution requires?

When discussing processor allocation, we usually insist that the target design should be regular and locally connected, and therefore, linear allocation functions (see also section 5) have been considered by several authors dealing with automatic synthesis methods. If we find a non-linear allocation function that requires fewer processors, then the resulting solution is not guaranteed to be as regular as the *pure* systolic schemes. However, finding the minimum number of processors for any allocation function will give us a lower bound of what can be achieved, and this will give us an indication on the efficiency of our favorite solutions.

**Proposition 3.2** *The minimum number of processors $A(n)$ for the solution of an instance of the APP of size $n$ in optimal time $T(n) = 5n - 2$ is $A(n) = n^2/3 + O(n)$. More precisely:*

$$\begin{array}{lll} A(n) = \frac{n(n+2)+3}{3} & \text{if} & n \bmod 3 = 0 \quad \text{or} \quad n \bmod 3 = 1 \\ A(n) = \frac{n(n+2)+1}{3} & \text{if} & n \bmod 3 = 2 \end{array}$$

The proof is given elsewhere [3].

| Reference | Application | Area | Time | Period |
|---|---|---|---|---|
| Benaini et al. [34] | General APP | $n^2/2$ | $5n$ | $2n$ |
| Cappello-Scheiman [7] | General APP | $n^2/3$ | $5n$ | $3n$ |
| Benaini-Robert [3] | General APP | $n^2/3$ | $5n$ | $3n$ |
| Delosme [14] | General APP | $1.25n^2$ | $4n$ | $n$ |

**Table 3** Comparison of some systolic arrays solving the APP.

The best array that we have seen so far requires $A = n^2/2 + O(n)$ cells. Hence, there remains a gap of $n^2/6$ cells to suppress with this optimal $A(n)$. This is achieved by the array proposed by Benaini and Robert [3]. We can add the four rows of table 3 to table 2. The array of Cappello-Scheiman [7] is very similar to that of Benaini-Robert [3] and has been obtained independently.

In this subsection, we have established the *systolic complexity* of the APP. We believe that the design of space-time-minimal arrays represents an interesting and important contribution to the knowledge of the systolic model. This contribution is twofold:

- From the theoretic point of view, we can prove optimality results that will contribute to our basic understanding of the limits and potential of systolic computation.
- From the practical point of view, we can compare any existing solution against the one that minimizes both time and space. Therefore, if other design criteria (including technical implementation constraints) are to be taken into account, we have a good basis for performance evaluation.

## 5 EXTENDING SYNTHESIS METHODS

As mentioned in chapter 1, methods for synthesizing systolic arrays from uniform recurrence equations (UREs) or uniform directed acyclic graphs (DAGs) are well understood [12, 19, 26, 27, 29, 28]. The idea is to extract from the original sequential algorithm a dependence graph (DG), where all incoming arcs to a given node come from a fixed-size neighborhood, so that dependencies are local. Space-time transformations are then used for scheduling the DG (timing function) and for mapping nodes onto physical processors (allocation function).

Both linear and piece-wise linear mappings can be derived in a systematic way, and methods exist to optimize given criteria within a constraint on execution time (for latency-limited applications), the period (for throughput-limited processing), or the number of processors (for area-bound applications).

We first briefly review the well-known basic synthesis method. Then we focus upon extensions of this original method, such as more efficient scheduling and bounded broadcast facilities, and also the embedding in a more complete system design trajectory, which also requires, for instance, extraction of the uniform DG and partitioning of the solution on a fixed size array. Where appropriate, we will also refer to the methods described in the other chapters.

We will assume here that the (non-uniform) dependence graph of the APP has already been extracted from the initial textual specification. In practice, this step can be performed automatically from applicative languages in a relatively simple way, as explained briefly in chapter 6; or from procedural nested loop notations expressed in any standard programming language, with more complex procedures, as discussed in chapters 4 and 5.

## 1 Scheduling the DG for the APP

Using the first formulation of the APP, we have the following system of equations. Input equations:

$$1 \le i \le n, 1 \le j \le n \Rightarrow A(i,j,0) = a_{ij}$$

Computation equations:

$$\begin{array}{lcl}
1 \le k \le n & \Rightarrow & A(k,k,k) = (A(i,j,k-1))^{\star} \\
1 \le k \le n, 1 \le i \le n, i \ne k & \Rightarrow & A(i,k,k) = A(i,k,k-1) \otimes A(k,k,k) \\
1 \le k \le n, 1 \le j \le n, j \ne k & \Rightarrow & A(k,j,k) = A(k,k,k) \otimes A(k,j,k-1) \\
1 \le k \le n, 1 \le i \le n, & & \\
1 \le j \le n, i \ne k, j \ne k & \Rightarrow & A(i,j,k) = A(i,j,k-1) \oplus \\
 & & A(i,k,k)(x)A(k,j,k-1)
\end{array}$$

Output equations:

$$1 \le i \le n, 1 \le j \le n \Rightarrow a_{ij} = A(i,j,n)$$

The dependence graph corresponding to the APP algorithm contains opposite dependence vectors: for instance, node (3,2,2) depends on node (2,2,2); hence the dependence vector (1,0,0). But node (1,2,2) also depends upon node (2,2,2); hence the dependence vector (−1,0,0). The existence of these two opposite vectors prevents us from using a linear timing function. The scheduling of this formulation therefore requires complicated manipulations, such as domain translation [29] (an automated synthesis technique, re-indexing, to perform this function is described in chapter 6). However, using the second formulation of the APP, where we have padded the matrix $A$ with two copies of the identity matrix: $A := \begin{pmatrix} A & I_n \\ I_n & 0 \end{pmatrix}$, and now operating on the new $2n \times 2n$ matrix $A$, we have the following system of equations. Input equations:

$$\begin{array}{lll}
1 \le i \le n, 1 \le j \le n, k = 0 & \Rightarrow & A(i,j,k) = a_{ij} \\
1 \le k \le n & \Rightarrow & A(n+k,k,k-1) = 1 \\
1 \le k \le n, k+1 \le j \le n+k & \Rightarrow & A(n+k,j,k-1) = 0 \\
1 \le k \le n & \Rightarrow & A(k,n+k,k-1) = 1 \\
1 \le k \le n, k+1 \le i \le n+k-1 & \Rightarrow & A(i,n+k,k-1) = 0
\end{array}$$

Computation equations:

$$\begin{array}{lll}
1 \le k \le n & \Rightarrow & A(k,k,k) = A(i,j,k-1)^{\star} \\
1 \le k \le n, k+1 \le i \le n+k & \Rightarrow & A(i,k,k) = A(i,k,k-1) \otimes \\
 & & A(k,k,k) \\
1 \le k \le n, k+1 \le j \le n+k & \Rightarrow & A(k,j,k) = A(k,k,k) \otimes \\
 & & A(k,j,k-1) \\
1 \le k \le n, k+1 \le i \le n+k, & \Rightarrow & A(i,j,k) = A(i,j,k-1) \oplus \\
k+1 \le j \le n+k & & A(i,k,k) \otimes A(k,j,k-1)
\end{array}$$

Output equations:

$$1 \le i \le n, 1 \le j \le n \Rightarrow a_{ij} = A(i+n, j+n, n)$$

The computation domain is $D_n = \{(i,j,k), 1 \le k \le n, k \le i,j \le n+k\}$. The DG is represented in figure 2 for $n = 4$. Note that the longest path is $(1,1,1) \rightarrow (2,1,1) \rightarrow (2,2,1) \rightarrow (2,2,2) \rightarrow (3,2,2) \rightarrow \ldots \rightarrow (n,n,n) \rightarrow (n+1,n,n) \rightarrow (n+1,n+1,n)$, for which the length is $3n$.

The time to execute the operations in the DG is determined when we assign both a timing function (schedule) and an allocation function (mapping) to the nodes, subject to the following constraints:

- A node can be computed only when its predecessors (the nodes on which it depends) have been computed at previous steps.
- No processor can compute two different nodes at the same time-step.

The minimal time to schedule the DG is clearly equal to the length of the longest path; hence $t_{opt}(n) = 3n$. Note that achieving this bound would imply

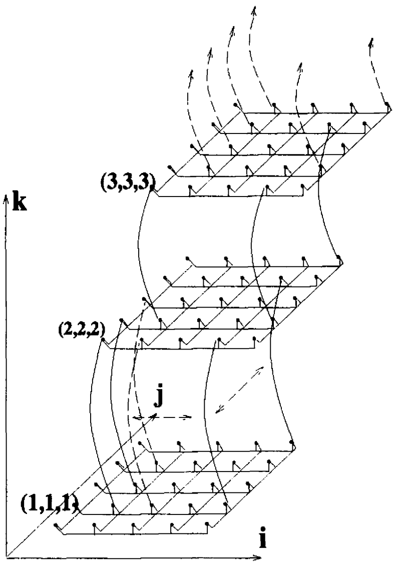


**Figure 2** Non-uniform dependence graph for the APP.

having enough processors and global communications: node $(n, n, n)$ cannot be computed before step $3n - 2$; its value must be broadcast to $n$ processors so as to compute nodes $(i, n, n), n + 1 \leq i \leq 2n$ at time $3n - 1$; and finally the $n^2$ nodes $(i, j, n), n + 1 \leq i, j \leq 2n$ must be computed at time $3n$. Also, note that node $(k, k, k)$ cannot be computed before step $3k - 2$.

There are several algorithmic variations for expressing the DG. The DG of figure 2 (or an equivalent form) is the most commonly used; we take it to be fixed in the following. We point out that we deal with a generic form of the APP. Several modifications can be made when dealing with specific instances. In the Gauss-Jordan algorithm, for instance, we can replace the following equations:

```
$a_{kk}^k := (a_{kk}^{k-1})^\star$ ;
**for** $i := k+1$ to $k+n$ **do**
    $a_{ik}^k := a_{ik}^{k-1} \otimes a_{kk}^k$ ;
```

by the following:

```
**for** $i := k+1$ to $k+n$ **do**
    $a_{ik}^k := a_{ik}^{k-1} / a_{kk}^{k-1}$ ;
```

Then, the minimal time to schedule the DG becomes $2n$. For the transitive closure or shortest path formulations, the closure operation * even disappears!

The DG of figure 2 is not yet uniform, as there are non-local dependences: node $(i, j, k)$ depends on node $(i, k, k)$ and $(k, j, k)$ for all $i, j > k$. An SIMD solution would look as follows:

```
compute node $(k, k, k)$;                                  /* time step $3k - 2$ */
for all $i > k$ do in parallel                             /* time step $3k - 1$ */
    broadcast $a(k, k, k)$ to compute nodes $(i, k, k)$;
for all $i, j > k$ do in parallel                          /* time step $3k$ */
    broadcast $a(i, k, k)$ and $a(k, j, k)$ to compute node $(i, j, k)$;
```

Broadcasting a given data item to an arbitrary number of processors is reasonable in an SIMD environment, but it is not for VLSI implementations, where a limited fan-out is required [10].

## 2 Systolic DG for the APP

The systolic answer to the problem is to localize the broadcasts in the DG before scheduling and mapping its nodes [15], so as to synthesize an architecture where all communications are made local. Such a derivation process is well understood. Some recent synthesis results are discussed in chapter 6. The natural idea is to replace the broadcast of a variable by its pipelined propagation along the direction of the dependence vector. In this way, we obtain the DG of figure 3.

Note that all dependences are local now. The set of dependence vectors is $\Theta = \{\theta_a, \theta_l, \theta_u\}$, with $\theta_a = (0, 0, 1)$, $\theta_l = (0, 1, 0)$, and $\theta_u = (1, 0, 0)$. For the sake of simplicity, we assume that each vertex of $D^n$ has the same computation time, although the operations are not the same in all the vertices. A schedule (or timing function) is a mapping $t : D^n \rightarrow N$ such that if computation at vertex $x \in D^n$ depends on those at vertex $y \in D^n$, then $t(x) > t(y)$. Given $t$, the total execution time is $T(n) = Max(t(x); x \in D^n)$. Usually we want to determine a schedule $t$ such that $T(n)$ is minimal. We find here that node $(i, j, k)$ can be scheduled at time $i + j + k - 2$. This has also been proven to be the unique time-optimal schedule [3].

To obtain systolic designs, there remains the task of choosing an allocation function that maps nodes to physical processors while preserving the depen-

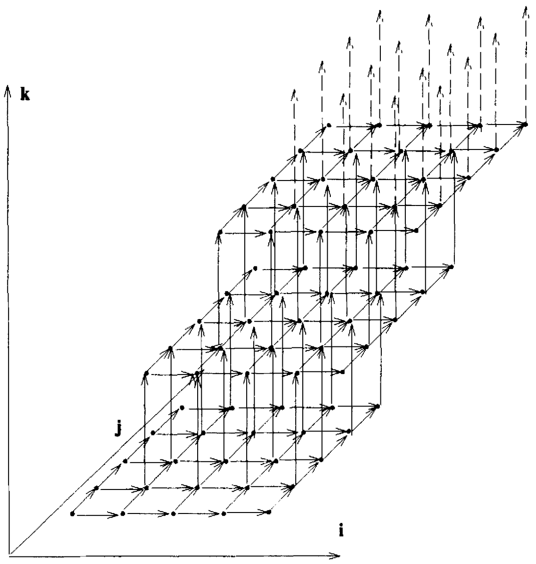


**Figure 3** Uniform DG for the APP.

dences. The simplest solution is to take the allocation function *alloc* to be linear, which ensures a good regularity and local interconnections in the final architecture. More precisely, we take *alloc* to be an orthogonal projection along a direction $u$, where $u$ is a vector which is not orthogonal to isochronous hyperplanes. If we choose $u = (0, 1, 0)$, we have $alloc(i, j, k) = (i, 0, k)$, and we obtain the Robert-Trystram [34], Kung-Lo-Lewis [23], and Lewis-Kung [22] solutions. If we let $u = (0, 0, 1)$, we obtain a solution where coefficients are updated in place, which implies to include an additional time needed to load and unload the array.

In collaboration with our IRISA partner, we have succeeded in applying the dependence mapping strategy to retrieve the folded array of section 4 in a systematic way [2]. Automated synthesis techniques that include extensions to the basic linear scheduling and allocation techniques, and that can handle the APP demonstrator, are also presented in chapters 4, 5, and 6. Moreover, if several different processor elements with different modes have to be dealt with, they need to be controlled by local controllers. For this purpose, control signals have to be distributed, as discussed in chapter 4. One particular extension that is very useful for decreasing the execution time even further is discussed in the next subsection.

## 3 Bounded broadcast facility

The question is the following: how can we decrease execution time even further? From the problem specification, we know that node $(k, k, k)$ must be computed at (in fact, cannot be computed before) time $3k - 2$. If we want to improve on the bound $5n - 2$, we have to compute several nodes $(i, k, k), i > k$ in parallel, and then to update several nodes $(i, j, k)$, $i, j > k$ in parallel.

In well-known systolic solutions listed in tables 1–3, the pivot coefficient $a_{kk}^k$ (node $(k, k, k)$ in the DG) is computed at time $t(k, k, k) = 3k - 2$ in cell $(k, k)$. Then it is systolically propagated from cell to cell so as to reach cell $(i, k)$ at time $t(k, k, k) + i - k$ for computing $a_{ik}^k := a_{ik}^{k-1} \otimes a_{kk}^k$ (node $(i, k, k)$ in the DG). As already said, node $(k, k, k)$ cannot be computed before step $3k - 2$. But we can use a bounded broadcast facility and propagate the pivot coefficient $a_{kk}^k$ to the next $b$ cells within one single time-step, where $b$ is a parameter to be adjusted. In other words, we still compute node $(i, k, k)$ in cell $(i, k)$, but at time $3k - 2 + \lfloor \frac{i-k}{b} \rfloor + 1$. Scheduling the whole DG is easy: node $(i, j, k)$ is computed at time $3k-2+(\lfloor \frac{i-k}{b} \rfloor + 1) + (j-k)$ in cell $(i, k)$. The total execution time is thus $4n + \frac{n}{b} + o(1)$. By a method similar to those of Benaini-Robert [3] and Cappello-Scheiman [7], we can fold the array so as to use only the first $\lceil \frac{n}{3} \rceil$ rows, thereby reducing the cell count down to $\frac{n^2}{3} + O(n)$ , at the price of increasing the period up to $3n$.

We would like to point out that the scope-$b$ broadcast enables us to parametrize the localization of the DG of figure 2. The parameter $b$ can be viewed either as the maximal *length* of the dependence vectors, or as the maximum number of *copies* of a given variable (the fan-out of the array). Hence, the parameter $b$ can be adjusted to cope with current integration constraints. Further results on the scope-$b$ broadcast transformation are available elsewhere [31, 8, 30].

Another solution to improve the execution time is to compute several indices $(i, j, k)$ in parallel with $i$ and $k$ fixed. Let $c$ be the number of $j$-indices such that nodes $(i, j, k)$ are computed simultaneously for $i$ and $k$ fixed. This leads to an array with the same number of cells as in well-known solutions (namely $n^2$), but with a lower execution time $(4n + \frac{n}{c})$, which comes at the price of an increase in the period $(\frac{3c-2}{c}n \leq 3n$ instead of $n)$.

We now mix both transformations to speed up the computations along $i$ and $j$ indices. We obtain an array of $n^2$ cells with vertical wrap-around connections. Index $(i, j, k)$ is executed at time $t(i, j, k) = 3(k-1) + 1 + \lceil \frac{j-k}{c} \rceil + \lceil \frac{i-k}{b} \rceil + 1$ in cell $(c(i-1) + 1 + ((j-k) \bmod c), k \bmod c)$. The total execution time of the

| Reference | Application | Area | Time | Period |
|---|---|---|---|---|
| $b$-broadcast [31] | General APP | $n^2$ | $4n + n/b$ | $n$ |
| $b$-$c$-transformation [32] | General APP | $n^2$ | $3n + n/b + n/c$ | $n$ |

**Table 4** APP arrays with limited broadcast facility.

array is $T = 3n + \lceil \frac{n}{c} \rceil + \lceil \frac{n}{b} \rceil - 2$, and the period is still equal to $\frac{3c-2}{c}n$. We summarize our results in table 4 [30].

In this section, we have introduced the scope-$b$ broadcast transformation to improve the performances of systolic arrays. The parameter $b$ can be used to parametrize the uniformization of a given DG by the programmer. The parameter $b$ can be viewed either as the maximal *length* of the dependence vectors or as the maximum number of *copies* of a given variable (the fan-out of the array). Therefore, the parameter $b$ can be adjusted to cope with current integration constraints. We have shown how to improve existing solutions using limited broadcast facilities. In the particular case $b = c = 2$, we have proposed an array with a connectivity equivalent to that of a 2-D torus, but with an execution time of only $4n$, a 20% improvement over the best previously known solution. The design methodology can also be extended to cope with limited broadcast facilities [30].

## 6 PARTITIONING ISSUES

Recent work has shown that the synthesis method described in section 5 (based on a projection vector and a scheduling vector) can be extended to generate systolic implementations on a fixed number of processors. The main idea of all these extensions is to merge many cells into a single processor so as to compress the array. This step is called partitioning and can take two different forms: the LPGS (locally parallel globally sequential) form and the LSGP (locally sequential globally parallel) form (see also chapter 4).

The first approach, studied by Moldovan [27], is to partition the array into blocks whose size is the number of available processors, say $p$, and to compute each block, one after the other. The different points of computation in the current block are allocated and scheduled in the $p$ processors, in accordance to the dependence constraints. This method requires small local memories, but a large external buffer is needed to store the data used by the next block.

The second approach, independently studied by several researchers [5, 12, 11], is totally different. A virtual array is obtained by the usual method, and it is then partitioned into $p$ blocks of virtual processors, each block being allocated to one physical processor. Of course, the different points allocated to the same processor have to be computed at different times in the array in such a way that they can be sequentially executed by the physical processor. This method permits synthesizing systolic arrays with a fixed number of cells and, as a particular case, permits improving the efficiency of the cells in a systolic array obtained by the usual projection method. It is described in chapter 4. This method is largely oriented towards applications where the period or throughput is important.

In this section, we present one such method [12, 11], using the APP as a test-vehicle. The main qualities of this approach are threefold:

- It is oriented towards minimization of execution time for a given number of processors.
- It preserves the inherent regularity of the dependence graph by allowing a regular paving of the array with rectangles.
- It reduces the number of communications in the final array, by merging neighboring cells. *Neighbor* here means that there are communications from one cell to the other.

## 1 Projection method and compression

As already stated, the allocation and timing functions in the usual synthesis method are chosen linear. The uniform domain of computation is projected along a projection vector onto an hyperplane, and this allocation plus the timing function describe the array. If we look at the resulting array more precisely, we can remark that the points projected along the projection vector onto the same cell are evaluated periodically, say with a period of $c$ time units. This gives the intuition that $c$ cells could be merged in a same physical processor. This should allow both an increase of the activity of the cells, and a compression of an array: indeed, we just have to slow down the algorithm, i.e., to increase $c$, and then we can compress the array by a factor $c$.

A priori, we could merge any $c$ cells in a same physical processor if they never are active at the same time. However, the best way seems to cluster cells belonging to parallelepipeds whose edges are parallel to the projection of dependence vectors, and this for at least three reasons:

- In general, the array obtained after projection is a regular array with boundaries parallel to certain dependence vectors. Thus, a paving with such parallelepipeds will give a good paving, i.e., will require the smallest number of processors.
- Using boxes with edges parallel to the projection of dependence vectors as a pattern for the partitioning will reduce the number of external communications.
- Merging neighboring cells is better if we want to pipeline many problems on the same array. Indeed, the delay between two problems would be longer if we merge into the same physical processor a cell active at the beginning together with a cell active much later, i.e., available for a second execution later, too.

To describe all the properties of the resulting array in a precise framework, several mathematical concepts must be introduced. They allow for selection of the parallelepipeds that are suitable for a good partitioning. The complete description of the techniques for projection and compression is available elsewhere [11, 12, 13].

## 2 The example of the APP

We will illustrate the entire method on the example of the APP. The initial dependence graph and the computation domain after space-time mapping have been presented in section 5. Assuming that we want a compression factor $c = 3$, the vector projection $\vec{s}$ is chosen in the plane $(\vec{i}, \vec{j})$ such that the input data are projected onto a line which will become one of the boundaries of the array. To obtain the minimal surface for the array, we choose $\vec{s} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$. We complete $\vec{s}$ into a unimodular matrix $S$. $\vec{s} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, thus $S =$

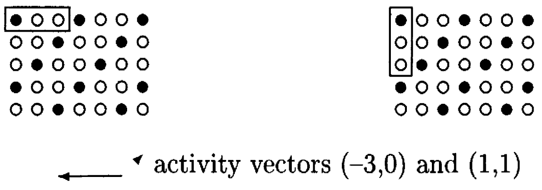


**Figure 4** Two feasible regular partitions. • denotes an active cell, ∘ an inactive one.

$S^{-1} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$. Then we choose a timing function $\vec{\tau}$ such that $(\vec{\tau}; \vec{s}) = c$ and we complete the vector into another unimodular matrix $T^{-1}$. Here the best scheduling vector is $\vec{\tau} = \begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix}$. We find $\begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$, so $T^{-1} = \begin{pmatrix} 1 & 3 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$ and $T = \begin{pmatrix} 1 & -3 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$. The different possible tilings are indicated by the different Hermite forms of the *last* $(n-1) \times (n-1)$ submatrix of $S^{-1}T$. Here, the product $S^{-1}T$ is equal to $\begin{pmatrix} 0 & 1 & 0 \\ 1 & -3 & -1 \\ 0 & 0 & 1 \end{pmatrix}$, which shows that, in the resulting array, $\begin{pmatrix} -3 \\ 0 \end{pmatrix}$ and $\begin{pmatrix} -1 \\ 1 \end{pmatrix}$ form a basis of the lattice of the cells active at the same time. The two Hermite forms of $A = \begin{pmatrix} -3 & -1 \\ 0 & 1 \end{pmatrix}$ are $\begin{pmatrix} 3 & 1 \\ 0 & 1 \end{pmatrix}$ and $\begin{pmatrix} 0 & 1 \\ 1 & 3 \end{pmatrix}$, which leads to two different tilings (see figure 4): a tiling with boxes $3 \times 1$ and a tiling with boxes $1 \times 3$.

Another possibility is to use the projection vector $\vec{s} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$. We have $S = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$, and $S^{-1} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix}$. To get $c = 3$, we take $\vec{\tau} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$, hence $T^{-1} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$, and $T = \begin{pmatrix} 1 & -1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$. The resulting array has $n^2/3 + O(n)$ cells, an execution time equal to $5n - 2$, and a period equal to $3n$. It is, therefore, a new solution for a space-time-minimal array.

## 7 CONCLUSION

In this chapter, we have surveyed recent developments of systolic algorithms and synthesis methods, working out the example of the APP. We have shown how algorithmicians have developed increasingly powerful (but also increasingly complex) solutions. We have also shown how synthesis methods have been successfully extended to cope with these algorithmic/architectural improvements, especially in the field of DG uniformization (by re-indexing and localization), scheduling/allocation techniques for space-time-minimal arrays, and partitioning techniques for the efficient mapping of a computational DG onto a fixed-size processor array.

Many of the new solutions and methods reported here have been derived within the scope of NANA, owing to the cross-fertilization between partners of complementary expertise. Chapters 4, 5 and 6 introduce several novel and practically oriented array synthesis approaches that have been stimulated by our experience with the APP and other demonstrators.

## REFERENCES

[1] H. M. Ahmed, J. M. Delosme, and M. Morf. Highly concurrent computing structures for matrix arithmetic and signal processing. *Computer*, 15, number 1, pages 65–82, 1982.

[2] A. Benaini, P. Quinton, Y. Robert, B. Tourancheau, and Y. Saouter. Synthesis of a new systolic architecture for the algebraic path problem. *Science of Computer Programming*, 15, number 2-3, pages 135–158, 1990.

[3] A. Benaini and Y. Robert. Space-time-minimal systolic arrays for gaussian elimination and the algebraic path problem. *Parallel Computing*, 15, pages 211–225, 1990.

[4] A. Benaini, Y. Robert, B. Tourancheau. A new systolic architecture for the algebraic path problem. In J. McCanny et al., editors, *Systolic array processors*, pages 73–82. Prentice Hall, 1989.

[5] J. Bu, P. Dewilde, and E. F. Deprettere. A design methodology for fixed-size systolic arrays. In S. Y. Kung et al., editors, *Application specific array processors*, pages 591–602. IEEE Computer Society Press, 1991.

[6] P. R. Cappello. A space-time-minimal systolic array for matrix product. In J. McCanny et al., editors, *Systolic array processors*, pages 347–356. Prentice Hall, 1989.

[7] P. R Cappello and C. J. Scheiman. A processor-time minimal systolic array for transitive closure. In S. Y. Kung et al., editors, *Application specific array processors*, pages 19–30. IEEE Computer Society Press, 1990.

[8] P. Cappello and Y. Yaacoby. Bounded broadcast in systolic arrays. Technical Report TRCS88-13, Department of Computer Science, University of California, Santa Barbara, 1988.

[9] Y. H. Cheng and K. Hwang. Partitioned matrix algorithm for VLSI arithmetic systems. *IEEE Trans. on Computers*, C-31, pages 1215–1224, Mar 1982.

[10] L. A. Conway and C. A. Mead. *Introduction to VLSI systems.* Addison-Wesley, 1980.

[11] A. Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *Integration, the VLSI Journal*, 12, number 3, pages 293–304, 1991.

[12] A. Darte and J. M. Delosme. Partitioning for array processors. Technical Report 90-23, LIP-IMAG, Ecole Normale Supérieure de Lyon, France, 1991.

[13] A. Darte, T. Risset, and Y. Robert. Synthesizing systolic arrays: some recent developments. In M. Valero et al., editors, *Application specific array processors*, pages 372–386. IEEE Computer Society Press, 1991.

[14] J. M. Delosme. A parallel algorithm for the algebraic path problem. In M. Cosnard et al., editors, *Parallel and Distributed Algorithms,* pages 67–78. North Holland, 1989.

[15] V. Van Dongen and P. Quinton. Uniformization of linear recurrence equations: a step towards the automatic synthesis of systolic arrays. In K. Bromley et al., editors, *International Conference on Systolic Arrays*, pages 473–482. IEEE Computer Society Press, 1988.

[16] R. Dorairaj and G. Lakhani. A VLSI implementation of all-pair shortest path problem. In *ICPP 87* pages 207–209. S. K. Sahni, editor, The Pennsylvania State University Press, 1987.

[17] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation of combinatorial algorithms. In *Caltech Conference on VLSI*, pages 509–525, 1979.

[18] S. Hansen and J. G. Nash. Modified Faddeev algorithm for matrix manipulation. In *Real-time Signal Processing VII*, pages 39–46. Society of Photo-Optical Instrumentation Engineers, 1984.

[19] S. Y. Kung. *VLSI array processors.* Prentice Hall, 1988.

[20] H. T. Kung and M. S. Lam. Fault-tolerance and two-level pipelining in vlsi systolic arrays. *Journal of Parallel and Distributed Computing*, 1, pages 32–63, 1984.

[21] H. T. Kung and C. E. Leiserson. Systolic arrays for VLSI. In C. A. Mead and L. A. Conway, *Introduction to VLSI systems*, chapter 8.3. Addison-Wesley, 1980.

[22] S. Y. Kung and P. S. Lewis. An optimal systolic array for the algebraic path problem. *IEEE Trans. on Computers*, C-40, pages 100–105, 1991.

[23] S. Y. Kung, P. S. Lewis, and S. C. Lo. Optimal systolic design for the transitive closure and shortest path problems. *IEEE Trans. on Computers*, C-36, number 5, pages 603–614, 1987.

[24] S. Y. Kung and S. C. Lo. A spiral systolic architecture/algorithm for transitive closure problems. *Proc. IEEE Int. Conf. on Computer Design*, Port Chester NY, pages 622–626, Oct 1985.

[25] M. R. Kramer and J. Van Leeuwen. Systolic computation and VLSI. *Foundations of Computer Science IV*, 36, pages 75–103, 1983.

[26] T. Lang and J. H. Moreno. Matrix computations on systolic-type meshes: an introduction to the multi-mesh graph. *Computer*, 24, number 4, pages 32–51, 1990.

[27] D. I. Moldovan. Mapping an arbitrarily large QR algorithm into fixed size systolic arrays. *IEEE Trans. on Computers*, C-35, number 1, pages 1–12, 1986.

[28] P. Quinton. Mapping recurrences on parallel architectures. In L. P. Kartashev and S. I. Kartashev, editors, *Supercomputing ICS 88*, pages 39–46. International Supercomputing Institute, 1988.

[29] P. Quinton and Y. Robert. *Algorithmes et Architectures Systoliques.* Masson, 1989.

[30] T. Risset. Linear systolic arrays for matrix multiplication: comparisons of existing methods and new results. In *Proc. 2nd Workshop on Algorithms and VLSI parallel architecture*, pages 163–174, 1991.

[31] T. Risset and Y. Robert. Uniform but non-local DAGs: a trade-off between pure systolic and SIMD solutions. In *Application Specific Array Processors 91*, pages 296–308. IEEE Computer Society Press, 1991.

[32] T. Risset and Y. Robert. Synthesis of processor arrays for the algebraic path problem: unifying old results and deriving new architectures. *Parallel Processing Letters*, 1, pages 19–28, 1991.

[33] Y. Robert and M. Tchuente. Résolution systolique de systèmes linéaires denses. *RAIRO Modélisation et Analyse Numérique*, 19, pages 315–326, 1985.

[34] Y. Robert and D. Trystram. Systolic solution of the algebraic path problem. In W. Moore, A. McCabe, and R. Urquhart, editors, *Systolic Arrays*, pages 171–180. Adam Hilger, Bristol, 1987.

[35] G. Rote. A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion). *Computing*, 34, pages 191–219, 1985.

[36] J. D. Ullman. *Computational aspects of VLSI.* Computer Science Press, 1984.

4

# HIFI: FROM PARALLEL ALGORITHM TO FIXED-SIZE VLSI PROCESSOR ARRAY

**Peter Held, Patrick Dewilde**
**Ed Deprettere, Paul Wielage**

*Delft University of Technology*

## ABSTRACT

HiFi is a comprehensive design system for massively parallel computing hardware. It covers the design path from behavioral and algorithmic specification to the definition of an architecture or even an electronic circuit. Conversely, it allows for interactive construction of structural objects that represent computations, their simulation and verification. In principle, HiFi can handle any algorithmic specification, regular, irregular, or partially regular, and it can generate any type of architecture. However, its design functions and the embedded synthesis techniques are geared towards exploitation of regularity to a maximal extent.

## 1 INTRODUCTION

The main problem in designing high speed, massively parallel, electronic hardware is the exploitation of the properties of the selected algorithm to maximal electronic benefit. In the course of the design trajectory, these properties may get lost because the existing mathematical structure drowns in architectural considerations. With HiFi, we have attempted to build a system in which the connection between algebraic and structural properties remain present at each level of refinement. The HiFi design trajectory will not "flatten out" the algorithm, but deals with regular parts in an algebraic fashion. Another key feature of HiFi is *parametrization.* Design steps are mathematical mapping operations that in principle are independent of the parameter instances.

It is a remarkable fact that it is indeed possible to build a design system with such properties! Around 1983–85, the development of the HiFi system was based on the work of Sailesh Rao [23] and S. Y. Kung [18]. The first version of HiFi was built around 1985 [17] and implemented Rao's and Kung's ideas, using a powerful new data model (the HiFi model, which will be presented briefly in this chapter) and object oriented programming. This early system did not, however, address the main problem: how to map large algorithms of arbitrary size, to an architecture of given, fixed size. The breakthrough in solving this problem was obtained in the late 1980s more or less simultaneously by Bu [3] and Thiele [26]. In that period, two central problems were solved: the generation of dependence graphs that are as regular as possible from a loop specification, and the tiling, clustering, and partitioning of regular graphs to a regular architecture. The main point was to keep the operations truly geometric, so that they are *not* dependent on any particular indexing scheme, but exclusively exploit the geometric (i.e., index-independent) properties of the algorithm. From 1989, our efforts have been directed to extending these schemes from regular algorithms to piece-wise regular algorithms, i.e., algorithms consisting of regular pieces with regular interconnects between them. It goes without saying that any algorithm is trivially a piecewise regular algorithm, although this is mainly of theoretical concern due to practical complexity problems. The point is, however, that large regular pieces get special, *parametrized* treatment. These problems were all solved in principle around 1990, and their solutions are now implemented in HiFi as design functions.

A design system will distinguish itself by the way it handles the key concepts *hierarchy* and *abstraction*. HiFi utilizes a generalized signal flow graph (SFG) as unifying object, on which it then superimposes the necessary geometric structures (piecewise regularity, parametrization, ...). A special case of SFG is the dependence graph (DG), which is a faithful, architecture-independent representation of an algorithmic specification. The actual architecture is itself represented as a generalized SFG. The main problem that we faced when designing HiFi was how to represent hierarchies of piecewise regular SFGs, including hierarchies of interfaces. We believe that we have solved this difficult problem in a satisfactory, index-independent, geometric way. Other researchers have developed similar approaches [24].

In the remainder of the chapter we will make a semi-formal tour of HiFi. We will first introduce the general design philosophy in section 2. Next, the algorithm specification model is introduced, and a realistic demonstrator from document processing is presented at this level (section 3). The most important architecture-level elements of the design model are discussed in section 4. This

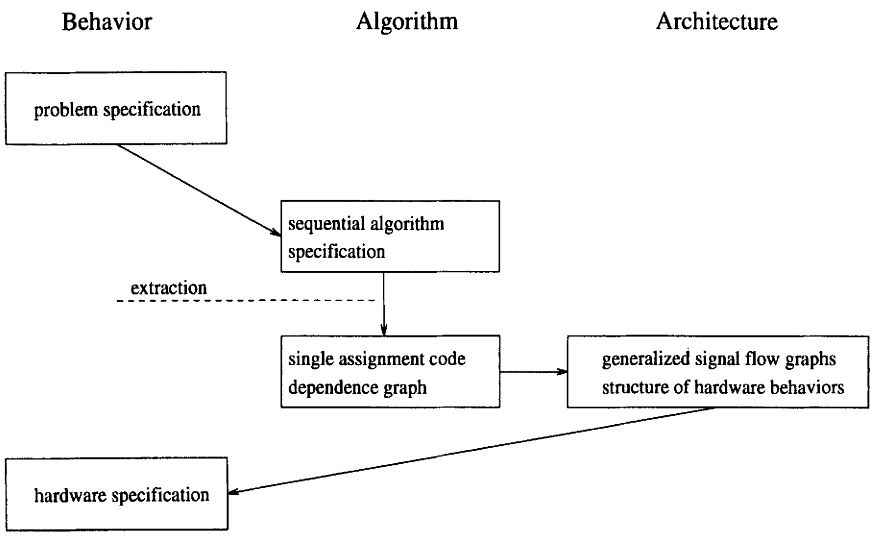

Behavior
Algorithm
Architecture
problem specification
sequential algorithm specification
extraction
single assignment code dependence graph
generalized signal flow graphs structure of hardware behaviors
hardware specification

**Figure 1** The three kinds of design descriptions.

is then followed by the main synthesis-related design functions, i.e., flow-graph extraction, space-time transformation, and partitioning-clustering in sections 5 and 6.

## 2 DESIGN PHILOSOPHY

The HiFi system allows the designer to move interactively from one level of specification to another, either in a top-down or in a bottom-up manner. The top-down direction typically starts out from a behavioral specification of the problem (described in chapter 2) and refines it to a structure of lower level behaviors according to a chosen algorithm.

As shown in figure 1, HiFi contains representations for objects of the type *behavior, algorithm,* and *architecture.* The last category will typically be represented by a generalized kind of signal flow graphs (SFGs). For the algorithmic level, we will have objects like *nested loop programs, single assignment forms,* and *dependence graphs.* Behavioral descriptions are typically given from outside the system: at the top level as specification of the problem, at the bottom level as specification of hardware to be used. The task of the design trajectory is to transform the specified behavior (which is assumed to be given here) into a complete architecture of hardware specifications. Thus, we have to start with the search for an algorithm that implements the behavior and is suitable for realization in hardware.

## 3 ALGORITHMIC SPECIFICATION

We can describe an algorithm as "some composition of primitive nodes with a functional behavior which can serve as a realization of the given more complex behavior." This part of the trajectory requires the most inspiration from the designer, since it cannot be formalized and the choices greatly influence the final performance of the resulting hardware. Aspects of the algorithm that must be taken into account here are: parallelism, regularity, numerical stability, and simplicity of its constituting primitive nodes [14]. We detail the algorithm to such an extent that it consists of a composition of primitive nodes, which have a known implementation in the hardware domain.

Typically, an algorithm will be specified in the form of a *nested loop program* (NLP) [3]. It is useful to include the specification of a nested loop program as an integral part of the design trajectory, mainly because it can be specified in an easily comprehensible syntax and because such a program often can be found in a library.

Unfortunately, an NLP does not explicitly show the parallelism of the computations. Therefore, the first step in the design trajectory is to convert the NLP into a dependence graph (see chapter 2) [4]. The resulting DG will have a *node* for each instance of the calculation of a function, e.g., *Floyd* in the example below. Its *edges* will represent the passing of arguments between those functional nodes and hence implement "function calls." We will discuss the conversion step in more detail in section 5. Similar functionality is provided in the approach of chapter 5 but for a partly different target domain.

The demonstrator that will be used throughout this chapter is the Floyd-Steinberg algorithm for the half-toning of documents. The problem to be solved is the conversion of an image of $n$-bit pixels to an image of 1-bit pixels while retaining as much of the image quality as possible [21]. Such an algorithm is needed, for instance, when one wants to print a photograph on white paper using only black ink. When an $n$-bit pixel is converted into a 1-bit pixel, e.g., through thresholding, an error is made. In the Floyd-Steinberg algorithm, this

```
for $j = 1$ to $N$
    for $i = 1$ to $M$
        $\begin{pmatrix} z(j,i) \\ ea(j,i) \\ eb(j,i) \\ ec(j,i) \\ ed(j,i) \end{pmatrix} = Floyd \begin{pmatrix} g(j,i) \\ ea(j,i-1) \\ eb(j-1,i-1) \\ ec(j-1,i) \\ ed(j-1,i+1) \end{pmatrix}$
    end
end
```

**Figure 2** The nested loop program in pseudo-MATLAB code for the Floyd-Steinberg algorithm.

```
function [ $z, ea, eb, ec, ed$ ] = Floyd( $g, ea, eb, ec, ed$ )
begin
    $E = (7 * ea + 1 * eb + 5 * ec + 3 * ed)/16$;

    $v = g + E$;        // v is the "corrected" pixel value.

    // T(hreshold) = 128. w is the value of v, rounded to 0 or 255.
    if $v < T$ then $w = 0$;$z = 0$; else $w = 255$; $z = 1$;
    $e = v - w$;
    $ea = eb = ec = ed = e$;
end
```

**Figure 2** *(continued)* Floyd-Steinberg algorithm.

error is propagated to the not yet converted pixels in the image neighborhood and is used there to compensate for the error previously made. The simplified NLP of the algorithm for an image of $M \times N$ 8-bit pixels is shown in figure 2, where $g(j,i)$ is the input and $z(j,i)$ is the output. The algorithm can be tuned by changing the weights of the propagated errors or by changing the threshold value in the function *Floyd*.

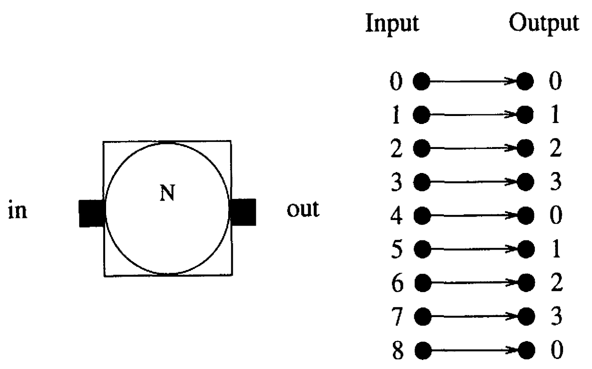


**Figure 3** The modulo-4 node with its behavior defined as an input-output mapping between its input data and expected output data.

## 4 ARCHITECTURE MODEL

However complex an algorithm may be, it ultimately has to be executed by a collection of primitive hardware units, which HIFI considers to be finite state machines. We model a finite state machine by an *AST node.* This architectural model was inspired by the concept of applicative state transitions (ASTs) as published by Backus [2], and, in acknowledgement of that fact, we have named our node accordingly.

The capabilities of a primitive hardware unit are specified by a number of behavior descriptions corresponding to each of its states. The behavior can change from state to state (e.g., an ALU may in one state execute a logic function and in another an arithmetic operation). Therefore, we say that an AST node has a temporal behavior.

HIFI looks upon each part of a hardware unit that carries out a specific behavior as a black box with a number of input and output ports. Its behavior is defined by a relation between the data supplied at its input ports and the expected data at its output ports. If we do this for all possible input values, the behavior, as it reveals itself to the outside world, is completely specified without restricting the eventual realization. For instance, we can specify a modulo-4 operator by writing down all the combinations of input-output data. Suppose that the possible input values are the numbers 0 to 8. Then its behavior is defined by the input-output table included in figure 3.

In many cases, an explicit specification of the behavior in the form of an input-output table is infeasible. HiFi allows an implicit specification by means of a function description, whose structure, however, need not be relevant to the ultimate architecture. This is, for instance, the case for the function *Floyd* in the algorithm introduced in section 3.

The second type of node in HiFi expresses parallelism and is called a structure node. It is a network of nodes, of either type, in which ports of nodes are connected by edges. The nodes in such a network are concurrent processes that are only aware of their local state and communicate asynchronously with each other by edges. The communication between the AST nodes of such a network is according the model of communicating sequential processes (CSP) [15]. In the context of CSP, communication is seen as a shared event between two subprocesses. This means that the production of data in one AST process is synchronized with the consumption of data in another AST.

## 5 DESIGN TRAJECTORY

We can now think of two extreme conceptual architectures to execute an algorithm specified by a DG. At one end, we have an architecture consisting of a structure node that has been obtained by direct mapping of each node of the DG onto an AST node and of each dependency of the DG onto an edge. This corresponds to parallel execution of the algorithm. In HiFi, a design is actually entered as such an architecture. This is feasible because we focus on regular architectures that can be described in a reduced way, as shown below. At the other end, we find an architecture consisting of a single AST node that has a state for each node of the DG where it executes the appropriate function. This corresponds to sequential execution of the algorithm. This last architecture is very costly in number of time steps needed for execution of the algorithm, while the first one is expensive in the amount of hardware (space). The goal of the HiFi design trajectory (see figure 4) is to allow the designer to weigh the costs of various choices. The designer can continuously make a tradeoff between the time, space, and memory needed by the array processor. The processor array will be designed in such a way that its size is independent of the "size" of the algorithm, the processors are optimally used, and its throughput rate is balanced with the I/O speed of the host processor.

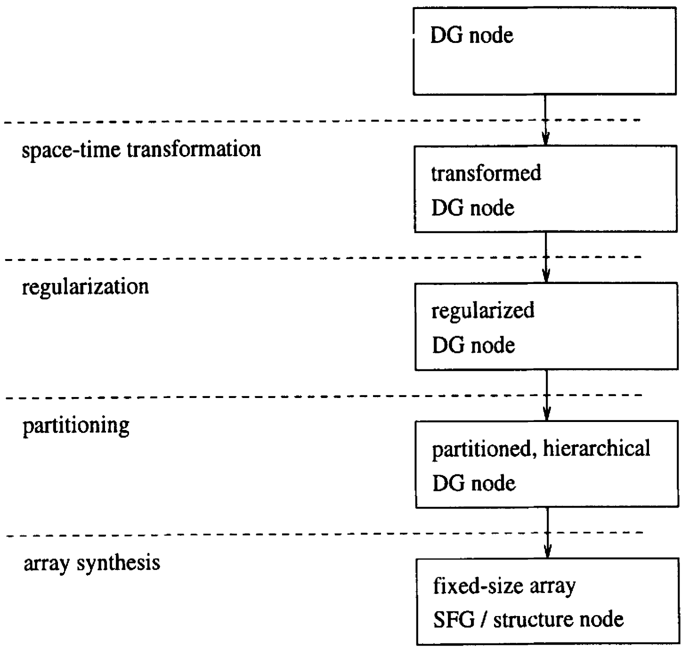

DG node
space-time transformation
transformed DG node
regularization
regularized DG node
partitioning
partitioned, hierarchical DG node
array synthesis
fixed-size array SFG / structure node

**Figure 4** Overview of the HiFi design trajectory for regular processor arrays.

## 1 Regular DG node

Since we focus here on a regular design, the structure node has special constructs for exploring regularity. This is achieved by arranging the network elements in a number of sets of indexed elements [11, 20].

**Definition 4.1** *Let $L$ be an $n \times m$ integer matrix and $O$ be a vector in $\mathbf{Z}^n$, where $Z$ denotes the set of integral numbers. A* **lattice** *$l(L,O)$ is a set of regularly spaced integral points $I \in \mathbf{Z}^n$ characterized by the relation:*

$$I \in l \Leftrightarrow \exists K \in \mathbf{Z}^m : I = L \cdot K + O$$

**Definition 4.2** *Let $l$ be a lattice. A* **domain**, *$D$, is a set of integral points $I$ of the lattice $l$ enclosed by a polytope. Let $A$ be an $r \times n$ integer matrix and $C$ be a constant vector in $\mathbf{Z}^r$. Then a domain is characterized by:*

$$I \in D \Leftrightarrow I \in l : A \cdot I \leq C$$

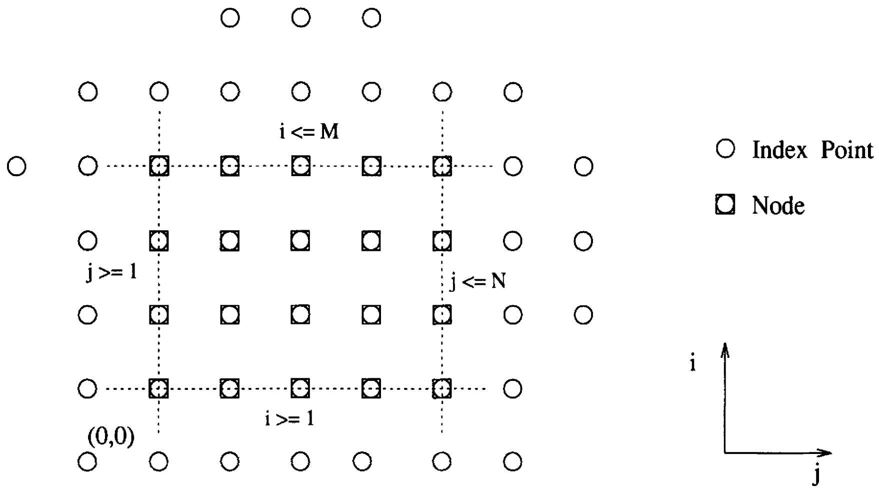


**Figure 5** The node domain of the Floyd-Steinberg algorithm, for $N = 5$ and $M = 4$. The dotted lines are the half planes, which are derived from the bounds of the FOR-loop statements.

We now consider a structure node in which the set of nodes is partitioned in a number of subsets. Nodes in a subset have identical behavior. These subsets are called *node domains.* Let $D$ be a domain and $N$ be a type of node. Then a node domain $V_{ND}$ is characterized by the pair $(N, D)$ and consists of an indexed set of nodes $V_{ND} = \{N_I | I \in D\}$. The node domain of the Floyd-Steinberg algorithm is given in figure 5. The boundary planes of the domain are derived from the upper and lower bound expressions of the FOR-loop statements of its nested loop program.

In a similar way, we define a so-called *port domain* as a set of indexed ports, denoted by $(p, D)$ with $p$ an input or output port of a node. There are four types of port domains: *input* and *output* port domains are used to define the interface of a structure node; *destination* and *origin* port domains are used to define the edges of a regular node by means of dependence relations.

**Definition 4.3** *Let $(p_{in}, I_{in})$ be an element of a destination port domain $(p_{in}, D_{in})$ and let $(p_{out}, I_{out})$ be an element of an origin port domain $(p_{out}, D_{out})$. A* **dependence relation** *defines a set of edges $E'$ by a function $F$ from the destination port domain to the origin port domain, as follows:*

$$(p_{in}, p_{out}) \in E' \Leftrightarrow \forall I_{in} \in D_{in} \exists I_{out} \in D_{out} : I_{out} = F(I_{in})$$

*We require the function $F$ to be affine. Let $A$ be an $m \times n$ integer matrix and let $C$ be a constant vector in $\mathbf{Z}^n$. Then $F$ will be of the form:*

$$F(I_{in}) = I_{out} = A \cdot I_{in} + C$$

In our demonstrator, the functions of the dependence relations are constant: $F_{ea} = (0, -1)^t, F_{eb} = (-1, -1)^t, F_{ec} = (-1, 0)^t$ and $F_{ed} = (-1, 1)^t$ (see figure 8 for an example).

A structure node is called a *regular node*, if its set of nodes is completely defined by a collection of node domains and its set of edges is completely defined by a collection of dependence relations. We now define a *dependence graph node*, denoted *DG node*, as an acyclic regular node with the property that each element executes exactly once. It represents an architecture that has a one-to-one correspondence with the algorithmic specification given in the form of a dependence graph (DG). More specifically, a DG node is a structure node with characteristic properties overloaded from the DG. This means that each of its AST nodes is fired exactly once and that each edge carries exactly one data token.

## 2 Flow dependence extraction

We will now provide the procedure for converting a nested loop program (NLP) into a DG node by extracting the flow dependencies between the operations (iterations) of the NLP. In general, this is a very difficult problem and the procedure we are going to explain will only work for so-called static and linear NLPs. More specifically, the NLP may only contain statements that are of the following type:

- *FOR-statements*, in which the upper and lower bound expressions are affine functions on the loop iterators.
- *IF-THEN-ELSE constructs*, in which the condition is an affine expression.
- *function-call statements*, in which a function is called with a number of right-hand-side variables as arguments and in which the result is assigned to a number of left-hand-side variables.

Furthermore, let $I$ be an index vector containing loop iterators and let $f(I)$ be an affine function on $I$. Then all variables of the program are of the form

```
for $i = 1$ to $N$
    for $j = i + 1$ to $N$
        for $k = i$ to $M$
            if $k = i$ then
                $[A(i,k), A(j,k), phi(i,j)] = \text{Fvectorize}(A(i,k), A(j,k));$
            else
                $[A(i,k), A(j,k)] = \text{Frotate}(A(i,k), A(j,k), phi(i,j));$
            end
        end
    end
end
```

**Figure 6** The MATLAB code for the QR factorization that can be extracted.

$v(f(I))$, with $v$ the name of the variable. The function $f(I)$ is called the indexing function of the variable. Figure 6 shows the QR factorization algorithm as an example of a correct NLP.

The first step in the extraction process is to parse the MATLAB code and to convert it into a number of node domains. We create a node domain for every function-call statement separately. Its domain, the context of the function call, is derived from the expressions of the active loop and conditional statements. The index vector of a node domain is formed by the iterators of the active FOR-statements. For example, Floyd-Steinberg has one node domain, whereas the QR algorithm has two node domains.

Next, we have to find the flow dependence relations that may exist between the functions (nodes). A function is flow dependent on another function if and only if it is called with an argument value that has been produced earlier by the other function. If no such function exists, the data are considered to be input data of the program. We will simplify the explanation by considering only flow dependencies between iterations (they could exist within iterations as well).

To solve the problem [3, 12], we can, fortunately, apply a divide-and-conquer strategy. First we assume that variables with different names can never reference the same data. For instance, the variable $ea(j,i)$ and $ec(j-1,i)$ of *Floyd* can never cause flow dependence relations because they have different names. So, we sort the variables by name into sets of variables with equal names. Next, we divide each such set in a set of result variables and a set of argument variables. The value referenced by an argument variable can only be defined by an assignment via one of its corresponding result variables. Now we first search for dependence relations between the argument variable and each of its result variables separately. After we have found the solution for each such argument-result pair, we combine them to obtain the final result.

Each variable has its own context given by the node domain of its corresponding function-call statement. Let $f$ and $g$ be indexing functions. Let the argument variable be $v(f(I_{arg}))$—for instance, the variable $ec(j-1,i)$—and let one of its result variables be $v(g(I_{res}))$—for instance, $ec(j,i)$. The argument variable is referencing an element of $v$ via its indexing function. The crucial question to be answered is the following: which iteration $I_{res}$ has assigned a value to this element by writing to the result variable?

Let $\prec$ denote the lexicographical ordering operator. We can formulate three conditions that the iteration vector $I_{res}$ must satisfy:

- $g(I_{res}) = f(I_{arg})$, the indexing functions of both variables must reference the same element of the variable.
- $I_{res} \prec I_{arg}$, the assignment to the element at iteration $I_{res}$ must precede its reading.
- The iteration $I_{res}$ must be the lexicographically largest of the set of iterations that satisfy the first two given conditions.

It turns out that these constraints can be formulated as a parametrized linear programming problem [10], in which the feasible set is formed by the context of the result variable and in which the cost function is defined by the lexicographical ordering of the iterations. This latter fact is the key to the solution. Not only are we allowed to define such a non-linear cost function, but it is also smoothly integrated in the way an LP algorithm works. The LP algorithm we are using is called PIP [9].

Note that the indices of $I_{arg}$ enter as parameters into the LP problem. To make this clear, we have depicted the node domain of the Floyd-Steinberg algorithm once more in figure 7. It is divided into two port domains denoted by A and B. The point is that the solution depends on the position of index vector $I_{arg}$. If it lies in port domain A, its dependence relation is just the constant vector: $F_{ec} = (-1,0)^t$. If it lies in port domain B (on the edge), no solution exists. Generally, the solution returned by PIP is a list of pairs consisting of a port

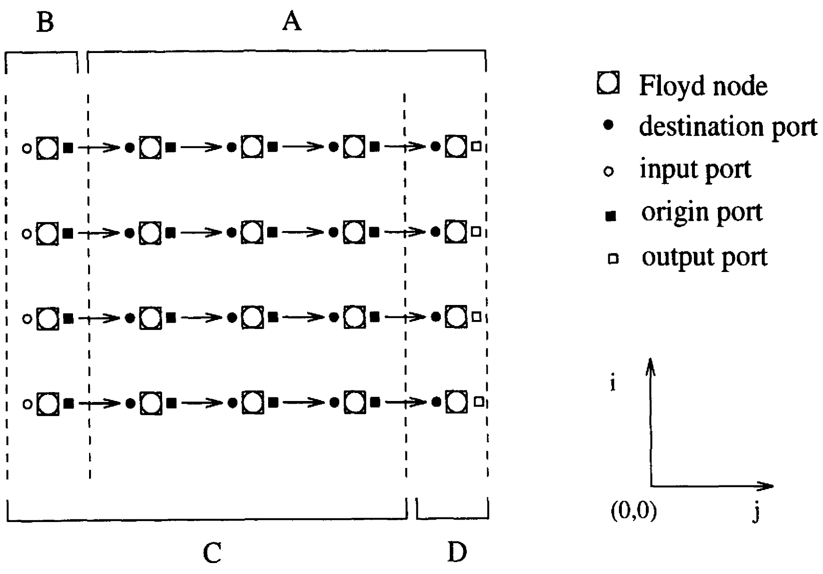


**Figure 7** The port domains found by PIP for the argument-result pair of variable $ec$ of the Floyd-Steinberg algorithm. It has found one dependence relation. Its function is constant and given by $F_{ec} = (-1, 0)^t$, its destination domain is A, and its origin domain is C. B is an input port domain and D is an output port domain.

domain and the function of the dependency found for that domain. In case the function is undefined, the port domain is an input port domain. In all other cases, the domains are destination port domains. Note that the dependence relations are of the form as defined in the section 5. This means that the unknown iteration vector $I_{res}$ is expressed exactly in terms of the known vector $I_{arg}$, which is just what we are searching for.

After solving the PIP problem for each argument-result pair, we have to combine these results. This is a relatively easy step. It consists of finding the lexicographically largest index vector of the set of found index vectors $I_{res}$ [10]. The final result is still expressed in terms of dependence relations. The extracted DG for the Floyd-Steinberg algorithm is given in figure 8. Finally, table 1 lists the CPU time needed by the extraction tool to find the flow dependencies for the Floyd-Steinberg, the QR, and an FIR filter algorithm. It also shows the number of node domains, port domains, and dependencies of the extracted DGs.

| Algorithm | CPU time | # node dom. | # port dom. | # depend. |
|---|---|---|---|---|
| Floyd-Steinberg | 10.0h | 1 | 26 | 5 |
| QR-factorization | 44.4h | 2 | 32 | 9 |
| FIR-filter | 5.7h | 1 | 5 | 3 |

**Table 1** CPU times for flow dependency extraction.

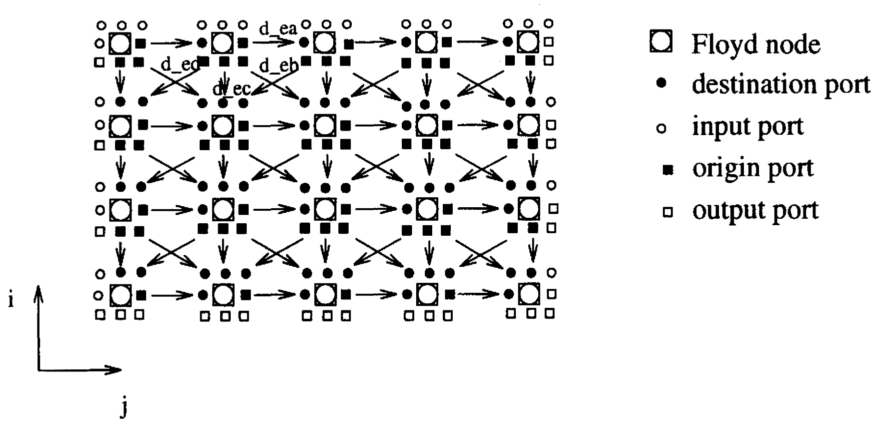


**Figure 8** The DG extracted from the Floyd-Steinberg algorithm. The dependencies d_ea, d_eb, d_ec, and d_ed denote the error propagation.

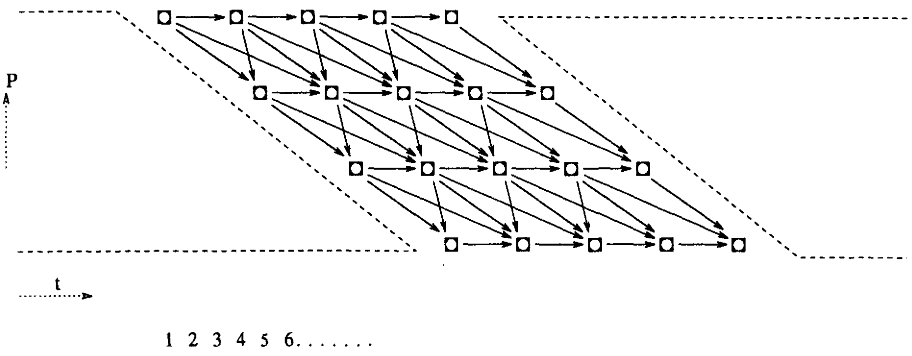


**Figure 9** Transformed dependence graph of the Floyd-Steinberg algorithm, given that $\lambda = [3,4]^t$ and $P = [0,1]^t$.

## 3 Space-time transformation

In general, the extracted DG is not necessarily uniform. In order to remedy this situation, several techniques are discussed in chapters 3 and 6. This leads to uniform recurrence equations or UREs. The next design step is a linear space-time transformation [18, 16, 3]. Chapters 3, 5, and 6 discuss automated techniques to steer this step, including several extensions which are not directly essential for our demonstrator here. The space-time (S-T) transformation step is concerned with the allocation of a processor and a time slot to each node of the DG. To that end, we specify a space-time transformation matrix $T = \begin{bmatrix} P \\ \lambda^t \end{bmatrix}$, in which $\lambda$ is a schedule vector and $P$ is a projection matrix describing the processor assignment. Thus, we reindex each node by transforming its index $\gamma$ to $T \cdot \gamma$. Figure 9 shows the transformed dependence graph of Floyd-Steinberg for $\lambda = [3,4]^t$ and $P = [0,1]^t$. In this figure, the vertical axis can be interpreted as the processor axis and the horizontal axis as the time axis.

In case of a systolic array, the schedule vector $\lambda$ specifies that the AST node at index point $\gamma$ is to be executed at time step $\lambda^t \cdot \gamma$. The vector $\lambda$ also specifies that the data which are transmitted over an edge defined by a dependence vector $d_i$ is delayed $\lambda^t \cdot d_i$ time steps, so that the data will arrive in time at the node of the DG by which they are to be processed. In case of a wavefront processor array, the schedule vector $\lambda$ only specifies the ordering of the computations and the data storage capacity of the edges. Furthermore, the node at index point $\gamma$ will be mapped onto the processor at index point $P \cdot \gamma$ during the array synthesis step. Note that the S-T transformation only defines the S-T allocation conceptually. The actual allocation is decided on during array synthesis (see section 6).

## 4 Regularization

At this point, we have obtained a dependence graph consisting of regular subgraphs in each of which a different operation takes place. If these parts have to be mapped to the same array processor, a scheduling and memory allocation problem arises, which we will now address.

One way to solve this problem is by replacing each node in the DG by a compound node that is capable of performing any one of the possible behaviors of the nodes. This corresponds to a processing element (PE). The selection of the proper behavior of a compound node is then done by supplying control data to the regularized DG. The process of combining all possible behaviors into an array consisting of a set of identical compound nodes is called *regularization*. The compound node can either be described by an AST node or by a structure node (hence hierarchy!).

The control edges that are necessary for this selection process are "woven" into the dependence graph. This is done in such a way that each node domain and port domain in the original DG node can be uniquely identified by the data at these control edges. For each boundary between domains, a "one bit" control dependency is added to the new dependence graph that runs in parallel with this boundary [25, 27]. Figure 10 shows the regularized version of the dependence graph of Floyd-Steinberg.

One way of implementing the compound node is by a structure node, which contains all the different functions as individual nodes, a number of input and output selectors that provide for the routing of data from and to the nodes, and a number of control edges. Figure 11 shows the structure node of the Floyd-Steinberg DG. The dashed edges in the figure are the edges that control the data flow in the structure node. Another, more effective, way is to combine the functions into a single AST, as, for example, is done in an ALU.

A DG may be regularized in many ways: not all parts of the graph have to participate. We have defined a design function "regularize" which allows the designer to select parts that must be merged. It will automatically generate a new global part encompassing the selected pieces.

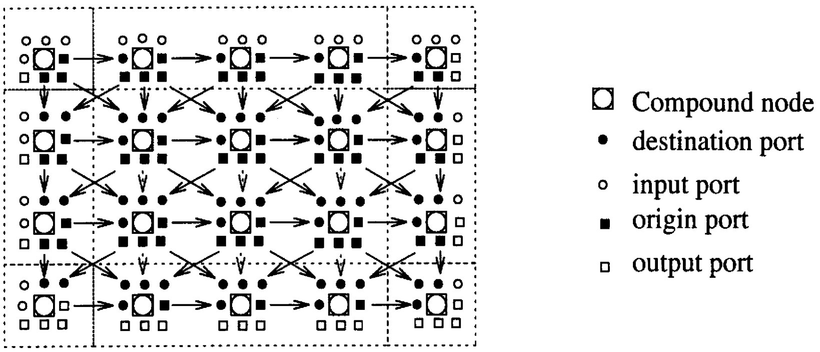


**Figure 10** Regularized version of the piecewise regular DG. Inside the dotted boxes, the control data that arrive at the ports of the compound nodes are the same. The control data define the actual functionality of the compound node.

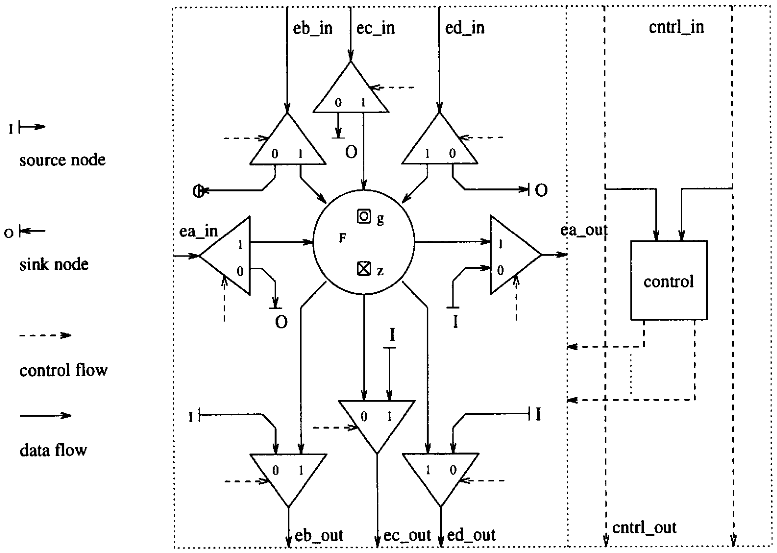


**Figure 11** Compound node of the regularized Floyd-Steinberg DG implemented as a structure node. The triangular nodes denote input (i_*xxx*) or output (o_*xxx*) selectors. The circular node is the original AST node. The bars denote source and sink nodes.

## 6 FIXED-SIZE ARCHITECTURE DESIGN

In general, the regularized DG is still not suited to be mapped directly on hardware because it would result in a processor array of a size dependent on the parameters in the original description. This is, for instance, true for the Floyd-Steinberg demonstrator where the parameters $M$ and $N$ can vary over wide ranges.

### 1 Partitioning

We have defined a design step *partitioning-clustering* [5, 6, 7] to decompose the regularized DG in a network of similar reduced-size DGs, called tiles. We call this network the tile graph. An alternative approach, based on the same principles but tuned towards latency-driven applications, is discussed in chapter 3.

In figure 12, we illustrate the partitioning procedure. The DG is cut by $n$ sets of uniformly spaced and parallel hyperplanes that have $\lambda_1, \ldots, \lambda_n$ as normals. These partitioning planes are always chosen parallel to the boundary planes of the regularized DG and their interspacing is chosen such that all tiles have equal size. The regularity in function, dependency, and shape guarantees that both the tile and tile graph can be described, in a reduced way, by DG nodes. The construction of both graphs starts with selection of the partition planes that define the shape of the tile. Generally, this shape will not be congruent with the DG. Therefore, it may be necessary to adjust the shape of the DG by extending it with nodes. This shape adjustment is considered as a form of regularization.

The tile has one node domain and is structurally equivalent with the regularized DG. The node domain of the tile graph iterates the tiles. It is the same as the domain of the original DG, but spread out on a lattice with the size of the tile. Figure 12 also shows that dependencies cut by partition planes result in intertile dependencies at the tile graph level, and in input and output ports at the tile level. Note that these ports form the interface of the tile. The intertile dependencies are hierarchical and composed of edges between the ports of the port domains of the tile.

### 2 Partitioning schemes

After partitioning, we have achieved a hierarchical decomposition of the DG in two DGs, the tile graph and the tile, both having a reduced and possibly

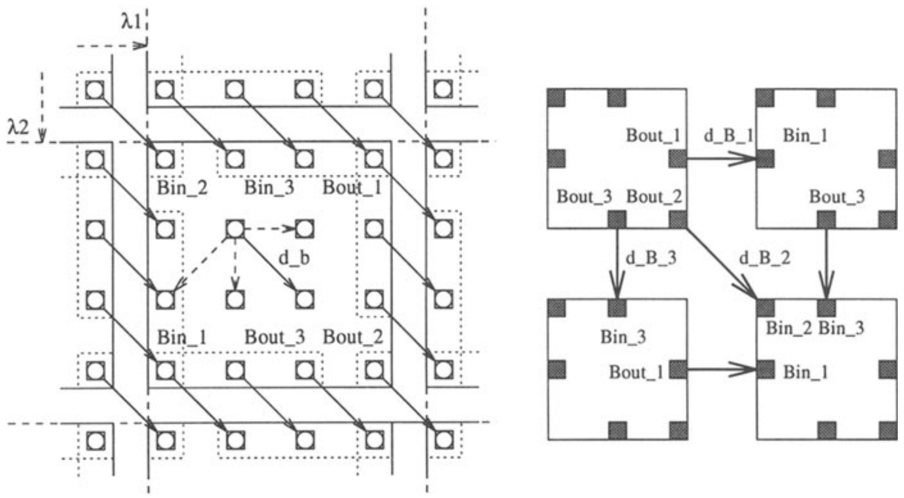


**Figure 12** A part of the Floyd-Steinberg DG partitioned by two set of parallel and equidistant hyperplanes (left). In order to reconstruct, e.g., the dependency d_eb $= (-1, 1)^t$ at tile graph level, the tile needs three input and output ports, Bin_i and Bout_i, respectively. On the right, the figures show the hierarchical dependencies d_B_1, d_B_2, and d_B_3, that complete the reconstruction of d_eb at tile graph level.

fixed size. This allows us to apply the space-time transformation step at both levels independently. There are various schemes for doing so. Each results in an architecture with its specific memory, control, and processor structure. The two basic schedule and allocation schemes [16] are:

- Locally Sequential, Globally Parallel (LSGP)

  Here, the tiles are processed in parallel on a fixed-size processor array of which each processor executes the operations of a tile sequentially. Consequently, a processor of the array needs internal memory to store intermediate data of a tile. This scheme is used for increasing processor efficiency or slowing down I/O rate.

- Locally Parallel, Globally Sequential (LPGS)

  This scheme works the other way around. Here, the tiles are sequentially processed on a fixed-size array that executes the operations of a single tile in parallel. It needs memory external to the array to store data for another tile. Typically, LPGS is used to execute huge and parametrized algorithms on a relatively small and fixed size array.

## 3 Final array synthesis

At this point we have reached the stage where, via regularization and partitioning, a simple dependence graph at the top hierarchical level has been obtained, for example, as shown in figure 12. The tile graph is a regular structure of tiles that have a reduced and possibly fixed size. However, the tile graph still contains an individual node for each computation and is, therefore, not a realistic architecture in which processor elements are reused as much as possible. To that end, we introduce the "array synthesis" step that projects the DG onto a fixed-size architecture. Obviously, the nodes of the resulting architecture no longer possess the property of single execution. Consequently, this step will always be the final one of our trajectory. The synthesis step must fit within the concept of hierarchy and stepwise refinement.

Array synthesis is inextricably tied with S-T transformation in the sense that it realizes the array as has been defined conceptually during S-T transformation (see section 5). The procedure is straightforward and is based on the projection of nodes and dependencies (with similar properties) onto each other. Due to S-T transformation, the dimensions of each node domain are tagged with either a *space* or an *ordering* interpretation. The processor space of the resulting array is obtained by projection of the node domains on the space dimensions. The execution order of the operations projected onto a single processor is given by the time dimensions. Figure 13 illustrates the synthesis of the Floyd-Steinberg tile. It shows the tile before and after projection. The port domains, which form the interface of the DG tile, are not projected but are converted into so-called port adaptors.

In order to understand the function of port adaptors, suppose that an output port of a tile is connected to an input port of another tile by a hierarchical edge and that we have projected both tiles. We have illustrated this in figure 14. It shows the input and output adaptors corresponding to the input and output ports, respectively. The edges between the adaptors form the intermediate tile storage. The processor on the left writes to this storage by selecting the appropriate edge via its output adaptor. The processor in the tile on the right reads from this storage via its input adaptor. For addressing the proper storage location (edge), the processor has its own local index control, based on the scheduling information. This way of interfacing allows us to define a S-T allocation for each node (tile) individually, at the expense of a complex random access selection mechanism.

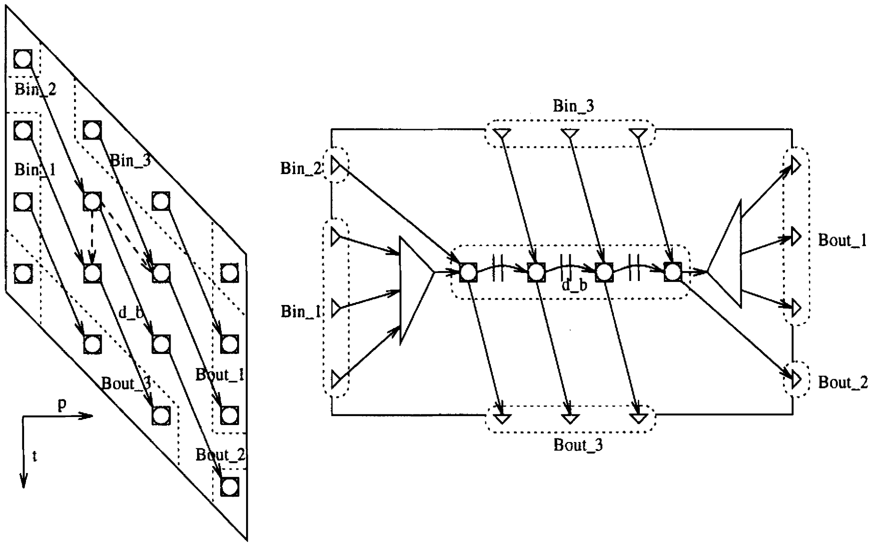


**Figure 13** The result of array synthesis (right) of the space-time transformed Floyd-Steinberg tile (left). The triangles are the adapters that resolve the fan-in and fan-out problems of projection.

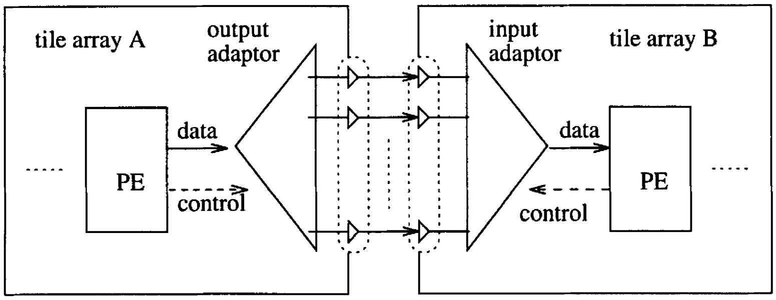


**Figure 14** Interconnection structure between two processors, resulting after projection of two connected tiles. The triangles represent the IO adapters. The edges between the adapters form the intertile storage. Processor A writes to the storage via its output adaptor. Processor B reads from the storage via its input adaptor.

In case the S-T allocation of the tiles are the same, the interface structure can be simplified to a FIFO because the order of writing and reading is similar although with unknown time delay. In our case, this requirement will always be fulfilled because the tiles are the same after partitioning.

## 7 CONCLUSION

One major conclusion from our efforts in building HiFi stands out: it is indeed possible to build a design system that covers the complete trajectory from behavioral specification to electronic hardware and utilizes just one consistent set of mathematical concepts for doing so. Because of its uniformity and the way it handles regularity, the model we use caters for hierarchy and abstraction in a natural way. Object-oriented programming takes care of uniformity in documentation and representation, and the possibility of extensive interaction [14]. Most important, however, is the automation of synthesis tasks made possible by the mathematical construction. The results for the Floyd-Steinberg demonstrator clearly illustrate the importance of the dependence graph extraction, the regularization and partitioning steps, and the design automation techniques that have been outlined in this chapter.

## REFERENCES

[1] J. Annevelink. *HIFI, a design method for implementing signal processing algorithms.* PhD thesis, Delft University of Technology, Jan 1987.

[2] J. Backus. Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Comm. ACM*, 21, pages 613–641, Aug 1978.

[3] J. Bu. *Systematic design of regular VLSI processor arrays.* PhD thesis, Delft University of Technology, May 1990.

[4] J. Bu, E. Deprettere, and L. Thiele. Systolic array implementation of nested loop programs. *Proc. Int. Conf. Application Specific Array Processing*, Vol. 4, pages 31–42, Sep 1990.

[5] J. Bu and E. Deprettere. Processor clustering for the design of optimal fixed-size systolic arrays. *Algorithms and Parallel VLSI Architectures*, Vol. A, pages 341–362, North Holland, Elsevier, Amsterdam, 1991.

[6] E. Deprettere. Example of combined algorithm development and architecture design. *Proc. Advanced Signal Processing Algorithms, Architectures, and Implementations III*, San Diego, California, July 1992.

[7] E. Deprettere. Cellular broadcast in regular array design. *Proc. VLSI Signal Processing Workshop*, Computer Science Press, 1992.

[8] P. Dewilde and E. Deprettere. Architectural synthesis of large, nearly regular algorithms: design trajectory and environment. *Annales des télécommunications*, Vol. 46, pages 48–59, 1991.

[9] P. Feautrier. Parametric integer programming. *Recherche Opérationnelle; Operations Research*, 22, number 3, pages 243–268, 1988.

[10] P. Feautrier. Data flow analysis of array and scalar references. *Recherche Opérationnelle; Operations Research*, 1991.

[11] F. Fernandez and P. Quinton. *Extension of Chernikova's algorithm for solving general mixed linear programming problems.* Internal report, IRISA, Rennes, France, 1988.

[12] P. C. Held. *HiPars Users' Guide.* Internal report, Delft University of Technology, Delft, Nov 1991.

[13] P. N. Hilfinger. *Silage: a language for signal processing.* Internal report, University of California, Berkeley, 1984.

[14] A. J. van der Hoeven. *Concepts and implementation of a design system for digital signal processor arrays.* PhD thesis, Delft University, Delft, October 1992.

[15] C. A. R. Hoare. *Communicating sequential processes.* Prentice-Hall International, 1985.

[16] A. K. Jainandunsing. *Parallel algorithms for solving systems of linear equations and their mapping on systolic arrays.* PhD thesis, Delft University of Technology, January 1989.

[17] S. Y. Kung, J. Annevelink, and P. Dewilde. Hierarchical iterative flowgraph integration for VLSI array processors. *VLSI Signal Processing*, IEEE Press, New York, 1984.

[18] S. Y. Kung. *VLSI Array Processors.* Prentice-Hall International, 1988.

[19] R. Lipsett, C. F. Schaefer, and C. Ussery. *VHDL: Hardware Description and Design.* Kluwer Academic Publishers, Boston, 1989.

[20] G. L. Nemhauser and L. A. Wolsey. *Integer and combinatorial optimization.* John Wiley & Sons, Inc., 1988.

[21] W. M. Newman and R. F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, 1981.

[22] P. Quinton. *Systolic Arrays.* Esprit Project BRA 3280, Deliverable report INRIA/Y1m12/2.2/1, IRISA, Rennes, France, April 1990.

[23] S. K. Rao. *Regular iterative algorithms and their implementations on processor arrays.* PhD thesis, Information System Lab, Stanford University, October 1985.

[24] J. Teich and L. Thiele. Partitioning of processor arrays: a piecewise regular approach. *INTEGRATION: The VLSI Journal*, 1992.

[25] A. J. Teigh and L. Thiele. Control Generation in the Design of Processor Arrays. *Int. Journal on VLSI and Signal Processing*, 3, number 2, pages 77–92, 1991.

[26] L. Thiele. On the hierarchical design of VLSI processor arrays. *IEEE Symp. on Circuits and Systems*, Helsinki, pages 2517–2520, 1988.

[27] P. Wielage. *The partitioning of piecewise regular dependence graphs.* Master's Thesis, nr 91-109, Delft University of Technology, January 1992.

5

# ON THE DESIGN OF TWO-LEVEL PIPELINED PROCESSOR ARRAYS

**D. J. Soudris, E. D. Kyriakis-Bitzaros**
**V. R. Paliouras, M. K. Birbas**
**T. Stouraitis, C. E. Goutis**

*University of Patras*

## ABSTRACT

This chapter addresses the design of two-level pipelined processor arrays. The parallelism of algorithms is exploited both in word-level and in bit-level operations. Given an algorithm in the form of a Fortran-like nested loop program, a two-step procedure is applied. First, any word-level parallelism is exploited by using loop transformation techniques, which include a uniformization method, if required, and a decomposition of the index space into disjoint sets, which may be executed in parallel. Second, the architecture of the processing element is specified in detail by analyzing its operation at the bit level. Processors using any arithmetic system may be described. The overall design methodology is illustrated by systematically deriving a processor array for the one-dimensional (1-D) convolution algorithm. It is based on an inner product step processor that utilizes residue number system arithmetic.

## 1 INTRODUCTION

Recent advances in VLSI technology have made considerable impact on the design of general- and special-purpose parallel processors for numerically intensive applications, as required in, for example, high-performance scientific computing, involving the solution of difference equations. The same is true in signal and image processing, such as in high throughput radar or sonar applications. The increasing demandfor concurrent processing has led to the development of compilers for multiprocessors [16] and to the hardware implementation of application-specific processor arrays [8].

An algorithm comprises two levels of parallelism: the word level and the bit level. The parallelism of the word-level operations affects the topology of the processor array that implements the algorithm, while the parallelism at the bit level affects the functionality and the structure of each processing element (PE) of the array. The topic of word-level parallelism has been extensively studied during the last decade, and many systematic methodologies have been presented in the literature for mapping various classes of iterative algorithms onto regular processor arrays, and particularly onto systolic arrays. These methodologies embody fundamental concepts from the seminal work on uniform recurrences by Karp et al. [5] and the hyperplane and coordinate methods of Lamport [10] as well as the systolic array concept [7]. The existing methodologies can be grouped into three broad categories with respect to the initial description of the algorithm that they can manipulate. Methodologies of the first category handle algorithms expressed in terms of uniform recurrence equations (UREs) or regular iterative algorithms (RIAs), which are characterized by constant dependencies [8, 12, 19, 25]. The second category models the algorithm using affine recurrent equations (AREs), single assignment codes (SACs), or weak single assignment codes (WSACs), whose dependencies may be linear functions of the loop indices [17, 24, 18, 20]. Finally, there are a few methodologies [2, 1] that use imperative nested loops for algorithm specification.

The aforementioned methodologies exploit the inherent parallelism and pipelining of the algorithm only at the word level; they do not address the detailed design of the PE, which performs an operation within the array. An exception to this is described in chapter 6, but also there the existing parallelism at the bit-level has not been thoroughly exploited. Only two major approaches for designing a processor array down to the bit level have been proposed [6, 11]. The first one uses two-level pipelining in a way similar to the methodology presented here, while the second one treats the entire problem at the outset as a bit-level problem. A careful study of these design approaches reveals that they cover a limited number of applications without a formal general description of the array architectures. However, the first approach is superior, since a bit-level design approach can be embedded in various word-level methodologies. The second approach seems rather restricted because there is no formal method for expressing a word-level algorithm in its equivalent bit-level form.

In this chapter, a two-level systematic methodology for synthesis of regular processor arrays is described. In the first step, an algorithm expressed in Fortran-like nested loops is transformed into a URE with parametric dependence vectors (DVs) (see also chapters 6 and 4). The derived URE is mapped to the desired regular array architecture by using a decomposition technique, which is based on the existence of independent subsets of variable instances in the index space [9]. This property has also been used in compiler optimization approaches [3, 15, 21], and provides more efficient solutions compared to other processor array design techniques, which transform the whole index space into processor space and time. When the DVs are linearly independent, data movements are required only between neighboring PEs, while the execution time of the algorithm is optimum or near-to-optimum depending on the available hardware.

The second step is a systematic graph-based methodology for designing the PEs of the array, starting from the bit-level expression of an operation and employing as basic building block the one-bit full adder (FA). The proposed methodology synthesizes bit-level systolic arrays for a wide class of arithmetic and logical operators, which are common in real-life applications. These operators should be described by multiple sums of products and may be expressed in any arithmetic system, e.g., residue number system (RNS) [23], binary 2's complement, or signed-magnitude. A product should be any function of the input bits times any function of powers of two, which is the adopted radix. The target architecture style used here is the regular array processor style. The dependence graph of the bit-level algorithm, which can be descibed formally by a set of UREs, is used for the systematic synthesis of the bit-level architectures.

The remainder of the chapter is organized as follows. In section 2, the transformation technique of a nested loop to URE form is discussed. Next, the mapping of the derived URE to the word-level processor array using a decomposition technique is presented in section 3. The proposed bit-level methodology for deriving the FA-based arrays is then developed in section 4. The synthesis steps in the proposed methodology are illustrated throughout the design trajectory by means of a realistic application: a 1-D convolution algorithm which is used in many scientific applications and might also be employed in other domains, e.g., in string matching for text recognition. Several alternative regular arrays for different system requirements are produced. The PE of the processor array operates using RNS arithmetic to obtain very high throughput. Finally, some conclusions are offered in section 5.

## 2 TRANSFORMATION OF NESTED LOOPS TO UREs

The first goal in our methodology is to transform an algorithm given in nested-loop form, which may include non-constant dependencies, to an equivalent URE with localized parametric DVs. The motivation for this starting point is also discussed in chapter 4. The applicability of such a transformation is restricted by the complexity of the index functions of the feedback variables. Therefore, our attention is focused on WSACs [20], which are characterized by identical linear index functions of the feedback variable. In this approach UREs are derived directly, in contrast to the technique described in chapter 4 where the dependence graph (DG) is extracted.

### 1 Definitions

Definitions of the fundamental terms used throughout this chapter are given next. The general form of a nested loop considered here is:

```
for $i_1 = l_1$ to $u_1$, $step_1$
    ⋮
    for $i_N = l_N$ to $u_N$, $step_N$
        loop body;
    next $i_N$
    ⋮
next $i_1$
```

The set $I^N = \{\mathbf{i} \mid \mathbf{i} = (i_1, i_2, \ldots, i_N), l_j < i_j < u_j,\ j = 1, 2, \ldots, N\}$ is called *index space* or *iteration space* of the loop. Any element $\mathbf{i} \in I^N$ is called an *index vector.* Without loss of generality, it can be assumed that $l_j = 1$ and $step_j = 1$. Any statement of the loop body has the form:

$$X^p(\mathbf{z}(\mathbf{i})) = \Phi(X^1(\mathbf{f}^1(\mathbf{i})), X^2(\mathbf{f}^2(\mathbf{i})), \ldots, X^p(\mathbf{f}^p(\mathbf{i})), \ldots, X^q(\mathbf{f}^q(\mathbf{i})), \ldots)$$

where $\mathbf{f}^q(\mathbf{i}) = (f_1^q(\mathbf{i}), f_2^q(\mathbf{i}), \ldots, f_{n_q}^q(\mathbf{i}))$ and $\mathbf{z}(\mathbf{i}) = (z_1(\mathbf{i}), z_2(\mathbf{i}), \ldots, z_{n_p}(\mathbf{i}))$, with $f_r^q(\mathbf{i})$ and $z_r(\mathbf{i})$, being functions of the loop indices and $n_q$ being the dimension of $X^q$, $n_q \leq N$.

A variable that appears in both sides of a statement is called a *feedback variable.* Any element of the set $\{X^q(\mathbf{f}^q(\mathbf{i})) \mid \mathbf{i} \in I^N\}$ is called a *variable instance.* Variables with common name but different index functions are considered as different variables. The set of the index space points where a variable appears is called the *propagation space* of the variable.

Let a variable instance computed at a point $\mathbf{i}_1$ use the value of a variable instance generated at another point $\mathbf{i}_2$. The vector $\mathbf{d} = \mathbf{i}_2 - \mathbf{i}_1$ is called *dependence vector.* The *dependence graph* (DG) is a directed graph, where each node corresponds to a point $\mathbf{i} \in I^N$ and each edge corresponds to a DV joining two dependent points of the index space.

## 2 Transformation methodology

In order to transform a nested loop with linear dependencies into an equivalent URE form, the propagation space of each variable should be determined and the localization of the data movements should be performed. Generally, propagation exists when the propagation space of a variable instance contains more than one node of the index space. The propagation in the index space can take two different forms, namely *broadcast operations* and *"fan-in"*. More specifically, broadcasting occurs when a value of an instance is distributed to many nodes of the space, while fan-in occurs when different values of an instance are concentrated from many nodes to one node. The latter is the main feature of WSACs [20].

Formally, the propagation space of a variable $X^q(\mathbf{f}^q(\mathbf{i}))$ is the linear subspace of the index space defined by the set of equations

$$\mathbf{f}^q_\kappa(\mathbf{i}) = c_\kappa, \kappa = 1, \ldots, n_q \tag{1}$$

where $c_\kappa$ represents constants. The set of index space points where a variable instance appears is the union of the propagation spaces of all variables addressing the instance.

Propagation of a variable $X^q(\mathbf{f}^q(\mathbf{i}))$ occurs if equation (1) has multiple solutions, which means that at least one of the following two conditions is satisfied:

- $n_q < N$
- There is at least one $f^q_\mu(\mathbf{i}) \in \{f^q_\kappa(\mathbf{i}) \mid \kappa = 1, \ldots, n_q\}$ for which the following relation holds:

$$\overline{f}^q_\mu(\mathbf{i}) = \sum_{\nu=1,\nu\neq\mu}^{n_q} a_\nu \overline{f}^q_\nu(\mathbf{i}) \tag{2}$$

  where $\overline{f}^q_\mu(\mathbf{i})$, $\overline{f}^q_\nu(\mathbf{i})$ are the linear parts of $f^q_\mu(\mathbf{i})$ and $f^q_\nu(\mathbf{i})$, and $a_\nu$ are constants.

Consequently, the dimension $D_s$ of the propagation space of $X^q$ is $D_s = D_1 + D_2$, where $D_1 = N - n_q$ and $D_2$ is the number of $f^q_\mu(\mathbf{i})$ functions satisfying equation (2).

According to the above conditions, broadcasting occurs when a variable appears in the right-hand side of the statement only. Fan-in is associated with feedback variables and exists when the statement can be written in the following form:

$$\begin{aligned} X^p(\mathbf{z}(\mathbf{i})) \quad = \quad & X^p(\mathbf{f}^p(\mathbf{i})) \odot \Phi'(X^1(\mathbf{f}^1(\mathbf{i})), \ldots, X^{p-1}(\mathbf{f}^{p-1}(\mathbf{i})), \\ & X^{p+1}(\mathbf{f}^{p+1}(\mathbf{i})), \ldots, X^u(\mathbf{f}^u(\mathbf{i}))) \qquad (3) \end{aligned}$$

where $\mathbf{f}^p(\mathbf{i}) = \mathbf{z}(\mathbf{i})$ and $\odot$ denotes an associative and commutative operator. The partial results $\Phi'(X^1(\mathbf{f}^1(\mathbf{i})), \ldots, X^u(\mathbf{f}^u(\mathbf{i})))$ of each variable instance $X^p(c_1, c_2, \ldots, c_{n_q})$ of the feedback variable are distributed over the associated propagation space. Each variable instance is updated continuously in all nodes of its propagation space due to $f^p_\tau(\mathbf{i}) = z_\tau(\mathbf{i})$. Since the operator $\odot$ is associative and commutative, the partial results should be combined. The final result is placed in any node of the variable's propagation space.

In this case, the DVs can be seen as integer vectors that express the movement of the value of a variable instance within its propagation space. Consider any two nodes $\mathbf{i}_1 = (i_{11}, \ldots, i_{1N})$ and $\mathbf{i}_2 = (i_{21}, \ldots, i_{2N})$ included in the propagation space of a variable. The coordinates of $\mathbf{i}_1$ and $\mathbf{i}_2$ are not independent since equation (1) holds for both nodes. Therefore, the following equations should be satisfied:

$$\mathbf{f}^q_\kappa(\mathbf{i}_2) = \mathbf{f}^q_\kappa(\mathbf{i}_1), \quad \kappa = 1, 2, \ldots, n_q \qquad (4)$$

The number of the independent equations in equation (4) is equal to $N - D_s$. Therefore, $D_s$ additional equations of the form:

$$i_{2\nu} = i_{1\nu} + d_\nu, \quad \nu = 1, 2, \ldots, D_s \qquad (5)$$

should be used, where the parameters $d_\nu$ are arbitrary integers [1]. The combination of equations (4) and (5) results in solutions for the remaining coordinates. The derived solutions have the form:

$$i_{2\nu} = i_{1\nu} + d_\nu, \quad \nu = D_s + 1, \ldots, N \qquad (6)$$

where $d_\nu = \phi_\nu(d_1, d_2, \ldots, d_{D_s})$ and $\phi_\nu$ is a linear function of $(d_1, d_2, \ldots, d_{D_s})$.

Therefore, the DVs can be written as $\mathbf{d} = (d_1, d_2, \ldots, d_{D_s}, d_{D_s+1}, \ldots, d_N)$, where the first $D_s$ coordinates can be arbitrarily selected, while the remaining $N - D_s$ coordinates are linear combinations of the first ones. The initial nested loop can be rewritten in a URE form using the derived parametric DVs. Due to the localization of the propagation space, the broadcast variables should be initialized in a set of points of the index space specified by the corresponding parameters $d_i$, $i = 1, 2, \ldots, N$. In case of fan-in variables, localization leads to the generation of partial results, which should be concentrated using the $\odot$ operator in order to obtain the correct results from the transformed algorithm. A more formal and analytical description of the above methodology can be found elsewhere [22].

This localization technique differs from the approach presented in chapter 6 in that a single choice is made before the space-time mapping (see section 3) is considered. As indicated in chapter 6, the pros and cons of the different approaches depend on the application domain.

## 3 Application to 1-D convolution

As mentioned, the proposed methodology will be illustrated by the systematic derivation of a variety of regular processor arrays implementing 1-D convolution. The initial description of the algorithm in nested loop form [8] can be represented as follows:

```
for $i = 0$ to $2n - 2$
    for $j = 1$ to $i$
        $y(i) = y(i) + u(j)w(i - j);$
    next $j$
next $i$
```

Since the statement in the loop body is of the form described by equation (3), fan-in occurs for the variable $y(i)$. The direction of the fan-in is along the $j$-axis since equation (1) results in $i = const.$ The other two variables $u(j)$ and $w(i - j)$ satisfy the conditions for broadcasting. The former is broadcast along the $i$-direction since equation (1) results in $j = const.$ The $w(i - j)$ variable is broadcast along the $(i - j)$-direction since equation (1) results in $i - j = const.$

For the determination of the parametric DV for $y(i)$, consider any two nodes $\mathbf{i}_1 = (i_1, j_1)$ and $\mathbf{i}_2 = (i_2, j_2)$ on its propagation space. From equation (4) we obtain:

$$i_2 = i_1 \tag{7}$$

Since $D_s = 1$, it can be derived from equation (5) that one complementary equation of the form

$$j_2 = j_1 + d_{1j} \tag{8}$$

should be used. Therefore, the associated DV is $\mathbf{d}_1 = (0, d_{1j})$. Similarly, the DVs for the variables $u(j)$ and $w(i-j)$ are found to be $\mathbf{d}_2 = (d_{2i}, 0)$ and $\mathbf{d}_3 = (d_{3i}, d_{3i})$, respectively.

Using the derived parametric DVs, the original code of the algorithm is transformed to the following parametric URE:

```
for $i = 0$ to $2n - 2$
    for $j = 1$ to $i$
        $u(i,j) = u(i - d_{2i}, j);$
        $w(i,j) = w(i - d_{3i}, j - d_{3i});$
        $y(i,j) = y(i, j - d_{1j}) + u(i,j)w(i,j);$
    next $j$
next $i$
```

The initialization points for variables $u(i)$ and $w(i-j)$ are specified by the parameters $d_{2i}$ and $d_{3i}$, respectively. The final correct results are obtained by adding the partial results located in the last $d_{1j}$ nodes of each column.

## 3 WORD-LEVEL ARRAY DESIGN

The space-time mapping of the parametric URE derived in the previous step is accomplished by decomposing the index space to independent subsets of variable instances. The number of these subsets depends on the DVs and can be modified by alternative selection of the URE parameters. This results in a variety of array architectures in terms of size, PE utilization, and interconnection patterns. This method is complementary to the approaches introduced in chapters 3, 4, and 6, as indicated there. However, many of the proposed techniques can be combined.

### 1 Mapping of UREs

In most existing regular processor array synthesis methodologies [8, 12, 19], the index space of a recurrence is viewed as an entity, while linear allocation and timing functions are used to determine the parallel execution of the algorithm. A different approach to the mapping problem is presented here, based on the existence of independent subsets of variables in the index space of many applications [9, 3, 15, 21]. This separation leads to processor arrays with higher PE utilization rate and simpler interconnection patterns. Also, mapping of an $N$-D algorithm on an $N$-D processor array can be performed, in contrast to the aforementioned mapping methods. Recurrences of dimension $N$ having $m_d = N$ linearly independent DVs are examined first. Then, the results are extended to cover the case of linearly dependent DVs, as well as the case of $m_d \neq N$ linearly independent DVs. For simplicity, the execution time of the statements is assumed to be unitary.

Any point, $\mathbf{i}$, of the index space can be expressed as an integer-coefficient linear combination of the DVs, plus an integer initial vector $\mathbf{i}_m \in I^N$ [15]:

$$\mathbf{i} = \mathbf{i}_m + \sum_{j=1}^{m_d} a_j \mathbf{d}_j \tag{9}$$

For each $\mathbf{i}_m$, a subset of the index space is defined. Any point of a subset can be used as an initial vector for this subset. Therefore, if $N_G$ is the number of the different subsets of the index space, then only $N_G$ initial vectors are required and thus $m = 1, 2, \ldots, N_G, N_G \geq 1$. In general, the number of independent subsets is equal to the greatest common divisor of the minor determinants of the dependence matrix [25]. Let $T_1 = \{\mathbf{i} \mid t(\mathbf{i}) = 1\}$, where $\mathbf{i}$ is a point of the index space and $t(\mathbf{i})$ is its execution time [5]. If $\mathbf{i}_m \in T_1, m = 1, 2, \ldots, N_G$ then any point of the index space can be written in the form of equation (9), each point of the index space belongs to one subset only, and there is no dependence between different subsets.

The computation of $\{a_j\}$ in equation (9) is equivalent to a change of base in an $N$-D Euclidian space. In general, the DVs which form the new base are neither orthogonal nor unitary. Peir and Cytron [15] use the "minimum distance" method to transform the dependence matrix into an upper triangular matrix. This transformation reduces the required calculations for the computation of $\{a_j\}$. Alternative techniques for the determination of the independent subsets have also been suggested [3, 21]. These overcome the problems of the "minimum distance" when the DVs are linearly dependent, and produce a set of initial vectors for the labeling of the independent subsets. Still, without affecting the algorithm, the new coordinate system can be considered orthogonal and the vectors of its base unitary, since the sequence of the operations is not affected by such a normalization.

The mapping of the normalized subsets is accomplished by the determination of a timing and an allocation function. The timing function is obtained by solving an integer programming problem for each subset. The problem constraints are

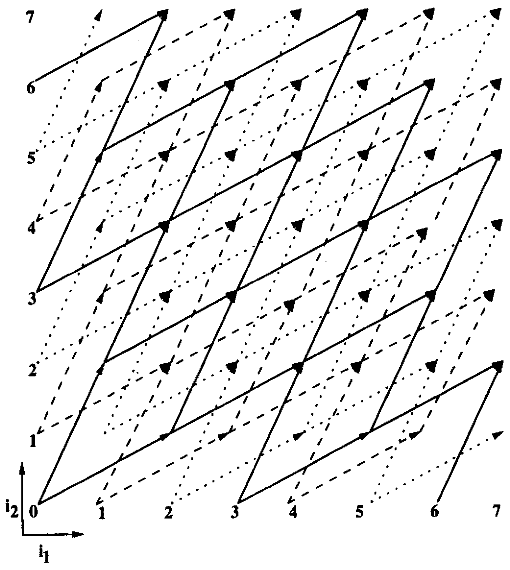


**Figure 1** The three independent subsets within the index space.

similar to those used by Moldovan and Fortes [12]. Due to the normalization of the graph of the subsets, the above problem provides the hyperplane normal to the direction $(1,1,\ldots,1)$ as the optimum solution, i.e. the execution time of each point is:

$$t(\mathbf{i}) = \sum_{j=1}^{m_d} a_j - \min\{\sum_{j=1}^{m_d} a_j\} + 1. \tag{10}$$

As an example, consider a recurrence with DVs $\mathbf{d}_1 = (2,1)$ and $\mathbf{d}_2 = (1,2)$. In the index space, there exist three independent subsets labeled $\mathbf{r}_1 = (0,0)$, $\mathbf{r}_2 = (1,0)$, and $\mathbf{r}_3 = (2,0)$, as shown in figure 1. The normalized graphs of the subsets are shown in figure 2 together with the timing function $i'_1 + i'_2 = c$, $c = 1,2,\ldots$

Finally, the subsets are allocated to an $M$-D processor array. If $M < N$, then $k = M - N$ projections should be performed on the graph of each subset. The projections are performed along the axes in order to preserve the nearest-neighbor communications. The resulting $M$-D graph can be mapped one to one onto the PEs of a mesh-connected array.

The execution time of each subset is optimum when $k = 0$ or $k = 1$. This is true since there are no simultaneous operations along the direction of any axis,

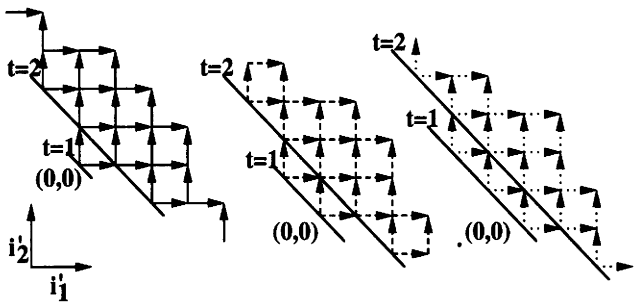


**Figure 2** The three normalized subsets.

so the operations can be executed successively on a single PE. Consequently, assuming that there is no overlap in the execution of different subgraphs, the upper bound for the execution time (for $k \geq 2$) is $t_{up} = Qt_s$, where $t_s$ is the critical-path time of the $(M+1)$-D subgraph, and $Q$ is the number of the $(M+1)$-D subgraphs in which the $N$-D graph of the subset can be decomposed. For example, a cubic graph having an edge of length $n$ and $n^3$ nodes can be decomposed in to $n$ planar subgraphs, each containing $n^2$ nodes.

When the subsets can be executed on the array without simultaneous operations being allocated to one PE, the execution time of the algorithm is optimum if $k \leq 1$, or near-to-optimum if $k \geq 2$. However, a large number of PEs are required. Assuming that all subsets are executed in pipeline fashion by the processor array, the upper bound of the total execution time of the algorithm is $t_{tot} = t_{up} + P(N_G - 1)$, where $P$ is the maximum number of points of a subset mapped on a PE. The time $t_{tot}$ provides the upper bound for the execution time; this is because the possible overlap of the execution of the subsets cannot be taken into consideration, since the exact topology of each graph is not yet known.

A measure of the efficiency of the mapping method and the resulting processor array is the PE utilization, which is defined as:

$$\eta = \frac{N_{OP}}{N_{PE}T} \tag{11}$$

$N_{OP}$ is the number of operations, $N_{PE}$ is the number of PEs of the array processor, and $T$ is the execution time of the algorithm. Assuming that the graph of the subsets and the processor array are hypercubes with an edge of length $n$, the PE utilization is an increasing function in terms of $N_G$ and a

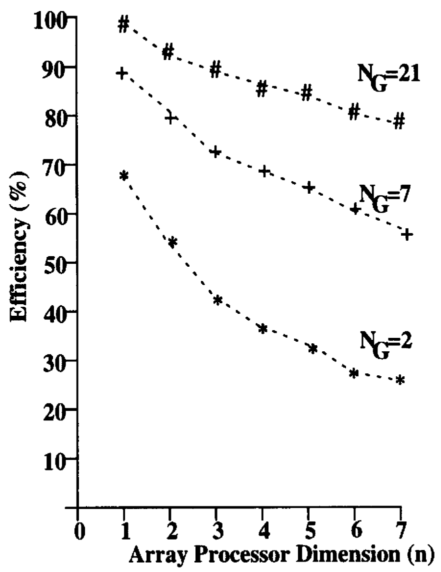


**Figure 3** PE utilization of a processor array.

decreasing one in terms of $n$. Figure 3 depicts the efficiency as a function of $M$ for different values of $N_G$. The efficiency decreases slightly for large values of $n$, e.g., for $n = 100$ it recedes about 1%.

So far, it has been assumed that $m_d = N$ and that the DVs are linearly independent. When $m_d < N$ and the DVs are linearly independent, the graph of any subset lies on a $m_d$-D subspace of the index space. In this case, our methodology can be applied without modification. Indeed, due to normalization, the derived graphs exhibit identical structure to the ones derived for the case $m_d = N$. In the general case, the DVs may be linearly dependent. Let $m_i$ ($m_i \leq N$) be the number of the linearly independent vectors and $\{\beta_{jl}\}$ the coefficients of the linear combination. In order to determine the independent subsets, a new base of linearly independent vectors $\mathbf{E}_j$ should be specified, so that the total number of the subsets is preserved and all DVs are written as an integer linear combination of the new base. Also, some additional constraints are imposed for the determination of a valid timing function. More specifically, the hyperplane normal to the direction $(1, 1, \ldots, 1)$ is not always a permissible timing function, depending on the values of $\{\beta_{jl}\}$. In this case, the relation given by Karp et al. [5] should be used to specify the timing function.

Also, non-nearest neighbor communication may be required in the processor array, depending on the values of $\{\beta_{jl}\}$ and the projection directions. In order to eliminate long-distance data movement, the projections should be performed along the directions characterized by the larger values of $\{\beta_{jl}\}$. The relations for the upper bound of the execution time will hold for a special-purpose array architecture only, which has all the required non-local links. If this is not the case, the delays for data movement using the existing links should be taken into account as a multiplication factor in the estimation for the total execution time. These techniques have been implemented in tools running on a PC.

## 2 Application to 1-D convolution

The proposed methodology is again illustrated for the 1-D convolution. We start from the URE derived in section 2.3. The DG of the algorithm for $n = 8$ considering that $d_{1j} = d_{2i} = d_{3i} = 1$ is similar to that presented by Kung [8] and has one subset of variables only. The initialization part of the index space contains the nodes of the line $j = 0$ for the variable $u(j)$ and the nodes of the line $i - j = 0$ for the variable $w(i - j)$. The results are obtained in the last node of each column. By using the timing function $t = i + j$, the total execution time is found to be 22 time units. Employing projection along the $i$-direction, the DG is mapped into a linear array comprising eight locally interconnected PEs with a PE utilization of 36%.

Assuming $d_{1j} = d_{3i} = 2$ and $d_{2i} = 1$, the derived DG is shown in figure 4. The initialization part for the variable $w(i, j)$ includes the first two rows, and the results are obtained by adding the partial results located in the last two nodes of each column, as is shown by the dashed line in figure 4. In this DG, two independent subsets of variables can be defined for $\mathbf{r}_1 = (0, 0)$ and $\mathbf{r}_2 = (1, 1)$. The nodes of the two subsets are shown by hatched and white circles, respectively. Using $\mathbf{d}_1$ and $\mathbf{d}_2$ as unitary vectors, after normalization the new DVs are $\mathbf{d}'_1 = (0, 1), \mathbf{d}'_2 = (1, 0)$, and $\mathbf{d}'_3 = (2, 1)$. The line $i' + j' = c$ represents a valid timing and each subset can be executed in 17 time units.

With a projection along the $i'$ axis, each subset can be executed by four linearly interconnected PEs. The two subsets can also be executed on the same hardware in pipeline fashion. In this case, the execution of the second subset should start eight time units after the first one. Consequently, the total execution time of the convolution algorithm increases to 25 units, while the PE utilization becomes 64%. By utilizing this allocation scheme, the required accumulation of the partial results of $y(i, j)$ is performed inside each PE, and thus there is no need for extra hardware and time. The resulting linear array is

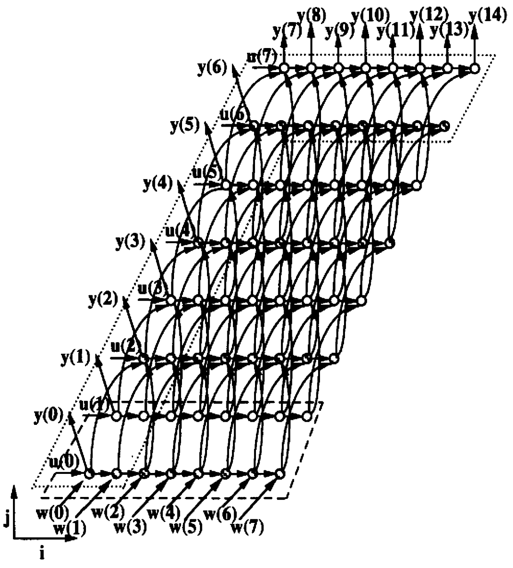


**Figure 4** The convolution DG for $d_{1j} = d_{3i} = 2$ and $d_{2i} = 1$.

shown in figure 5. Each PE is labeled with the time slots in which it is active for the execution of each subset. Two links are required for the data movement caused by $\mathbf{d}'_1$ and $\mathbf{d}'_3$, while the movement described by $\mathbf{d}'_2$ is realized within any PE.

An alternative architecture can be derived if projection along the $j'$-axis is performed. In this case, 14 PEs are required, and non-local links should be employed in order to achieve the minimum execution time. Moreover, the methodology offers the flexibility to produce a variety of alternative arrays, depending on the specifications, by selecting DVs that lead to different partitions of the index space.

## 4 BIT-LEVEL ARRAY DESIGN

As argued in section 1, the throughput of word-level arrays is in several cases not sufficient. Therefore, the bit-level parallelism should also be exploited. Assume that a PE of an array processor has $W$ words as inputs, each denoted $B_\ell, \ell = 0, 1, \ldots, W-1$, and $b_{\ell,v_\ell}$ are defined by:

$$B_\ell = \sum_{v_\ell=0}^{n_\ell-1} b_{\ell,v_\ell} 2^{v_\ell} \tag{12}$$

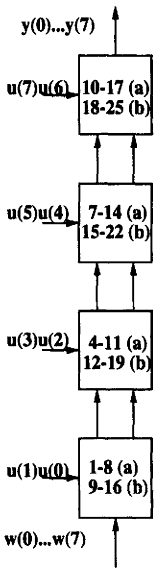


**Figure 5** The linear processor array for the 1-D convolution algorithm.

where $n_\ell$ is the word-length of the $\ell$th input $B_\ell$ and $b_{\ell,v_\ell} \in \{0,1\}$. Let $G$ be an output of the PE, which can be expressed in binary form as:

$$G = \sum_{i=0}^{n_{out}-1} g_i 2^i \tag{13}$$

with $n_{out} = \lfloor \log_2(\max\{G\}) \rfloor + 1$. The operator *max* denotes the maximum value that can be assumed by its operand for all possible input combinations. We focus on bit-level algorithms that can be expressed as nested sums:

$$G = \sum_{k=0}^{K-1} \{ \sum_{v_0=0}^{n_0-1} \cdots \sum_{v_{W-1}=0}^{n_{W-1}-1} z_k(b_{0,v_0}, \dots, b_{W-1,v_{W-1}}) h_k(2^{v_0}, \dots, 2^{v_{W-1}}) \} \tag{14}$$

where $z_k(\cdot)$ is a 1-bit valued function of the input bits, $h_k(\cdot)$ specifies the digital position to which any input bit is assigned, and $K$ is an integer that denotes the number of elementary sums that make up the algorithm.

Assume that $\mathbf{v} = (v_0, \dots, v_{W-1}) \in V = \{0,1,\dots,n_0-1\} \times \{0,1,\dots,n_1-1\} \times \cdots \times \{0,1,\dots,n_{W-1}-1\}$, where $V$ is an $W$-D integer space. Let

$$h_k(2^{v_0}, 2^{v_1}, \dots, 2^{v_{W-1}}) = \sum_{u=0}^{p-1} q_{k,u,\mathbf{v}} 2^u \tag{15}$$

where $p = \lfloor \log_2(\max_k\{h_k(2^{v_0}, 2^{v_1}, \ldots, 2^{v_{W-1}})\})\rfloor + 1$ is the word-length of $h_k(\cdot)$, and $q_{k,u,\mathbf{v}} \in \{0, 1\}$.

By substituting equations (13) and (15) into equation (14), the following is obtained:

$$\sum_{i=0}^{n-1} g_i 2^i = \sum_{u=0}^{p-1}\{\sum_{k=0}^{K-1}\sum_{v_0=0}^{n_0-1}\cdots\sum_{v_{W-1}=0}^{n_{W-1}-1} q_{k,u,\mathbf{v}} z_k(b_{0,v_0}, \ldots, b_{W-1,v_{W-1}})\}2^u \tag{16}$$

Due to the carries produced by the nested summations of equation (16), it holds that $p \leq n$. As can be seen from equation (16), $z_k(b_{0,v_0}, b_{1,v_1}, \ldots, b_{N_k-1,v_{N_k-1}})$ contributes to $g_i, i = 0, 1, \ldots, p-1$ if and only if $q_{k,i,\mathbf{v}} = 1$. Therefore, the number $\alpha_i$ of input bits that contribute to $g_i$ is specified by:

$$\alpha_i = \begin{cases} \sum_{k=0}^{K-1}\sum_{v_0=0}^{n_0-1}\sum_{v_1=0}^{n_1-1}\cdots\sum_{v_{W-1}=0}^{n_{W-1}-1} q_{k,i,\mathbf{v}} & 0 \leq i \leq p-1 \\ 0 & p \leq i \leq n-1 \end{cases} \tag{17}$$

## 1 The derivation of the bit-level DG

By definition, the dimension of the DG of an algorithm is identical to the index space dimension. In particular, the bit-level algorithm of equation (14) should be described by a $W$-D DG. However, the construction of the multi-dimensional DG is avoided by using combinatorial logic to compute the values of $z_k(\cdot)$, thus reducing the problem to multiple-operand binary addition, which can be represented by a 2-D DG. Moreover, in this DG, the properties of the target architecture may be embodied.

Among the variety of existing architecture styles available for the target architecture (see chapter 1) for the proposed bit-level processors, the regular array style [8, 7] is chosen, using the full adder (FA) as basic building block. The chosen style meets the limitations imposed by the VLSI technology and provides solutions with suboptimal latency, but at a high throughput rate. Assume that an FA and a link of the target architecture are mapped one-to-one to a node and an edge of a graph, which is described by the following UREs:

$$\begin{cases} s(i,j) &= s(i,j+1) \oplus x(i,j) \oplus c(i-1,j-1) \\ c(i,j) &= maj\{s(i,j+1), x(i,j), c(i-1,j-1)\} \\ x(i,j) &= x(i-1,j) \end{cases} \tag{18}$$

where the bits $s(\cdot)$, $x(\cdot)$, and $c(\cdot)$ correspond to the augend, the addend, and the carry bit, respectively. Also, $\oplus$ is the *XOR* operator and *maj* is the majority function. Equation (18) describes the binary addition.

The DG will be completely specified when the exact position of the nodes and the interconnecting edges are determined. The number of the nodes of the DG should be specified first. Let $A_i$ be the number of nodes contributing to the computation of the $i$th output bit, $g_i$. Also, let $\beta_i$ be the number of bits that should be added by the $A_i$ nodes. The $\beta_i$ bits include the $\alpha_i$ bits specified by equation (17), and the carry bits that result from the computation of the $(i-1)$th output bit. It can be noticed that the number of carry bits coincides with the number $A_{i-1}$. Hence:

$$\beta_i = \alpha_i + A_{i-1} \tag{19}$$

where $i = 0, 1, \ldots, n-1$ and $A_{-1} = 0$. The minimum number of nodes for obtaining the $i$th output bit is:

$$A_i = \left\lceil \frac{\beta_i - 1}{2} \right\rceil, \beta_i \geq 1 \tag{20}$$

By calculating $A_i$ and taking into account that the DV associated to the variable $x(\cdot)$ is $(0, -1)$, the structure of the DG is entirely described by determining for each $i$ the smallest value of $j$ at which a node is placed. This is accomplished by a series of lemmas, which have been formally proven [22]. In particular, the points of the index space where a node should be placed and the starting and the ending nodes of local and (possibly) non-local links, assuming a minimum-graph-path requirement [14], are specified. Therefore, the derived DG is not completely homogeneous, i.e., dependencies that are present only in certain of its portions may exist. Consequently, more than one set of UREs are required to describe the DG, one set per portion. Indeed, it has been proven [22] that the DG of any bit-level algorithm given by equation (14) may be described by seven sets of UREs for the specific target architecture.

## 2 Hardware realization

The last step of a graph-based methodology is the mapping of the derived DG onto the bit-level processor. More specifically, assuming that any node of the DG is mapped onto a PE of an FA-based array architecture, a high-throughput 2-D array is derived. Also, the DG can be projected, resulting in a variety of linear array architectures [8]. Many alternative architectures can be derived by taking into account various design specifications, such as hardware complexity, latency, and throughput. The architectures of the FA-based arrays could be systolic or wavefront arrays, with horizontal busses and/or local links. Since additions are associative and commutative, the order in which they are performed does not affect the final result. This leads to freedom in loading the input bits and a potential minimization of the bus width [22, 13].

Alternative FA-based array architectures can be derived using the concept of independent subsets of section 3.1, and a small number of additional FAs. The major feature of these architectures is that a significant reduction in the latency of the FA array can be achieved. For example, the following system of UREs:

$$\left\{\begin{array}{lcl} s(i,j) & = & s(i,j+2)\oplus x(i,j)\oplus c(i-1,j-1) \\ c(i,j) & = & \mathrm{maj}\,\{s(i,j+2),x(i,j),c(i-1,j-1)\} \\ x(i,j) & = & x(i-2,j) \end{array}\right. \tag{21}$$

also describes binary addition and results in two independent arrays. These two arrays may function concurrently and their outputs are added by additional diagonally located FAs.

## 3 Application to 1-D convolution

Continuing from the word-level architectures produced in section 3.2, we describe the design of the PE that performs a multiply-accumulate (MAC) operation. This type of operations may be computed by an inner product step processor (IPSP). The systematic design of an IPSP, operating in residue number system (RNS) arithmetic, will be described. Some RNS basics are discussed below. In RNS arithmetic [23], any integer $x$ is represented by an $L$-tuple $(X_1, X_2, \ldots, X_L)$, where $X_i = \langle x\rangle_{m_i}$, where $m_i$ is an element of the base $\{m_1, m_2, \ldots, m_L\}$ which contains relatively prime integers. The notation $\langle f\rangle_m$ denotes the operation $f$ modulo $m$. The representation is unique for $x = 0, 1, \ldots, x_{max}$, where $x_{max} = \prod_{i=1}^{L} m_i - 1$. Any computation requires $L$ parallel channels, performing the same operation, but each in a finite integer ring $R(m_i)$, $i = 1, 2, \ldots, L$. When the IPSP is adapted for arithmetic in $R(m)$, it is called IPSP$_m$. The input/output relation of the IPSP$_m$ is described by:

$$G_{out} = \langle B_0 + \langle B_1 \times B_2\rangle_m\rangle_m \tag{22}$$

Matching equation (22) to the convolution algorithm, $B_0$, $B_1$, $B_2$, and $G_{out}$ should be defined as $B_0 = \langle y(i, j - d_{1j})\rangle_m$, $B_1 = \langle u(i,j)\rangle_m$, $B_2 = \langle w(i,j)\rangle_m$, and $G_{out} = \langle y(i,j)\rangle_m$. The word length of all the inputs and the output is $n = \lfloor \log_2 m\rfloor + 1$. Hence, equation (22) becomes:

$$G_{out} = \langle \sum_{v_0=0}^{n-1} b_{0,v_0} 2^{v_0} + \sum_{v_0=0}^{n-1}\sum_{v_1=0}^{n-1} b_{1,v_0} b_{2,v_1} \langle 2^{v_0+v_1}\rangle_m \rangle_m \tag{23}$$

Due to the outer modulo operation involved, equation (23) does not belong to the class of bit-level algorithms described by equation (14). However, equation (23) can be decomposed into two parts that can be expressed in the form of equation (14) and one addition, leading to an architecture of three stages. The first stage computes the sum $G_0$:

$$G_o = \sum_{v_0=0}^{n-1} b_{0,v_0} 2^{v_0} + \sum_{v_0=0}^{n-1} \sum_{v_1=0}^{n-1} b_{1,v_0} b_{2,v_1} \langle 2^{v_0+v_1} \rangle_m \tag{24}$$

Generally, the word length of $G_0$ may be greater than $n$. The second stage transforms $G_0$ to $G_r$, which has a word length of $n$ bits and $\langle G_0 \rangle_m = \langle G_r \rangle_m$. This transformation is achieved using the following recursive formula:

$$G_k = \sum_{i=0}^{n_k-1} g_{k,i} 2^i = \sum_{i=0}^{n_{k-1}-1} g_{k-1,i} \langle 2^i \rangle_m \tag{25}$$

where $1 \leq k \leq r$ and $n_k$ is the word length of $G_k$. An algorithm exists to estimate the number of necessary recursions $r$ [22, 13]. However, $G_r$ may be greater than $m$. Therefore, the output of the second stage should be mapped during the third stage to its modulo $m$ value. The function of the third stage is described by the following equation:

$$G_{out} = \begin{cases} G_r, & G_r < m \\ G_r - m, & G_r \geq m \end{cases} \tag{26}$$

It may be implemented by one $n$-bit adder and simple control logic.

Applying the methodology of section 4.1, the DG structure of the first stage as well as that of each recursion of the second stage can be specified. Then, each DG is mapped onto an FA-based array architecture.

The complete architecture of an IPSP$_m$ consists of $n^2$ AND gates [4], the input interface, and the FA-based array, as it is depicted in figure 6. The outputs of the AND gates are driven to the input interface, which loads the $z_0$ and $z_1$ bits to the proper array row with the correct sequence and at the exact time instant. These are specified by the adopted allocation and timing schemes. The input interface may consist of a series of parallel-to-parallel or parallel-to-serial shift registers, depending on whether a bit-parallel or bit-serial loading scheme is adopted. The core of the cell architecture is the FA-based array derived above. Since an IPSP$_m$ architecture includes more than one FA-based array, unit delay cells should be inserted to preserve high pipelinability between successive arrays [22]. The design of the FA-based array of IPSP$_{29}$ is offered

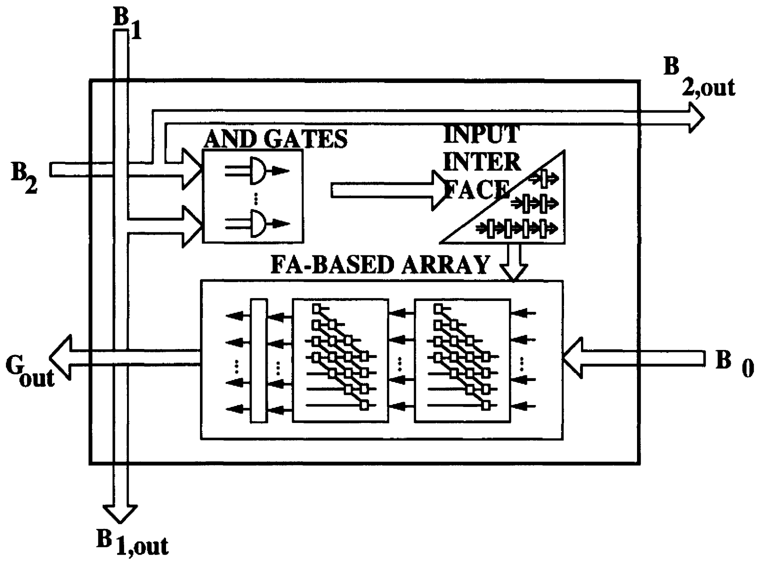


**Figure 6** The general structure of the IPSP.

as an example in figure 7. The architecture of $\text{IPSP}_{29}$ results after one-to-one mapping of each DG onto the FA-based array. The timing function $j = const$ and the bit-parallel loading scheme are employed for obtaining $G_{out}$.

## 5 CONCLUSION

In this chapter, a methodology for designing processor arrays, starting from the algorithmic level and ending with the bit-level architecture of each PE of the array, was described. The proposed methodology exploits two levels of parallelism: the word level, which is inherent to the algorithm, and the bit level, which exists in the execution of each word level operation. Given an algorithm in nested-loop form, the propagation space of each variable is determined, and an equivalent parametric URE form is obtained. The URE is mapped to the desired processor array after partitioning the index space to independent subsets of variable instances. Since the partitions depend on the DVs, trade-offs with respect to array size, interconnection pattern, efficiency, and total execution time of the algorithm can be performed by alternative selections of the URE parameters. The derivation of the topology of the processor array is followed by the detailed design of each PE, using a graph-based approach. The resulting PE is an FA-based array and includes an I/O interface. The method provides the designer with the capability of handling critical factors of a PE, such as

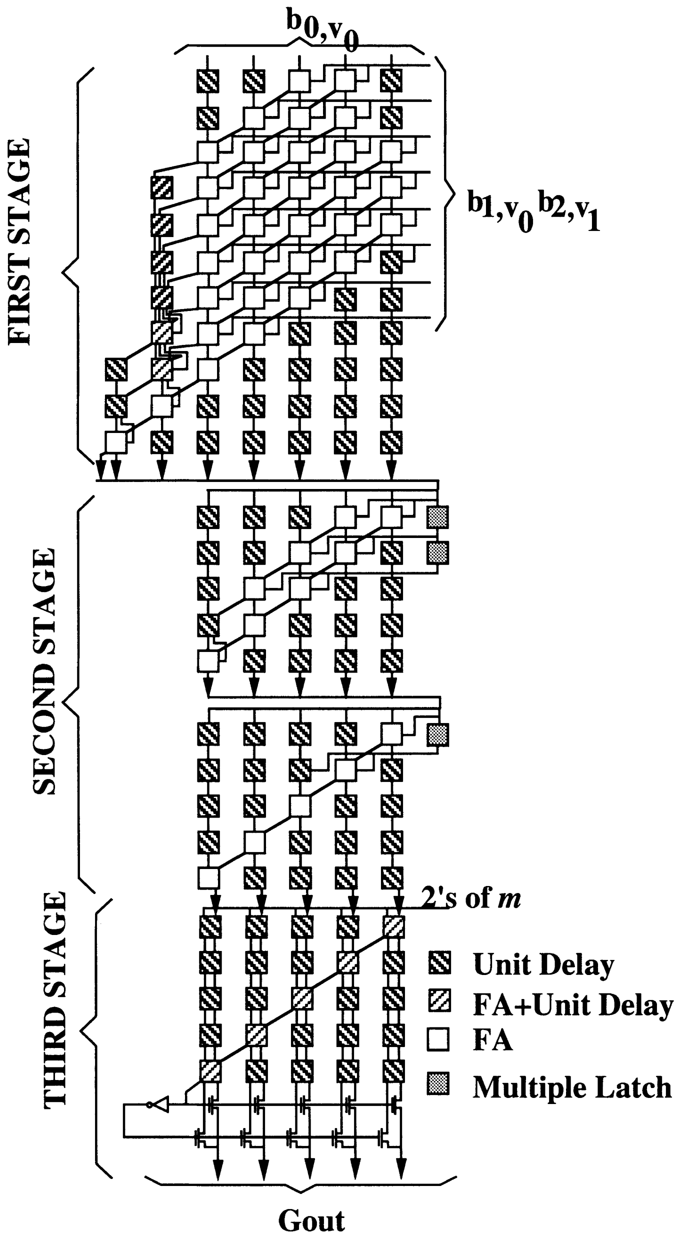


**Figure 7** The FA based architecture of the IPSP$_{29}$.

area, latency, and throughput, as well as various loading, allocation, and timing schemes.

The synthesis methodology described here is amenable to software implementation. Indeed, a synthesis tool incorporating the results presented in this chapter is already under development.

## REFERENCES

[1] M. Birbas, D. Soudris, and C. Goutis. Design methodology for mapping iterative algorithms on array architectures. *Proc. IEEE Int. Symp. on Circuits and Systems*, Singapore, pages 3058–3061, 1991.

[2] J. Bu. *Systematic design of regular VLSI processor arrays.* PhD thesis, Delft Univ. of Technology, May 1990.

[3] E. D'Hollander. Partitioning and labeling of index sets in do loops with constant dependence vectors. *Proc. IEEE Int. Conf. on Parallel Processing*, Vol. II, pages 139–144, 1989.

[4] K. Hwang. *Computer arithmetic: principles, architecture, and design.* John Wiley & Sons Inc., New York, 1979.

[5] R. Karp, R. Miller, and S. Winograd. The organization for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14, pages 563–590, 1967.

[6] H. T. Kung and M. Lam. Wafer-scale integration and two-level pipelined implementations of systolic arrays. *Journal of Parallel and Distributed Computing*, pages 32–63, 1984.

[7] H. T. Kung and C. Leiserson. Systolic arrays for VLSI. *SIAM Sparse Matrix Proceedings*, pages 245–282, Nov 1978.

[8] S. Y. Kung. *VLSI Array Processors.* Prentice-Hall, New Jersey, 1988.

[9] E. Kyriakis-Bitzaros and C. Goutis. An efficient decomposition technique for mapping nested loops with constant dependencies onto regular array processors. *Journal Parallel and Distributed Computing*, 16, pages 258–264, 1992.

[10] L. Lamport. The parallel execution of do loops. *Com. of ACM*, pages 83–93, Feb 1974.

[11] J. McCanny, J. McWhirter, and S. Kung. The use of data dependence graphs in the design of bit-level systolic arrays. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, 38, pages 787–793, May 1990.

[12] D. Moldovan and J. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Trans. on Computers*, C-35, pages 1–12, 1986.

[13] V. Paliouras, D. Soudris, and T. Stouraitis. Systematic derivation of the processing element of a systolic array based on residue number system. *Proc. IEEE Int. Symp. on Circuits and Systems*, San Diego, CA, 1992.

[14] C. Papadimitriou and K. Steiglitz. *Combinatorial optimization, algorithms and complexity.* Prentice Hall, New Jersey, 1982.

[15] J. Peir and R. Cytron. Minimum distance: a method for partitioning recurrences for multiprocessors. *IEEE Trans. on Computers*, C-38, number 8, pages 1203–1211, 1989.

[16] C. Polychronopoulos. *Parallel programming and compilers.* Kluwer Academic Publishers, Boston, 1988.

[17] P. Quinton and V. Van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1, pages 95–113, Kluwer, Boston, 1989.

[18] S. Rajopadhye. Synthesizing systolic arrays with control signals from recurrence equations. *Distributed Computing*, 3, pages 88–105, 1989.

[19] S. Rao and T. Kailath. Regular iterative algorithms and their implementation on processor arrays. *Proc. of IEEE*, 76, number 3, pages 259–269, 1988.

[20] V. Roychowdhury, S. Rao, L. Thiele, and T. Kailath. On the localization of algorithms for VLSI processor arrays. In R. Brodersen, H. Moscovitz, editors, *VLSI Signal Processing III*, pages 459–470, IEEE Press, 1988.

[21] W. Shang and J. Fortes. Independent partitioning of algorithms with uniform dependencies. *Proc. of Int. Conf. on Parallel Processing*, Vol. II, pages 26–33, 1988.

[22] D. Soudris and C. Goutis. Mapping nested loops with if statements. *ESPRIT 3281 technical report, PU/M30/C2/4*, L. Svensson, editor, IMEC, Belgium, Feb 1992.

[23] F. Taylor. Residue arithmetic: a tutorial with examples. *IEEE Computer Magazine*, pages 40–62, May 1984.

[24] L. Thiele. On hierarchical design of VLSI processor arrays. *Proc. IEEE Int. Symp. on Circuits and Systems*, pages 2517–2520, 1988.

[25] V. Van Dongen. Quasi-regular arrays: definition and design methodology. *Proc. IEEE Int. Conf. on Systolic Arrays*, 1989.

6

# REGULAR ARRAY SYNTHESIS FOR IMAGE AND VIDEO APPLICATIONS

**Jan Rosseel[1], Michaël van Swaaij[1]**
**Francky Catthoor[1], Hugo De Man[1]**
**Hervé Le Verge[2], Patrice Quinton[2]**

[1]*IMEC, Leuven*
[2]*IRISA-CNRS, Rennes*

## ABSTRACT

This chapter presents some results obtained at IMEC and IRISA in the field of regular array synthesis for real-time image and video applications. A fully tuned design methodology is presented that leads to an efficient array architecture for our target domain, starting from a true behavioral description. The power of this methodology is demonstrated on a complex real-life application: a full video motion estimation design. The necessary array synthesis techniques are also introduced, with emphasis on the non-conventional ones.

## 1 INTRODUCTION

In this chapter, we will investigate the design of regular array processor architectures (RAAs), starting from a high-level behavioral description of an application and based on space-time transformation methods. The target application domain is that of *high throughput,* real-time signal and data processing applications, such as front-end image and video processing. This includes algorithms exhibiting a *regular data flow:* sets of nested loops enclosing a body with a limited number of local conditionals. It has to be stressed that synthesis of an architecture for a specific real-time signal processing system means that a *specific instance* of an algorithm is implemented, with a *given throughput and I/O scheme* as constraints and *minimal area* as a goal. The main performance indicator is, therefore, not the input-output delay (latency) but the throughput to be achieved. The throughput is calculated via the *block pipelining period* (BPP) [4]. The target architecture style applied is the regular array style [24].

Many design methods have been proposed to synthesize RAAs. The majority of these methods (see chapter 1 for an overview) are based on an affine transformation (first described by Quinton [15] and Moldovan [12]) to map the index space of the application description to time and processor space. The use of such a transformation method simplifies the design task considerably and requires only a few parameters to characterize a design completely. Unfortunately, most of these methods start from a relatively low level specification, using sets of UREs (uniform recurrence equations) [3] or CUREs (conditional uniform recurrence equations) [17] to describe an application. Moreover, especially in the case of real-time signal processing applications, the resulting architecture is usually unnecessarily fast or too slow.

Therefore, we propose a novel design script to design RAAs for real-time applications. This script is described in section 2. The different steps of the design script are detailed in sections 4 through 7. The methods of the different steps are illustrated by applying them on a real-life application which is described in section 3.

## 2 A DESIGN SCRIPT

A suitable design trajectory for regular array synthesis of real-time applications is given in figure 1. The final result must be an optimized architecture with a throughput matching the system specification. In order to make the task of the designer easier, a higher-level application specification format, such as conditional affine recurrence equations (CAREs) [25] or conditional weak single assignment codes (CWSACs) [21], is introduced above the CURE level. This allows the use of important extensions such as broadcast operations, non-local dependencies, and global operations such as $\sum$ and $\prod$. For this purpose, localization techniques [16] have been introduced. The subsequent space-time transformation, which performs the actual (raw) architecture design, is based on the Quinton-Moldovan space-time mapping, with some extensions [18, 19]. As this mapping generates only architectures with specific, distinct throughput characteristics, depending on the parameters of the design, clustering techniques [13, 1] (see also chapter 4) can be used to adapt the throughput of the architecture.

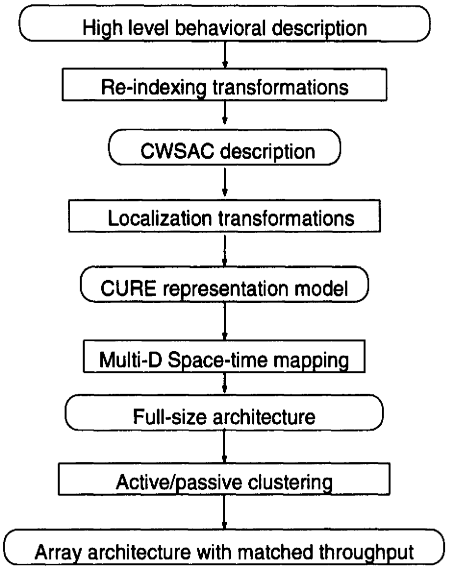

High level behavioral description
Re-indexing transformations
CWSAC description
Localization transformations
CURE representation model
Multi-D Space-time mapping
Full-size architecture
Active/passive clustering
Array architecture with matched throughput

**Figure 1** The proposed regular array synthesis methodology.

This design method, however, still does not start from true behavioral specifications for realistic image and video applications. Therefore, pre-transformations are needed before the space-time mapping and localization steps to transform the partly irregular high level description to regular WSACs by means of re-indexing [24].

Each of these crucial steps will now be illustrated by means of a red-thread demonstrator.

## 3 A REAL-LIFE VIDEO APPLICATION

An approximate but effective algorithm to compress video signals uses the idea of motion estimation (ME) [9]. In this algorithm, the position of a small part of the current frame in the previous image-frame is searched, i.e., one tries to determine the motion of part of the image between consecutive frames. Only the *motion vector* is then transmitted instead of all the pixels. This results in a significant compression of the video signal.

To estimate motion, the current image frame is divided up into small blocks of $m \times n$ pixels for which the "old" location in the previous frame is determined. This search is restricted to an appropriate reference window of $M' \times N'$ pixels

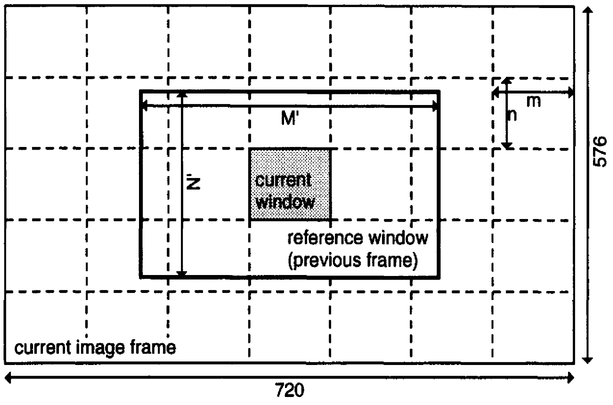


**Figure 2** Notational conventions: definitions for the determination of the motion vector between the blocks in the old and the new images.

```
**For all blocks** $(g, h)$:
    **For all positions** $(i, j)$ **in search frame:**
        $\Delta(g,h,i,j) = \sum_{k,l} |N(8 \cdot g + k, 8 \cdot h + l) - O(8 \cdot g + i + k, 8 \cdot h + j + l)|$;
    **next** $(i, j)$
    $\Delta_{opt}(g,h) = \min_{i,j} \Delta(g,h,i,j)$;
**next** $(g, h)$
```

**Figure 3** Initial application specification.

in the previous frame with the same center as the block which is currently considered (see figure 2). The motion vector is defined as the two-dimensional (2-D) difference vector between the center of the current block and the center of the $m \times n$ block in the old window that best matches the current block. So, theoretically, a compression factor of at most $\frac{m \times n}{2}$ can be reached.

An informal application specification is given in figure 3. In these equations, the following conventions hold: the pixels of the reference window are represented by the variable $O(k', l')$ and those of the current block by the variable $N(k, l)$.

The design of an architecture for this motion estimation application, using the tuned design script of figure 1, is presented in the following sections. It illustrates the individual steps for which a short summary of and a reference to the applied synthesis techniques are supplied.

## 4 DERIVING THE INITIAL DESCRIPTION

The high-level description we expect as input is an extended single assignment model. Many system designers prefer the use of procedural languages (e.g., FORTRAN or C) to describe their application. In order to derive the desired single assignment model from procedural code, techniques developed at the University of Delft (see chapter 4) and at the University of Patras (see chapter 5) can be used.

Alternatively, a single assignment model can be directly expressed using a functional language. The ALPHA language [7] designed at IRISA offers a convenient notation for the description of CAREs. An ALPHA program is a set of equations involving variables defined on integral polyhedral domains. Expressions of the language are built by combining elementary variables by means of five constructors:

- Arithmetic component-wise operators ($X + Y$, $X * Y$, etc.).
- The dependency operator. If $X$ is a variable, then $X.f$ is the functional composition of $X$ and the affine mapping $f$. This operator makes it possible to represent delays or spatial operations on variables.
- The restriction operator. $P : X$ denotes the mathematical restriction of expression $X$ to the (polyhedral) domain $P$.
- The **case** operator. If $X$ and $Y$ are two expressions with disjoint (polyhedral) domains dom$X$ and dom$Y$, then **case**$X$ ;$Y$**esac** denotes the conditional expression defined as $X$ on dom$X$ and $Y$ on dom$Y$.
- The reduction operator. **red**$(+, f, X)$ represents the $\Sigma$ operator applied to expression $X$, where $f$ is an affine mapping describing the range of summation [6].

Based on the denotational semantics of ALPHA [11], axiomatic rules are defined, and semantic-preserving transformations oriented towards the synthesis of systolic arrays are built from these rules. The ALPHA DU CENTAUR program offers an interactive environment for applying these transformations, either for the generation of a parallel program [8] or for the synthesis of a chip [2]. A few of these transformations will be discussed in more depth for the motion estimator below.

## 5 RE-INDEXING TRANSFORMATIONS

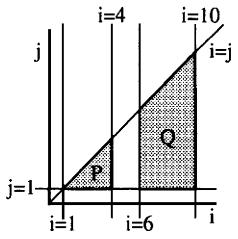
The goal of the re-indexing transformations is to transform the potentially irregular control flow of the original application description into a more regular and uniform one that can be passed to the other tools in the design trajectory. For this purpose, all signals, except those that still have to be localized (see section 6), are placed in a single index space so that the dependencies between signals all lie in a pointed cone in the $n$-dimensional index space. This restriction is necessary to ensure that a valid timing vector can be found during space-time mapping.

These data- and control-flow transformations are based on an extended *polyhedral dependence graph* (PDG) model [23] (see also chapter 7), in terms of which the actual optimization problem is defined and the transformations are performed. In this section, a short overview is given of the main concepts of the PDG. Detailed information can be found elsewhere [23, 22].

Loops and conditions form the *control-flow scope* of single assignment statements. They define regions or domains on the lattice set up by the loop iterators:

```
(i : 1..10) ::
   (j : 1..i) ::
      ... = if (i ≠ 5) ...;
```

*defines domains:*



Each integer point, which corresponds to specific values for $i$ and $j$, in the domains $P$ and $Q$ is associated with a single operation of the assignment statement. Sets of operations of a single statement are in this way captured by polytopes [14]. For example, $P = \{(i,j) \in \mathcal{Z}^2 | i \geq j,\ j \geq 1,\ i \leq 4\}$. The PDG model has two important features:

- It offers *complete, exact and explicit* (not just worst-case) characterization of *all* individual multidimensional dependencies.
- Complexity is *independent* of size parameters in the original description.

One or both features are missing in other models, more oriented towards different synthesis tasks, such as the signal flow graph (SFG) [5] (missing the first feature), and the stream model [10] (missing the first feature, and for non-constant loop boundaries the second one as well). Details on the model and on how to deal with singular index functions and signals with different dimensions can be found in our previous publications [22, 23].

Control and data flow can now be effectively modeled by a *placement* of the node space polytopes of the PDG in a *common node space.* Note that in our model, changes of placement have explicitly measurable effects on optimization criteria such as the minimum required number of storage locations, parallelism, and processor usage. As a result, it has been possible to design automated steering techniques for the re-indexing transformations oriented to the design of real-time application-specific processor arrays [24]. These optimization techniques have been implemented in the R4C4 synthesis tool at IMEC. The impact of this important synthesis task will now be illustrated with the red-thread example.

The behavioral description of the ME application given in figure 3 does not yet take into account the border effects. The search window is indeed smaller for blocks located at the borders of the image frame. The subwindows can thus be divided in nine classes, depending on the possible horizontal or vertical motion of the block in the reference window. A total of nine nested loop constructs with slightly different bounds must be written to represent this. Using the PDG model introduced above and the automated re-indexing transformations, this apparent irregularity can be partly removed. For this purpose, the polytopes corresponding to these nine set of loops can be brought into the same index space, by introducing statements that are conditional in the loop indices, and be placed closely together. The polytope enclosing the nine polytopes can then be transformed into one set of nested loops with control equations governing the border effects. A partial set of equations before and after the transformations is given in figure 4. See also figure 5 for a more intuitive view of the problem.

## 6 LOCALIZING TRANSFORMATIONS

After re-indexing, broadcast operations and global operations such as $\sum$ are still present in the description of figure 4. Localization techniques are needed to localize these operations to arrive at a uniform description with only local and constant dependencies, as required by the space-time mapping. We will

**Before transformation:**

```
/* Upper left corner block*/
$\Delta_{opt}(1,1) = \infty$
(i:0..M)::
    (j:0..N)::
        $\Delta(1,h,i,j) = 0$
        (k:1..m)::
            (l:1..n)::
                $\Delta(1,1,i,j) = \Delta(1,1,i,j) + \ldots$
        $\Delta_{opt}(1,1) = min(\ldots))$
⋮
/* blocks of (large) central area */
(g:2..G-1)::
    $\Delta_{opt}(g,1) = \infty$
    (i:0..M)::
        (j:-N..N)::
            ⋮
    (h:2..H-1)::
        $\Delta_{opt}(g,h) = \infty$
        (i:-M..M)::
            (j:-N..N)::
                $\Delta(1,h,i,j) = 0$
                (k:1..m)::
                    (l:1..n)::
                        $\Delta(g,h,i,j) = \Delta(g,h,i,j) + \ldots$
                $\Delta_{opt}(g,h) = min(\ldots)$
    ⋮
```

**After transformation:**

```
(g:1..G)::
    (h:1..H)::
        $\Delta_{opt}(g,h) = \infty$
        (i:-M..M)::
            (j:-N..N)::
                if $((2 \leq g \leq G-1 \vee (g = 1 \wedge i \geq 0) \vee (g = G \wedge i \leq 0))$
                    $\wedge(2 \leq h \leq H-1 \vee (h = 1 \wedge j \geq 0) \vee (h = H \wedge j \leq 0)))$
                        $\{\Delta(g,h,i,j) = 0\}$
                (k:1..m)::
                    (l:1..n)::
                        ⋮
```

**Figure 4** Sets of loops before and after automated re-indexing.

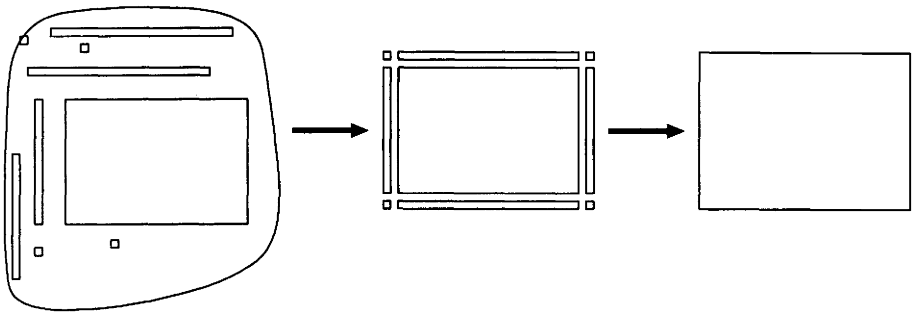


**Figure 5** Re-indexing reduces complexity.

describe the basics of the localization process on the example of a simple global operation taken from the ME application. More detailed literature about the applied localization procedure is available [16].

## 1 Basic localization method

To illustrate the localization principles, a localization of the $\sum$ operator of the ME application will be performed. The core operation in the ME application is the calculation of the $\Delta(g,h,i,j)$ values (see figure 3). In the remainder of this chapter, the ME application will be discussed with the outer loop (**For all blocks** $(g,h)$**:**) stripped off, in order not to overload notations and equations. So, with the removal of the $g$ and $h$ indices, the core operation becomes:

$$\begin{aligned}\Delta(i,j) &= \sum_{k,l} |N(k,l) - O(i+k,j+l)| \\ &\quad \forall (i,j,k,l) \in D = \left\{ (i,j,k,l) \mid 1 \le i,j,k,l \le M,N,m,n \right\} \end{aligned} \quad (1)$$

Consider the following general form of a global operation defined over an index space $D \subset Z^n$:

$$V(J) = \sum_{I \in f^{-1}(J) \cap D} v(I) \quad \forall J \in f(D) \quad (2)$$

where $f$ is an affine function $Z^n \to Z^m$ with $m < n$. To localize the equations, a technique called *null-space propagation* can be used in most cases. A propagation vector $P$ is chosen from the intersection of the null-space of $f$ ($= N_f$) with $D$. Partial results of the global operation will be propagated along this direction to the different index points where they are produced. This allows then to generate the global result by "accumulating" (or combining) the partial results over this propagation direction.

In general, one step of the localization "unrolls" one dimension of the null-space. A global or broadcast operation is thereby replaced by a set of localized operations and a set of new global or broadcast operations defined over a reduced null-space.

The process must be repeated until all domain indices in the null-space of $f$ have been treated, resulting in a completely localized description. In some cases, more elaborate localization techniques, such as routing and domain extensions, will be needed to localize an equation completely [16]. Also, support is needed to guide the choice of appropriate pipelining/routing vectors. This decision can also be automated [22].

Applied to the ME example, equation (1) can be replaced by the following set of operations where two successive localization steps have been performed (for the $\sum$ operation over the $\{k, l\}$ domain):

$$\Delta(i,j) = S_2(i,j,m,n) \tag{3}$$

$$S_2(i,j,k,n) = \begin{cases} S_2(i,j,k-1,n) + S_1(i,j,k,n) & \text{if } k > 1 \\ S_1(i,j,1,n) & \text{if } k = 1 \end{cases} \tag{4}$$

$$S_1(i,j,k,l) = \begin{cases} S_1(i,j,k,l-1) + |N(\ldots) - O(\ldots)| & \text{if } l > 1 \\ |N(\ldots) - O(\ldots)| & \text{if } l = 1 \end{cases} \tag{5}$$

## 2 Localization in Alpha du Centaur

All the localization transformations described above have been integrated in the ALPHA DU CENTAUR environment, developed at IRISA, so that an automated transformation is possible. A complete localization of the ME application, without the outer $(g, h)$ loop, using this interactive environment has been performed. The initial CARE description in ALPHA notation is listed in figure 6.

A number of transformations were applied to transform this description in a set of CUREs. All transformations were performed interactively, which is possible thanks to the adequate response time from the ALPHA DU CENTAUR environment. The following transformations were applied:

- Factorization of the 2-D global operations into two 1-D global operations.
- Localization of the global operations. These first two steps, resulting in completely localized global operations, are fully automated in ALPHA DU CENTAUR.

```
system video_codec( N   :   {k,l|k ≥ 1; l ≥ 1; 8 ≥ l; 8 ≥ k} of integer;
                    O   :   {ip,jp|ip ≥ 1; jp ≥ 1; 23 ≥ jp; 23 ≥ ip}
                                of integer )
returns ( Delta   :   integer   );
let
 Delta    =    red( min , (i,j →) ,
                red( + , (i,j,k,l → i,j) ,
                  {i,j,k,l|j ≥ 1; l ≥ 1; k ≥ 1; i ≥ 1; 8 ≥ k; 8 ≥ l; 16 ≥ i; 16 ≥ j} :
                   | N.(i,j,k,l → k,l) - O.(i,j,k,l → i+k−1, j+l−1) | ) );
tel ;
```

**Figure 6** CARE description of the ME application before localization, using ALPHA notation.

- Factorization of broadcast operations. This must be done manually.
- Localization of the broadcast operations. For this step, the user is asked to select a pipelining vector from a set of possible alternatives. The localization itself is then performed automatically.
- ALPHA DU CENTAUR does not place the intermediate variables that result from localizing global operations in the same index space. This can be done with a re-indexing transformation after the localization step. The localization procedure could also be adapted so that this step is not necessary, as in the case of the localization of broadcast operations.

The end result after applying these transformations is the set of ALPHA equations listed in figure 7. Notice that this set of CUREs is only one of the many that can be generated, based on decisions (such as selection of pipelining directions) taken in each of the transformation steps mentioned above.

## 3 A localization representation model

Localization transformations result in *one* application description suited for space-time mapping. However, the optimality of the architecture may heavily depend on how the localization was performed [20]. It is therefore necessary to couple the localization task with the space-time assignment. Due to complexity issues, we have found it impossible to perform the complete space-time assignment simultaneously with the localization task. A divide-and-conquer strategy that performs the two tasks sequentially, while monitoring the interaction be-

```
system video_codec ( N   :  {k, l|k ≥ 1; l ≥ 1; 8 ≥ l; 8 ≥ k} of integer;
                     O   :  {ip, jp|ip ≥ 1; jp ≥ 1; 23 ≥ jp; 23 ≥ ip}
                                  of integer)
returns (  Delta   :   integer   );
var
 O2   :   {i, j, k, l|1 = l; j ≥ 1; i ≥ 1; k ≥ 1; 16 ≥ j; 8 ≥ k; 16 ≥ i} ,
          {i, j, k, l|l ≥ 2; k ≥ 1; 16 = j; i ≥ 1; 8 ≥ k; 16 ≥ i; 8 ≥ l} of integer;
 . . .
 S2   :   {i, j, k, l|i = 0} , {i, j, k, l|16 ≥ i; i ≥ 1} of integer;
 D1   :   {i, j, k, l|j ≥ 1; l ≥ 1; k ≥ 1; i ≥ 1; 8 ≥ k; 8 ≥ l; 16 ≥ i; 16 ≥ j} ,
          {i, j, k, l|j ≥ 1; i ≥ 1; l = 0; k ≥ 1; 16 ≥ i; 8 ≥ k; 16 ≥ j} of integer ;
let
 Delta   =   S2.(→ 16, 16, 8, 8);
 D1      =   case
               {i, j, k, l|j ≥ 1; i ≥ 1; l = 0; k ≥ 1; 16 ≥ i; 8 ≥ k; 16 ≥ j} :
                 eltn( + ).(i, j, k, l →);
               {i, j, k, l|j ≥ 1; l ≥ 1; k ≥ 1; i ≥ 1; 8 ≥ k; 8 ≥ l; 16 ≥ i; 16 ≥ j} :
                 D1.(i, j, k, l → i, j, k, l − 1) + | N1 - O1 |;
             esac;
 . . .
 S2      =   case
               {i, j, k, l|i = 0} : eltn( min ).(i, j, k, l →);
               {i, j, k, l|16 ≥ i; i ≥ 1} :
                 min ( S2.(i, j, k, l → i − 1, 16, 8, 8) ,
                   S1.(i, j, k, l → i, 16, 8, 8) );
             esac;
 . . .
 O2      =   case
               {i, j, k, l|l ≥ 2; 8 = k; 16 = j; i ≥ 2; 16 ≥ i; 8 ≥ l} ,
               {i, j, k, l|l ≥ 2; k ≥ 1; 16 = j; 1 = i; 8 ≥ k; 8 ≥ l} ,
               {i, j, k, l|1 = l; 8 = k; j ≥ 1; i ≥ 2; 16 ≥ i; 16 ≥ j} ,
               {i, j, k, l|1 = l; j ≥ 1; k ≥ 1; 1 = i; 8 ≥ k; 16 ≥ j} :
                 O.(i, j, k, l → i + k − 1, j + l − 1);
             {i, j, k, l|l ≥ 2; k ≥ 1; 16 = j; i ≥ 2; 7 ≥ k; 16 ≥ i; 8 ≥ l} ,
               {i, j, k, l|1 = l; j ≥ 1; i ≥ 2; k ≥ 1; 16 ≥ j; 7 ≥ k; 16 ≥ i} :
                 O2.(i, j, k, l → i − 1, j, k + 1, l);
             esac;
 tel ;
```

**Figure 7** Part of a description of the ME application, localized using the ALPHA DU CENTAUR environment.

tween the tasks, is therefore preferred. Some designer feedback to come to an optimized solution is then allowed, too.

An initial idea for a solution to the problem is first to try to find a space-time assignment for the index space of the given problem, and then search for the best matching localization [20]. However, the coupling between the space-time assignment and the localization task must be tighter than a simple sequential execution. All possible localization alternatives should be known during the assignment phase, for two reasons:

- For some combinations of space-time assignment parameters, no localization can be found. It is best to detect this situation during the assignment phase instead of in a global feedback loop.
- Knowing all feasible localizations during assignment allows a small and fast optimization feedback loop in the assignment task itself. A tradeoff between the cost of non-matched throughput or a bad localization can then be made for a number of projection vector alternatives.

The approach we have consequently chosen is first to derive all possible localizations. After this, a space-time assignment with the best matched throughput is searched, together with an associated localization that minimizes the total cost of the architecture. The problem with this approach is the large number of localization alternatives. Fortunately, there is a lot of similarity between the localizations. Therefore, we have developed a localization representation model that allows one to represent all localizations for a given application description in a non-redundant way [20]. The model consists of the following parts:

- A graph to represent all routes one can follow to arrive at a feasible localization.
- A set of "dictionaries" that define regions (subspaces of the index space), dependence vectors, and operations.

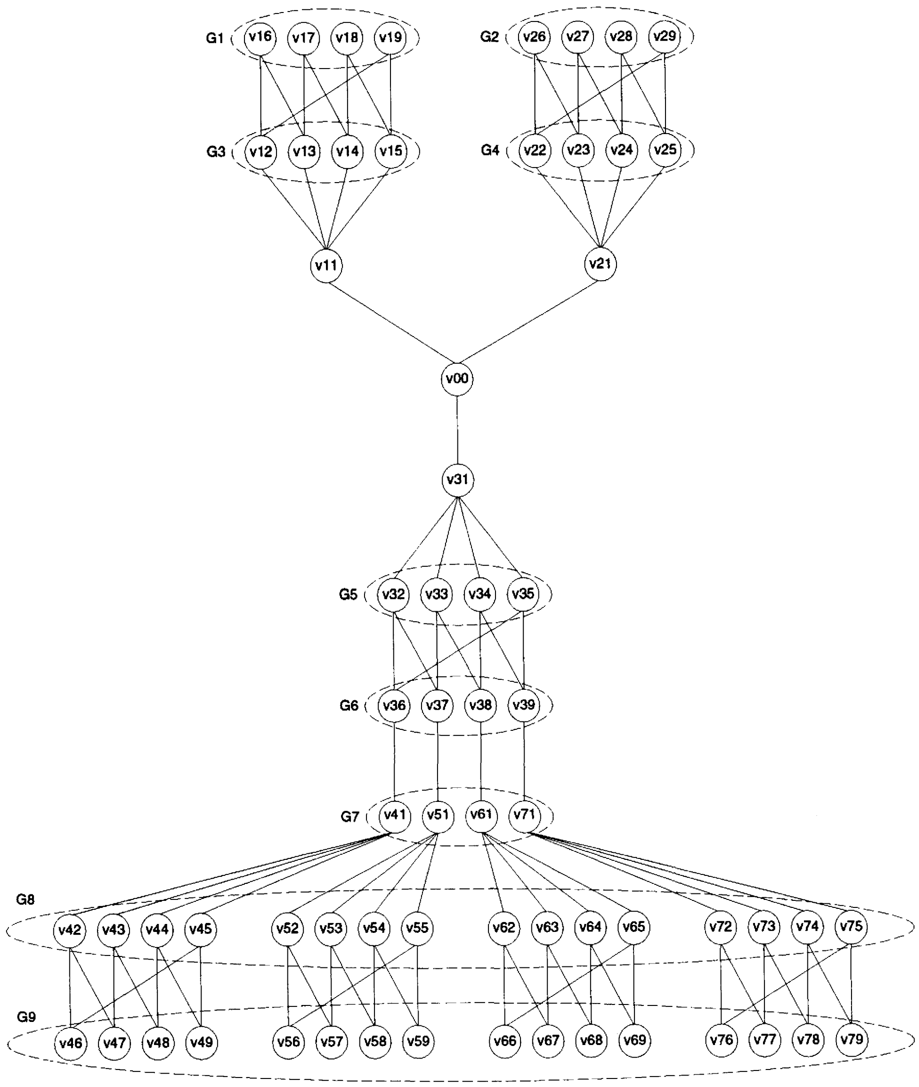


**Figure 8** The undecorated localization graph for the motion estimation application.

| Alternative | # of domains | # of dep. vectors | # of oper. |
|---|---|---|---|
| 4096 localizations | (4096×)19 | (4096×)8 | (4096×)18 |
| 1 representation model | 73 | 12 | 26 |

**Table 1** Comparison of optimization overhead for considering all localization alternatives.

For the ME description contained within the two inner loops of figure 3, there are 4,096 alternative ways to localize the initial high-level description. The number of alternatives is even larger when also considering the outer $(g, h)$ loop. The representation model was used to represent all localization alternatives. The localization graph is illustrated in figure 8. Some patterns can be recognized here:

- The top two groups represent the localization of the broadcast of the global variables representing the current and reference window. The form of these groups is typical for the localization of 2-D global or broadcast operations.
- The middle group represents the localization of the summation operator. Since this is also a 2-D operation, the localization form again looks similar.
- The bottom row of groups all represent the localization of the 2-D minimization operation. This graph must be repeated because the place where the minimization occurs depends on the localization chosen for the $\sum$ operation.

Using this representation model results in a large reduction of the search space if the best combination of space-time mapping and localization must be found. This is indicated in table 1, where the number of domain definitions, dependence vectors and operation definitions that must be considered when iterating over all localization alternatives is compared to what is needed when using the novel representation model. Furthermore, the model can also be used in synthesis environments which do not address array architectures [20].

## 7 SPACE-TIME MAPPING

In general, image and video processing applications lead to many nested loops in the initial description (up to six due to the single assignment model as in the complete ME application). Moreover, the complexity of the processing elements can be very high, potentially including internal pipelining. These real-life requirements have necessitated extensions to the basic space-time mapping method [12, 15].

### 1 Extended space-time mapping

Techniques that are tuned to throughput-based real-time processing have been presented elsewhere [19, 18]. The following extended method is used for the space-time mapping of the indices of the algorithmic index space $D$:

$$\begin{bmatrix} t_1 \\ \vdots \\ t_{n-2} \\ x \\ y \end{bmatrix} = \begin{bmatrix} t \\ p \end{bmatrix} = \tilde{T} \times i + \gamma = \begin{bmatrix} \pi_{11} & \cdots & \pi_{1n} \\ \vdots & & \vdots \\ \pi_{(n-2)1} & \cdots & \pi_{(n-2)n} \\ s_{(n-1)1} & \cdots & s_{(n-1)n} \\ s_{n1} & \cdots & s_{nn} \end{bmatrix} \times \begin{bmatrix} i_1 \\ \vdots \\ i_n \end{bmatrix} + \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \gamma_A \\ 0 \\ 0 \end{bmatrix}$$

The vector $t$ is called the *multidimensional time vector,* and it gives a time-index at which calculation of the *set* of variables defined at index $i$ will be *started.* The vector $p$ is the *space vector.* This vector gives the coordinates of the processing element (PE) where the set of variables defined at index $i$ will be executed. Four elements are distinguished in this transformation:

- The first $n-3$ rows of the transformation matrix are called the *multidimensional scheduling vectors.*
- The vector $\Pi_{n-2}$ is the *low-level scheduling vector*, corresponding to the conventional "time" assignment [12, 15].
- The last two rows correspond to the *placement vectors,* which form the *placement matrix* $\tilde{S}$.
- The column-vector $\gamma$ contains the *skewing parameter* $\gamma_A$. This parameter determines how much the calculation of a variable $A(i)$ is delayed relatively to the scheduling determined by $t$. This permits dealing with potentially complex processing elements that contain, for instance, internal pipelining.

```
for $g = 1$ to $G$
    for $h = 1$ to $H$
        for $j = -N$ to $+N$
            do $(i, k, l)$ in parallel
                $\Delta(g, h, i, j) = \sum_{k,l} |N(\ldots) - O(\ldots)|$;
                $\Delta_{opt}(g, h, j) = \min_i \Delta(g, h, i, j)$;
            next $(i, k, l)$
            $\Delta_{opt}(g, h) = \min(\Delta_{opt}(g, h), \Delta_{opt}(g, h, j))$;
        next $j$
    next $h$
next $g$
```

**Figure 9** Mixed parallel/sequential execution of the ME application.

Since only two placement vectors are used, the vector $p$, and thus also the architecture resulting from space-time mapping, will be two-dimensional. The other indices of the bijective mapping are attributed to a multidimensional time, which has to be sequentialized in order to be executable.

Multidimensional scheduling can be interpreted as follows: the first scheduling vector $\Pi_1$ partitions the index space $D$ into subdomains $D_i$ by using the relation $\Pi_1 \times I = i$. Each of these subdomains can be further partitioned by the other multidimensional scheduling vectors. This results in a set of three-dimensional subdomains $D_{t_1,\ldots,t_{n-3}}$. Each of these can be executed in parallel on a 2-D architecture. The scheduling of the 3-D subdomains on this 2-D architecture is given by the low-level scheduling vector $\Pi_{n-2}$ and the skewing parameter $\gamma_A$. The subdomains are executed sequentially on the same architecture, which can be derived from the placement vectors.

Applied to the ME application, this results in the mixed sequential/parallel execution given in figure 9. This corresponds with the parallel evaluation of the distance measures for all possible positions of the current block in its reference window at a certain height. The sequential execution follows the sequential scan of video images: first the top row of blocks is considered and, within this row, the blocks are evaluated from left to right. For a block at position $(g, h)$, the distance measure for all motion vectors with the same vertical displacement are evaluated in parallel. The set of vectors with the smallest vertical displacement is executed first, whereas the largest vertical displacement is considered last.

Using the linear transformation approach [12] results in pure systolic arrays: all operations of one index point (i.e., the kernel) are executed in one and the same clock cycle on their own piece of hardware. For algorithms with a complex kernel, this leads to large PEs with long critical paths. In that case, PEs must be pipelined and some hardware units may be shared for similar operations. This introduces the need to order/schedule the operations of one index point. For this, an affine timing function, using an operation dependent skewing factor $\gamma_A$, is necessary. Affine scheduling also becomes necessary when (limited) rippled interconnections must be allowed, and when dependence loops between index points in the dependence graph must be broken. It has also been shown [18] how local PE-scheduling and the global time-ordering of index points can be separated, allowing the use of better matched cost-functions and resulting in smaller optimization problems.

## 2 Optimization criteria

The ultimate goal of the overall space-time mapping step is to find a transformation matrix $\tilde{T}$ and a set of skewing parameters $\gamma_A$ that result in an optimized design. The optimization criteria include:

- Matched throughput. If possible, the mapping should match the requested throughput, since applying clustering techniques to reach matching will result in area overhead.
- Optimized PE design. The optimization of the PE design depends on the clock frequency needed for matched throughput, the level of hardware multiplexing used, the number of pipeline registers (as few as possible, as many as needed).
- Minimal I/O buffering. Arrays of I/O data should be spread in time as much as possible to reduce buffer cost. Also, the "natural" ordering of data-streams (as, e.g., the scanning order of video-images) should be followed as closely as possible.
- Efficiently used hardware. Hardware that is used only during part of the cycles should be avoided. If all PEs are used inefficiently, passive clustering can be applied, or if only part of the PEs are used inefficiently, selective clustering can be used. But all the clustering methods (see below) introduce overhead, so they should be avoided as much as possible by first choosing an optimal combination of localization and space-time transformation matrix.

- I/O and control signals should only be applied to PEs at the border of the architecture to avoid routing of these signals to the internals of the architecture.

- The load of the PEs should be balanced, i.e., ideally all PEs should perform the same amount of work for one pass of the algorithm.

These principles have been formalized in a method that has been applied to the ME demonstrator [19]. The transformation matrix at the left of equation (6) was found to be optimal.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} t \\ x \\ y \end{bmatrix} = \begin{bmatrix} 8 & 1 & 1 & 2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} i \\ j \\ k \\ l \end{bmatrix} + \begin{bmatrix} \gamma_A \\ 0 \\ 0 \end{bmatrix} \qquad (6)$$

In this case, it was possible to "linearize" the multidimensional timing in a simple way, and only internal pipelining is necessary within the PE with the appropriate skewing parameters $\gamma_A$ [19].

The resulting array architecture is shown in figure 10. The PE design is shown in figure 11. Note that this PE-design is the maximal one, performing all operations occurring in the complete application description. It is only needed in the cell at the lower right corner in figure 10. The PEs in the lowest row of cells in the array require only the first comparator block, whereas all the other rows do not require any comparators. Hence, most cells use only the first two or three stages of this pipelined data-path.

The optimization goals were reached to a large extent. Only 248 bytes of buffer memory outside of the array are needed, all PEs are active at 100% of all cycles, and the architecture has matched throughput. Communication inside the array is also simple, and no control or I/O is needed for PEs not located at the borders. This fully optimized result has been feasible by adopting the tuned methodology proposed in section 2 and with the help of the synthesis techniques discussed in the previous sections.

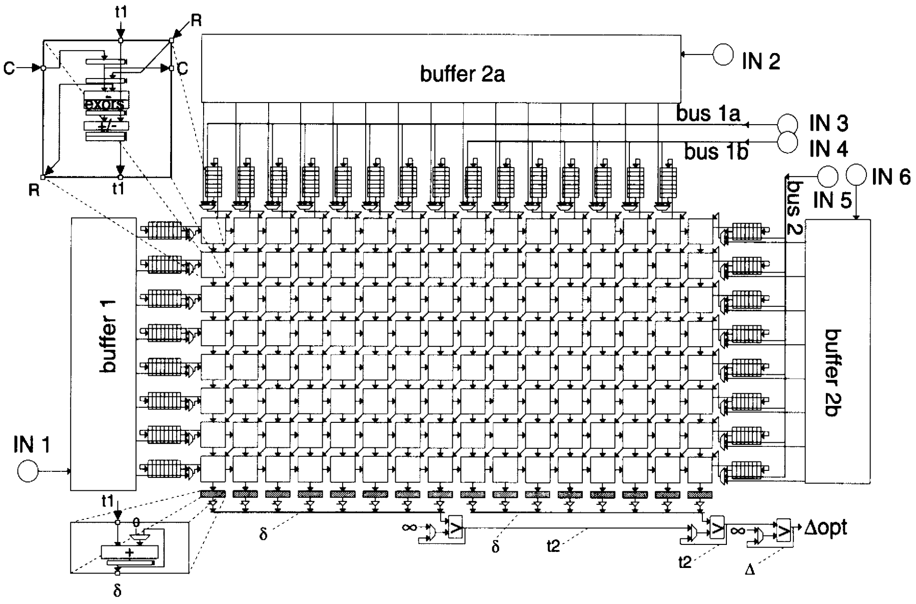


**Figure 10** Final architecture for the motion estimation algorithm.

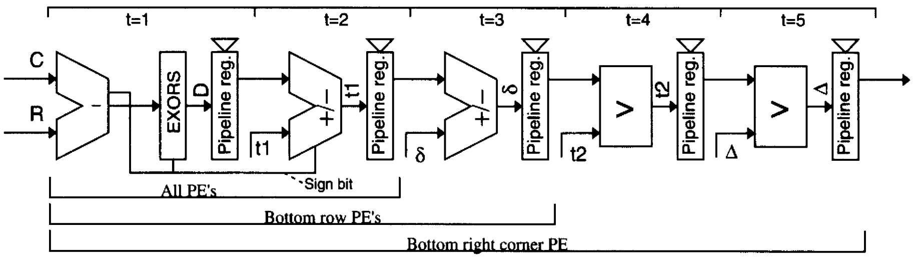


**Figure 11** The PE architecture for the ME example.

## 3 Comparison with related work in other chapters

The optimization strategy of the array design method developed at ENSL (see chapter 3) is latency-driven. The approach to first find the schedule with the minimal latency (which is largely dependent on the longest dependence path in the algorithm), and then the smallest architecture compatible with this schedule, would possibly result in a smaller architecture than the one presented here. The throughput, as defined by the block pipelining period, of such an architecture would be insufficient, however, because that criterion is not explicitly used in this complementary methodology.

The extensions presented in chapter 3 to allow (partial) broadcasting are, however, compatible with the approach presented here and could be used to reduce the number of pipelining registers of the architecture.

The overall design method developed at University of Patras (see chapter 5) is also not directly oriented to our particular application domain. This alternative method exploits the existence of independent subsets of variable instances, but these are not present in many video processing kernels, so no gains are possible here. Also, since the approach is partly latency-driven, an architecture with less than 100% PE load would be the result. However, the extension to bit-level pipelining can be integrated with the approach presented here to arrive at architectures with extremely high clock frequencies. This was not required for the ME demonstrator.

## 4 Clustering transformations

The last step in the methodology of section 2 involves array clustering techniques to map the architecture produced after space/time transformations on a smaller fixed size array, as described in chapters 3 and 4 [13, 1]. For the ME application, this clustering stage is not needed since the carefully chosen transformation matrix results in an architecture with already matched throughput and size. The PE usage is 100%, so passive clustering techniques to improve PE usage are also superfluous.

It must be stressed that, in general, these transformations are a necessity for many image and video algorithms. Indeed, the size parameters of the architecture are a function of the parameters of the application. The large image size parameters may therefore result in very large architectures. Given the large number of PEs, such architectures will be much too fast in most cases, necessitating the use of clustering techniques to adapt the throughput of the architecture.

## 8 CONCLUSION

It has been demonstrated that the proposed design script for regular array synthesis leads to efficiently designed architectures. The script has been developed in the context of the CATHEDRAL project. The re-indexing and localization tools, developed respectively at IMEC and IRISA, can be used to allow a higher input behavioral description of an application. The transformations provided by these tools can also be useful in other architecture synthesis areas.

The example design of a motion estimation application using the design script resulted in an architecture that can be compared with even the best manual designs. The careful selection of the transformation matrix produced an architecture with optimal I/O and control flow. Only control logic and I/O at the borders is needed, and the interface buffer sizes needed are very small. This shows that the proposed methodology and tools can indeed be used for complex real-life examples, with optimal results.

## REFERENCES

[1] J. Bu and E. Deprettere. A constructive procedure for processor clustering and array optimization. *Proc. International Symposium on Circuits and Systems*, pages 248–251, 1991.

[2] C. Dezan, E. Gautrin, H. Le Verge, P. Quinton, and Y. Saouter. Synthesis of systolic arrays by equation transformations. *Proc. of the IEEE International Conference on Application Specific Array Processors.* IEEE Computer Society Press, Sep 1991.

[3] R. M. Karp, R. E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the Association for Computing Machinery*, 14, number 3, pages 563–590, Jul 1967.

[4] S. Y. Kung. *Systolic array processors: performance analysis and design optimization*, chapter 4.4, pages 226–248. Prentice Hall, 1988.

[5] D. Lanneer, G. Goossens, F. Catthoor, M. Pauwels, and H. De Man. An object-oriented framework supporting the full high-level synthesis trajectory. *Proc. 10th Intnl. Symp. Comp. Hardw. Descr. Lang.*, CHDL-91, Marseille, France, pages 281–300, Apr 1991.

[6] H. Le Verge. Reduction operators in ALPHA. *Parle'92*, Paris, Jun 1992.

[7] H. Le Verge, C. Mauras, and P. Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3, pages 173–182, 1991.

[8] H. Le Verge and P. Quinton. Derivation of regular parallel algorithms with the ALPHA language. In J. P. Banâtre and D. Le Metayer, editors, *Research Direction in High-Level Parallel Programming Languages*, pages 298–308. Springer-Verlag, 1992.

[9] C. Lin and S. Kwatra. An adaptive algorithm for motion compensated color image coding. *IEEE Globecom*, 1984.

[10] P. Lippens, J. van Meerbergen, A. van der Werf, W. Verhaegh, B. McSweeney, J. Huisken, and O. McArdle. PHIDEO: a silicon compiler for high speed algorithms. *Proc. 2nd ACM/IEEE Europ. Design Automation Conf.*, Amsterdam, The Netherlands, pages 436–441, Feb 1991.

[11] C. Mauras. ALPHA: un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones. Thèse de l'Université de Rennes 1, IFSIC, Dec 1989.

[12] D. Moldovan. Advis: a software package for the design of systolic arrays. *Proc. IEEE Int. Conf. on Computer Design*, Port Chester NY, pages 158–164, Oct 1984.

[13] H. Nelis and E. Deprettere. Automatic design and partitioning of systolic/wavefront arrays for VLSI. *Circuits, Systems and Signal Processing*, 7, number 2, pages 235–252, 1988.

[14] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* John Wiley & Sons, New York, 1988.

[15] P. Quinton. Automatic synthesis of systolic arrays from recurrent uniform equations. *Proc. 11th Int. Symp. Computer Architecture*, Ann Arbor, pages 208–214, Jun 1984.

[16] P. Quinton and V. van Dongen. The mapping of linear recurrence equations on regular arrays. *Journal of VLSI Signal Processing*, 1, pages 95–113, 1989.

[17] S. Rajopadhye. *Synthesis, verification and optimization of systolic arrays.* PhD thesis, The University of Utah, Dept. of Computer Science, Dec 1986.

[18] J. Rosseel, F. Catthoor, and H. De Man. Extensions to linear mapping for regular arrays with complex processing elements. *Proc. of the IEEE International Conference on Application Specific Array Processors*, pages 156–167. IEEE Computer Society Press, 1990.

[19] J. Rosseel, F. Catthoor, and H. De Man. The systematic design of a motion estimation array architecture. *Proc. of the IEEE International Conference on Application Specific Array Processors*, pages 40–54. IEEE Computer Society Press, 1991.

[20] J. Rosseel, F. Catthoor, and H. De Man. The exploitation of global operations in affine space-time mapping. In Kung Yao et al., editors, *VLSI Signal Processing V*, pages 309–318, IEEE Press, 1992.

[21] V. Roychowdhury, S. Rao, L. Thiele, and T. Kailath. On the localization of algorithms for VLSI processor arrays. In R. W. Brodersen et al., editors, *VLSI Signal Processing III*, pages 459–470, IEEE Press, 1988.

[22] M. van Swaaij. Data flow geometry: exploiting regularity in system-level synthesis. PhD thesis, K. U. Leuven, Belgium, Dec 1992.

[23] M. van Swaaij, F. Franssen, F. Catthoor, and H. De Man. Modeling data flow and control flow for DSP system synthesis. *Proc. of the European Design Automation Conference*, pages 8–13, 1992.

[24] M. van Swaaij, J. Rosseel, F. Catthoor, and H. De Man. Synthesis of ASIC regular arrays for real-time image processing systems. In E. Deprettere et al., editors, *Algorithms and Parallel VLSI Architectures*, Vol. B, pages 329–342, Elsevier Science, 1991.

[25] Y. Yaacoby and P. Cappello. Scheduling a system of nonsingular affine recurrence equations onto a systolic array. *Journal of VLSI Signal Processing*, 1, pages 115–125, 1989.

7

# MEMORY AND DATA-PATH MAPPING FOR IMAGE AND VIDEO APPLICATIONS

**Werner Geurts[1], Frank Franssen[1]**
**Michaël van Swaaij[1], Francky Catthoor[1]**
**Hugo De Man[1], Marc Moonen[2]**

[1] *IMEC, Leuven*
[2] *ESAT, Univ. of Leuven*

## ABSTRACT

In this chapter, we will present a high-level synthesis methodology that is particularly suited for irregular high-throughput subsystems realized on an application-specific architecture. This CATHEDRAL-3 methodology is targeted to real-time signal processing applications with a low potential for time multiplexing, as occurring, for example, in image and video applications. The most crucial steps in this methodology are supported by appropriate synthesis techniques embedded in prototype tools. The emphasis lies on high-level synthesis supporting the dominant design cost factors, i.e., an area-efficient memory organization and a customized data-path configuration, both within the stringent throughput requirements. The power of the approach will be illustrated with realistic demonstrators.

## 1 INTRODUCTION

The CATHEDRAL-3 architecture synthesis methodology that we present is targeted to real-time signal- and data-processing applications that are data-flow-dominated and have a low potential for time multiplexing [18, 8]. It is particularly suited for the important domain of irregular high throughput applications, which must be mapped on application-specific architectures due to throughput bottlenecks or due to severe restrictions on area or power. Some examples of target applications are medium-level and front-end image and video processing modules, front-end audio, and user-end telecom [1].

This target application domain features a number of important characteristics which are heavily exploited in our target architecture style and our synthesis approach [1, 18, 8]:

- A regular computation-intensive signal flow—as occurs in algebraic analysis, filtering, or format conversion—is combined with nested branches and multiple data-dependent loops. This explicit irregularity complicates not only the controller synthesis but also other synthesis tasks, like scheduling and data-path allocation. These tasks need to deal with control-flow hierarchy explicitly, which is an important aspect of our approach (see section 3).
- Multidimensional signals of a substantial size (index ranges up to a few thousand) and dimensionality (up to four or five nested indices) are present within the loops. These need to be stored in bulk memories or in medium-size distributed memories, resulting in a severe memory management problem. Therefore, optimization of the memory size and the number of access ports, and especially transforming the control flow of the algorithm to allow a significant improvement of the memory-related cost, are important tasks in our synthesis script (see section 2). Actually, this design phase is even performed *before* the data-path mapping is started.
- The operations in the high-level specification are not only restricted to linear signal processing like multiply-accumulate but include also nonlinear operations. Moreover, the complexity of the hardware on which these operations need to be mapped can differ substantially. For instance, a multiplier will, even when it is pipelined, be slower and much larger than a hard-wired shifter. This issue leads to an additional complexity in module or operator allocation and selection (see section 3). Indeed, in order to maximally exploit a unique clock period $T_{CL}$ for our synchronous target architecture style, operators will need to be chained within a pipeline section, resulting in complex customized data-paths (see figure 7). These are commonly called application-specific units or ASUs. The complexity of these is usually larger than the highly multiplexed data-paths in chapters 8 and 9 due to the difference in target domain. High-level operations (such as division and floating-point multiplication) will need to be expanded into lower level operations that are better matched to these ASUs.
- The number of operations that have to be performed within a time frame is much larger than the achievable amount of operations per clock cycle on a single resource. The former may range from hundreds to thousands of millions of operations per second (MOPS), while the latter might be on

the order of 20 MOPS. This will require the allocation of a large number of parallel resources of different types. Moreover, the cost of the interconnections that are required to let these resources work in parallel is an important optimization criterion.

- The maximum hardware-sharing factor HSF = $T_{EVAL}/T_{CL}$ is typically in the range of 1–20. Here, $T_{EVAL}$ is the amount of time, specified by the designer, that is available for the evaluation of one instance of the algorithm. For signal processing applications with a fixed rate and single input and output signals, $T_{EVAL}$ is usually the sample period. This HSF expresses how many times any particular resource can be re-used or shared during the evaluation of one instance of the algorithm. Since the HSF is low but usually larger than 1, hardware should be partly shared, while still incorporating the stringent timing bottlenecks present in the application. As a result, the operator or data-path assignment problem is a crucial step during data-path-related synthesis (see section 3). Moreover, scheduling becomes extremely constrained and needs to deal with heavily pipelined operators or data-paths.

The traditional approach [13] toward synthesis for high throughput applications is to perform pipelined scheduling first, minimizing the number of functional units [21, 7, 11]. This step is followed by binding [7, 22], which aims to minimize the functional unit, register, and interconnect cost. The result of this so-called scheduling-first approach is that a large number of operations are distributed over many operators and it becomes very difficult to route the intermediate variables from their source operator to the destinations [18]. This is reflected in the large number of multiplexers and busses that are required. The irregularity introduced by the scheduling also results in an unacceptable growth in the size of the control unit.

We have introduced a synthesis methodology for data-path mapping that preserves the inherent structure of the algorithm [18]. This permits mapping highly resembling clusters of operations onto the same complex data-path. More details can be found in section 3, where our approach will be illustrated with a typical realistic demonstrator. There, a comparison will be made with some other recent approaches, HYPER [25] and PHIDEO [9], which are also dealing with hierarchically organized, high-throughput applications. Very few approaches in literature address the crucial memory management issues. In contrast, our methodology already provides an effective solution for several key aspects of this important problem. The results for a realistic demonstrator will be presented in section 2.

The synthesis objective that will be used throughout this chapter is the minimization of the total area consumed by all resources, i.e., memories, operators, and interconnect, under a user-specified throughput constraint (e.g., the sample rate). This is compatible with the requirements for real-time signal processing systems that are targeted to customized architectures but not fully power-dominated. Extensions to other optimization objectives, based on cost factors like low power, are feasible but will not be discussed here.

## 2 HIGH-LEVEL MEMORY MANAGEMENT

In most real-time signal- and data-processing systems, large quantities of data are processed. These data are most often specified as multidimensional (M-D) signals, i.e., arrays with one to four dimensions. Due to the applicative nature of our specification language, we frequently obtain intermediate signals with even up to six dimensions. Processing these large M-D signals in real time not only poses computational problems; their main effect lies in the required storage capacity and the access bottle-neck they cause between large-scale memories and the arithmetic processing in data-paths [1, 9, 33]. Examples of such systems are abundant in image, video, speech and radar processing, measurement systems, graphics, and in automotive and audio processing. The M-D signals are referenced by index expressions which may not be assumed to be linear and manifest; nonlinear and even data-dependent expressions occur.

This section provides an overview of the work performed on high-level synthesis methods intended for supporting the design decisions on M-D signal storage and access. The proposed synthesis script, i.e., the sequence of subtasks plus their optimization objective, will be provided together with an illustration of the power of this novel approach by means of a realistic demonstrator: an auto-correlation algorithm as, for example, needed in a CD audio interpolator [33].

### 1 The auto-correlation test-vehicle

The algorithm used in the auto-correlation application is given in SILAGE [6] notation in figure 1. The node numbers in the SILAGE code are used as labels for the corresponding recurrence equations. They are used as a reference throughout this section. Essentially, an input signal stream `in[i]` is broken up into consecutive sets of $N + P + 1$ signal elements. These are preprocessed with a simple scalar function `fin()`, resulting in the new set `s[i]` (node 1).

```
func main ( in : INT_S[NplusP] ) : INT_WM[] =
  begin
  (i: 0 .. Nmin1plusP) :: s[i] = fin (in[i]);          /* 1 */
  (i : 0 .. P ) :: r[i][0] = INT_AC(0);                /* 2 */
  (j : 0 .. Nmin1 ) ::
    begin
    a[j] = f1 ( s[j] );                                /* 3 */
    (k : 0 .. P ) ::
      begin
      b[k][j] = f1 ( s[k+j] );                         /* 4 */
      r[k][j+1] = r[k][j] + INT_AC ( a[j] * b[k][j] ); /* 5 */
      end;
    end;
  iscl0 = f3 ( r[0][N] ) - 15;
  help = f2 ( r[0][N] , -iscl0 );
  return[0] = f4 ( MAXM , help );
  (z: 1 .. P) ::
    begin
    return[z] = f5 ( r[z][N] , -iscl0 );               /* 6 */
    end;
  end;
```

**Figure 1** Initial SILAGE description of auto-correlation algorithm.

The description used here also includes the casting function `f1()` on the input samples. Then, each subset of $N$ elements (produced with the `k` iterator in node 4) is compared to the subset of the first $N$ elements (produced in node 3). This comparison uses an accumulated multiplication of the two sets (node 5) for each value of the iterator `k`, resulting in the auto-correlation coefficients `r[][]`. In the CD interpolation application, these coefficients are considered as Toeplitz matrix elements after some further postprocessing embedded in the functions `f2()`-`f5()`.

This description results in two sets of signal broadcast operations, labeled as node 4 and node 5. These in particular will result in a potentially large number of storage locations during memory management.

Note that statements defining a single signal instance—like `iscl0`, `help`, and `return[0]`—and simple assignments of one M-D signal to another—like `a[i] = b[i]`—are not considered as relevant M-D signal definitions at this stage. Hence, they are pruned from further analysis to decrease the complexity [31]. That is also why the functions `f1()`-`f5()` have been introduced in the description.

The parameter set used in the demonstrator corresponds to $N = 512$ input samples and 51 Toeplitz coefficients, i.e., a maximal shift of $P = 50$ relative to the input samples.

## 2 Motivation for a novel model compared to state-of-the-art

A primary task in high-level architecture synthesis is the extraction of the data flow from a given algorithm description. The term *data flow* is defined here as the combination of operations and dependencies between them that define the algorithm. In contrast, *control flow* is then defined as a (partial) ordering of computations meeting the restrictions of their dependencies. The control flow is usually specified by introducing loops and function hierarchy.

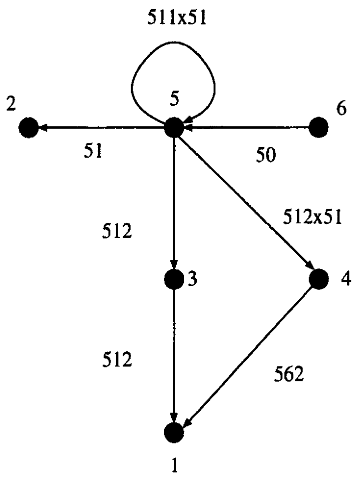
In order to arrive at efficient synthesis results, it has been recognized that transformations on algorithm specifications are crucial. Until now, this issue has been investigated mainly for data flow transformations on scalar processing, as in digital filters [20, 25]. Also, loop transformations have been studied in the context of software optimizing parallel compilers [10, 17, 19, 23]. We claim that transformations are even more crucial for the control flow in the presence of M-D signals. Indeed, studies on the effect of memory organization and the way M-D signals are stored in memory [33, 9] have shown that, for instance, loop transformations have effects that cause differences in realization cost of several orders of magnitude, rather than on the order of a few dozen percent, as many other types of optimizations in the remainder of the synthesis trajectory. Hence, in order to allow for maximal flexibility in control flow optimization, none of the possible control flows should be excluded in advance from the search space. This is, of course, with exception of those incoherent with a given data flow or those explicitly marked as undesirable by the designer. In other words, the *syntactical structure* of an algorithm specification *should not be used* to limit the set of possible control flows in the realization, as generally happens in conventional approaches.

Formalization of optimization tasks in the synthesis process can only be done when data flow and control flow are modeled properly and rigorously. Formalization is necessary to allow *globally* optimal design decisions, and also to be *more independent* from the way an algorithm has been specified. It has to be stressed that when dealing with M-D signals, it is in most practical cases not acceptable to "unroll" the nested loops because this leads to a huge number of scalars that cannot be handled in a realistic way. This means that scalar methods based on signal flow graphs (SFGs) as used in conventional synthesis [13] are not sufficient. Moreover, it is in general also insufficient to use enumeration-based symbolic analysis [34, 28], since in many applications—for example, in image and video processing applications—complexity becomes too high. For the irregular, high throughput target domain that we envision, much of the exploration freedom is also neglected when the ordering within the M-D signal constructs as specified by the designer is retained [9]. In general, *all* signal dependencies are indeed relevant to steer the required control flow transformations with a sufficient quality of the result, especially when subsets of signals are defined or consumed in different ways. Some nonsymbolic approaches that have been used for data flow analysis, such as solving an integer linear programming (ILP) formulation [5], also do not provide a sufficient modeling for this purpose [2]. For linear functions, the problem has been addressed in the field of regular array synthesis [24]. This work has been used as a basis here, but it is in itself not sufficient to deal with the memory-related aspects for irregular applications. Therefore, in the context of ASCIS and NANA, an extended polyhedral-based model has been presented that meets all the necessary requirements [29, 32, 2].

## 3 The principles of the model

A linear submodel has originally emerged as an efficient solution to extract hidden regularity in the context of regular array synthesis [29], as already described in section 5 of chapter 6. Extensions to describe nonlinear and data dependent M-D indexing behavior have been proposed, too [2]. The requirement for this added functionality is frequently encountered in, for example, image processing applications, due to the direct application of modulo functions on affine iterator (index) functions, or indirectly by the presence of conditional signal definitions that are used as indexing signals [2, 6].

In the proposed model, signals are defined in an M-D index space by means of index boundaries based on polytopes, i.e., convex M-D regions bounded by inequalities [16]. This has already been illustrated for a single M-D signal definition (by means of a recurrence equation) in chapter 6. The scope of the operation involved is defined as the region or domain on the lattice, set up by loop iterators and bounded by the iterator ranges, also called the *node space.* In a similar way, polytopes can be related to the signals defined by and used by operations, called *definition* and *operand spaces.* The relations between the M-D signals can then be described according to a dependence graph between the polytopes (or polyhedrals in general). This leads to the polyhedral dependence graph (PDG) concept, in which nodes represent the scope of statements (node



$$f(x) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} x + \begin{bmatrix} -1 \\ 0 \end{bmatrix}$$

$$P_a = \begin{bmatrix} 0 & 1 \\ 0 & -1 \\ 1 & 0 \\ -1 & 0 \end{bmatrix} x \geq \begin{bmatrix} 0 \\ -511 \\ 0 \\ 50 \end{bmatrix}$$

$$P_b = \begin{bmatrix} 0 & 1 \\ 0 & -1 \\ 1 & 0 \\ -1 & 0 \end{bmatrix} x \geq \begin{bmatrix} 0 \\ -512 \\ 0 \\ 50 \end{bmatrix}$$

**Figure 2** PDG of auto-correlation data flow and the intradependency relation for node 5.

spaces) and edges represent dependency relations given by M-D affine functions $f(x) = Ax + b$ [31].

An example of the relation between the different node spaces is now provided using our auto-correlation demonstrator application. The PDG for this algorithm is depicted in figure 2. The affine functions to be annotated to the edges are the result of mapping node spaces on their definition $P_a$ and operand spaces $P_b$ (see also figure 3 for the graphical presentation of the corresponding polytopes). Each of the polytopes, within its space, can be mathematically denoted by a set of inequalities $Cx \leq c$. An example is provided for the intradependency from node 5 to itself.

Because the optimization of control flow for minimizing memory storage (see below) is based on dependency costs between the different node spaces, the exact dependency relations are relevant information. For example, the exact number of dependencies that exist between the operations of node space 5 is determined by computing the intersection of both inequality sets belonging to polytopes $P_a$ and $P_b$. For node 5, this results in the inequality set of $P_a$, which means that $511 \times 51$ intradependencies are present. This information and the corresponding affine function $f(x)$ for node 5 is depicted in figure 2. To each edge of the PDG in figure 2, the number of dependencies is annotated.

In summary, the importance of using polytopes to indicate sets of lattice points is that this model is precise, it is concise, and it allows for easy dependency analysis without symbolic simulation. Furthermore, many powerful mathematical methods and techniques are available for handling polytopes [16, 26]. These methods and techniques can be readily used in the analysis and optimization problems.

## 4 Memory-related algorithm optimization

One of the key stages in organizing the memories is the selection of the initial control flow, in particular the loop organization and the indexing equations. Once this choice has been fixed, a procedural interpretation of the control flow can be used to derive an initial access and storage scheme. From the access scheme, an optimized number of memories and memory ports can be derived: for example, by a stream-based technique [9]. For this given memory allocation, the in-place storage for the M-D signals can be decided whenever their life times are (partially) nonoverlapping. Initial research has resulted in techniques that can deal with M-D signals [34, 28]. Other recent results are described in chapter 8. Finally, also the address-related issues can be dealt with [9, 28]. However, it has been shown that the initial control flow specification has a very pronounced effect on the eventual results of the other memory management stages [34, 31]. Typically, the control flow selection is now performed in an ad-hoc manner by the system designer, with little or no formal foundation or feedback on the effects. Therefore, synthesis techniques to support the designer with this decision are crucial but were totally missing until now.

We have derived such a technique founded on the extended PDG model. In this PDG, control flow can simply be expressed by a placement of the polytopes in a common Euclidian space, together with an ordering vector defined over this space [29, 30], as already described in chapter 6 for array synthesis. Techniques to steer this placement and ordering with cost functions directly associated with the crucial memory-related costs of access bandwidth and storage location count have been proposed [31]. Note that in our PDG control flow model, changes of a placement and ordering vector have *explicitly measurable* effects on optimization criteria such as the minimum required number of storage locations (the maximum number of dependencies cut by the hyperplane perpendicular to the ordering vector) and access bandwidth or parallelism (the operations on the same hyperplane). These types of criteria are not explicitly measurable in SFG- or DFG-type models [13]. Addressing cost is also implicitly incorporated in our model, since the affine index functions for the eventual M-D signals result

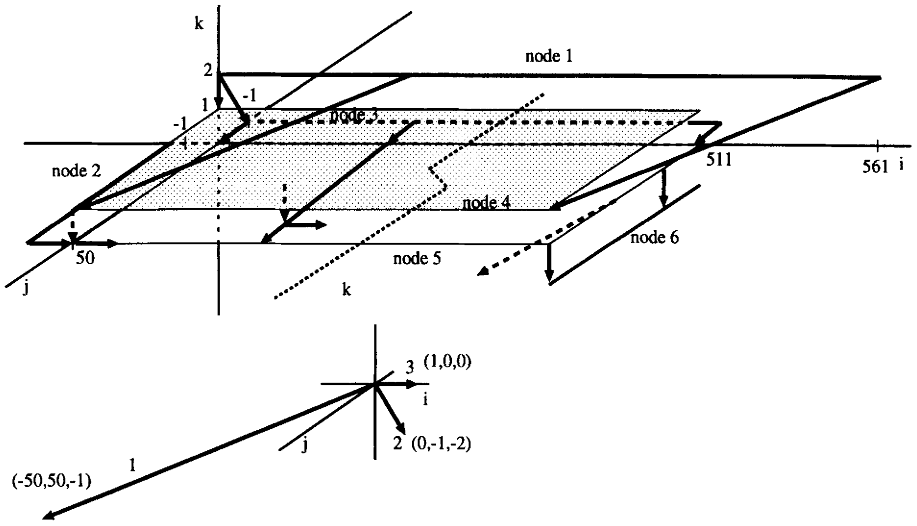


**Figure 3** Common node space topology after PDG placement for the auto-correlation with $N = 512$ and $P = 50$.

in linear address equations which can be efficiently realized in an incremental way with simple offset-based additions on a modulo basis.

The impact on storage requirements for the auto-correlation application will now be studied as an illustration.

### *PDG placement*

Given the algorithm data flow structure of figure 1, modeled by the PDG model, the definition of a control flow is performed in two phases. First, the placement of individual domains is done in a three-dimensional common node space. This placement step is performed incrementally, steered by accurate mixed ILP optimization techniques [31]. The result is depicted in figure 3. The extreme points of node spaces 1 and 3 are located at [0,0,2] and [561,0,2] and at [0,–1,0] and [511,–1,0], respectively. This means that both spaces are nicely aligned to each other and connected by 512 dependencies with direction [0,–1,–2]. The extreme points of node space 4 are located at [0,0,1], [511,50,1], [511,0,1] and [0,50,1].

Arcs in figure 2 represent the dependency directions in and between the different node spaces. For instance, the arcs running over node space 5 represent all

```
(i : -2 .. 562) ::
(j : -1 .. 561) ::
(k : -50 .. 1) ::
begin
return_29[-j-k+511][i-j+k-1][-i-k+512] =
      if((j==511) & (2*i==j-2*k+513) & (j<=-k+510) & (j>=-k+461))
      -> f2(r_56[-j-k+511][512][0])fi;
b_52[-k][j][-i+j-k-1] =
      if((i==j-k-1) & (j>=0) & (j<=511) & (k<=0))
      -> f1(s_58[j-k][0][0])fi;
r_56[-k][j+1][-i+j-k] =
      if((i==j-k) & (j>=0) & (j<=511) & (k<=0))
      -> add(r_56[-k][j][0],a_54[j][0][0],b_52[-k][j][0])fi;
s_58[j][-k][-i+j-k-2] =
      if((i==j-2*k-2) & (i==j-2) & (j>=0))
      -> fin()fi;
r_56[-j-k-1][-j-1][-i-k-1] =
      if((j==-1) & (2*i==j-2*k-1) & (j<=-k-1) & (j>=-k-51))
      -> nil()fi;
a_54[j][-k+1][-i+j-k] =
      if((i==j-2*k+1) & (i==j-1) & (j>=0) & (j<=511))
      -> f1(s_58[j][0][0])fi;
end;
```

**Figure 4** Re-indexed auto-correlation code after placement and ordering.

broadcast dependencies over a single column in the $j$ direction. The solution provided here is heavily optimized. As an example, look at the placement of node space 6, which has extreme points at [511,50,–1] and [511,1,–1]. This is optimal in terms of dependency length cost (related to storage locations) toward the other polytopes. Because a similar dependency direction is present between the spaces of nodes 4 and 5, this solution is also not unnecessarily constraining the search space for an ordering vector that is determined next.

## *Selecting an ordering vector*

Given this placement, an optimized control flow can now be expressed by applying a single affine transformation on the complete common node space in such a way that the new base vectors (iterators) indicate the sequence of loop ordering [31].

The placed and ordered result can be represented by a completely re-indexed version of the initial description in figure 1. The new (incomplete) SILAGE code is listed in figure 4.

The result of the ordering transformation on the common node space is indicated by three vectors at the bottom of figure 3. The three vectors indicate the order in which the operations in the common node space are to be computed. Originally, the inner $k$-iterator is the fastest changing iterator and the outer $i$-iterator the slowest one. In the optimized result, the fastest movement is along the [50,–50,–1] direction, followed by the [0,–1,–2] and [1,0,0] directions. This solution corresponds to a diagonal computation of the `r[][]` instances.

Note that although at first sight, the final code contains a large number of conditions, which may imply a complex control structure, most complex conditions can be reduced to simpler ones by a postprocessing stage. This is equivalent to finding a polytope representation with a minimal set of equations [12].

### *Effect on memory cost*

We will now illustrate the effect of the re-ordering on M-D signal storage requirements. The requirements can be computed per signal by using the window calculation method [28].

The control flow, defined by a procedural interpretation of the initial SILAGE description of figure 1, results in an inefficient solution. Indeed, the inner loop of the double nested loop construct is the fastest one executed. This type of control flow definition corresponds to a column-wise ordering of `b[][]` and `r[][]` instance computations. In terms of storage requirements, this means that all 562 input signal instances of `s[]` are read in before they are consumed. The same is true for all 50 instances of `r[0][]`, which are produced before one instance is consumed. Furthermore, $P$ intermediate signals `r[][]` have to be alive in parallel. The same is true for the column of `return[]` values.

This situation is improved significantly in the re-ordered description of figure 4. As a result of the diagonal ordering, the `a[]` instances have to be stored for the computation of $P(P/2)$ `r[][]` instances. Because the `a[]` instances are broadcast, the maximal number of instances of `a[]` to be stored is $P$. The instances of `b[][]` are broadcast over the diagonal, hence a single instance of `b[][]` is alive at any point of the ordering. The maximal number of simultaneously available instances of `s[]` is also 1, because the production of the broadcast `a[]` and `b[][]` instances requires the same `s[]` instance. Hence, at most $P$ intermediate signals `r[][]` have to be alive in parallel. Both the initial `r[][0]` instances and the `return[]` instances are required one at a time. These results are collected in table 1.

| Signal | Size | Initial | Optimized |
|---|---|---|---|
| `s[]` | 562 | 562 | 1 |
| `return[]` | 51 | 1 | 1 |
| `b[][]` | 26112 | 25600 | 1 |
| `a[]` | 512 | 1 | 51 |
| `r[][]` | 26112 | 25651 | 51 |
| `in[]` | 1 | 1 | 1 |

**Table 1** Initial signal sizes and their reduction for the optimized control flow definition given in figure 4.

From table 1, it is clear that the total memory requirements have been reduced with roughly three orders of magnitude. Based on these numbers per multi-dimensional signal, the total storage requirements can be further reduced by the in-place reduction of `r[][0]`, `r[][]` and `return[]` signals requiring only $P$ storage locations. This solution is equivalent with one found manually [33]. In general, the result is dependent on the values of the parameters. In order to show this, another experiment has been conducted for a second parameter set, with $P = 10$ and $N = 52$. In that case, the solution obtained is a column-wise computation, which corresponds to the one found manually.

Similar results have been achieved for other realistic test vehicles [31]. We would especially like to mention a complex updating singular value decomposition algorithm, needed, for example, in data acquisition [15].

### *Summary*

The experiments with the auto-correlation demonstrator have clearly illustrated that carefully deciding on control flow has a pronounced effect on the storage requirements. The potential effect is a reduction with orders of magnitude of the amount of storage locations needed. A similar effect has been observed for the access bandwidth, resulting in a significant decrease of the number of parallel memory ports needed in the architecture. The CPU time required for performing the control flow manipulation on this demonstrator amounts to 202 seconds on a DecStation 3100, which is very reasonable. This CPU time grows only linearly with the number of signal definitions and exponentially with the loop nesting degree (which in practice remains very limited). The optimal ordering also clearly differs for different parameters for the same algorithm.

One of the main reasons for the success of this approach is that in our PDG control flow model, changes of placement and ordering vectors have explicitly measurable effects on storage- and access-related optimization criteria. This is lacking in the conventional signal flow graph models.

## 3 HIGH-LEVEL DATA-PATH MAPPING

In section 1, it has been shown that we need complex application-specific data-paths or ASUs to match the characteristics of our target application domain. Moreover, the assignment of chained operations to these ASUs, the definition or their internal organization, and the pipeline scheduling have been identified as important subtasks during data-path mapping. It has also been motivated that the memory cost is very dominant, so the data-path mapping stage is postponed until after high-level memory management decisions related to M-D signal storage have been dealt with. The outcome of the memory management stage are constraints on the loop organization and the indexing, assignments of M-D signals to background memories, and address operations that have been added to the data flow graph.

The global objective of the CATHEDRAL-3 high-level data-path mapping design stage is then to map the basic arithmetic, logic, and relational operations in the application to a set of ASU architectures. For this purpose, a script specific to the target domain has been developed [18, 3, 8]. We will now illustrate the techniques for the most important subtasks by means of a realistic demonstrator design for a video conversion application.

### 1 A YUV-RGB video format conversion application

A color video signal which is produced by a camera or displayed by a cathode ray tube consists of the basic color components. In video applications, these are usually red, green, and blue. The human eye is, however, perceptually more sensitive for luminance variation than it is for color or chrominance variations. Therefore, it is possible to obtain a significant compression by re-encoding of the $(R, G, B)$ components to a luminance component $Y$, which has the same resolution as the original video signal, and two chrominance components $(U, V)$, which have half the resolution of the original video signal. In this way, the required bandwidth for transmission or the required memory capacity for storage can be reduced significantly.

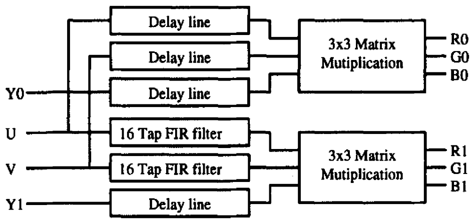


**Figure 5** Block diagram for YUV to RGB conversion application.

At the receiver side, the inverse conversion from $(Y, U, V)$ to $(R, G, B)$ has to take place. The block diagram for the main kernel of this conversion algorithm is given in figure 5, and the SILAGE [6] specification is shown in figure 6. Four main signal processing blocks are present: two interpolation filters, one per chrominance component; and two matrix operations, one for each set of outputs.

## 2 Decisions on data-path organization

First, the basic arithmetic, logic, and relational operations in the application are mapped to a set of dedicated ASUs. This is done in four basic synthesis steps in the script [18, 3, 8].

### *Data-path optimizing transformations*

The objective here is to steer high-level transformations on the DFG, including the application of algebraic laws, decisions on the remaining control flow freedom, and modification of operation cluster boundaries. Both global and local steering mechanisms are under development. For the YUV-RGB application, manually driven transformations have already been performed on the description in figure 6. For example, algebraic laws such as associativity have been applied to the FIRs to arrive at the special flow graph arrangement specified. Also, multi-rate to single-rate transformations have been applied already. As a result, the global sample rate requirement has become 6.75 MHz.

### *Operation clustering*

Here, operations are grouped into clusters in order to exploit the structure present in the flow graph. The objective function is cluster similarity within the timing constraints imposed. Within the single clock cycle constraint, which

```
#define WD fix<18, 0>
#define WC fix<12,11>
func main(U, V, Y0, Y1:WD) B1, G1, R1, B0, G0, R0: WD =
begin
/* FIR for U signal */
u0a =       (WD(C15 * (U   + U@15)) +
             WD(C13 * (U@1 + U@14)) );    /* FIR_U_a */
u0b = u0a + (WD(C11 * (U@2 + U@13)) +
             WD(C9  * (U@3 + U@12)) );    /* FIR_U_b */
u0c = u0b + (WD(C7  * (U@4 + U@11)) +
             WD(C5  * (U@5 + U@10)) );    /* FIR_U_c */
u0d = u0c + (WD(C3  * (U@6 + U@9 )) +
             WD(C1  * (U@7 + U@8 )) )<<1; /* FIR_U_d */
/* FIR for V signal */
v0a =       (WD(C15 * (V   + V@15)) +
             WD(C13 * (V@1 + V@14)) );    /* FIR_V_a */
v0b = v0a + (WD(C11 * (V@2 + V@13)) +
             WD(C9  * (V@3 + V@12)) );    /* FIR_V_b */
v0c = v0b + (WD(C7  * (V@4 + V@11)) +
             WD(C5  * (V@5 + V@10)) );    /* FIR_V_c */
v0d = v0c + (WD(C3  * (V@6 + V@9 )) +
             WD(C1  * (V@7 + V@8 )) )<<1; /* FIR_V_d */
/* four delay lines */
u0 = u0d; u1 = U@7; v0 = v0d; v1 = V@7; y0 = Y1@8; y1 = Y0@7;
/* two 3 by 3 matrix multiplications */
v0x =    v0 - 128;                        /* MAT_0_a */
R0  =    y0 + (WD( 350 * v0x)>>8);        /* MAT_0_a */
g0a =    y0 + WD(-179 * v0x);             /* MAT_0_a */
u0x =    u0 - 128;                        /* MAT_0_b */
G0  =    g0a + (WD( -85 * u0x)>>8);       /* MAT_0_b */
B0  =    y0 + (WD( 443 * u0x)>>8);        /* MAT_0_b */
v1x =    v1 - 128;                        /* MAT_1_a */
R1  =    y1 + (WD( 350 * v1x)>>8);        /* MAT_1_a */
g1a =    y1 + WD(-179 * v1x);             /* MAT_1_a */
u1x =    u1 - 128;                        /* MAT_1_b */
G1  =    g1a + (WD( -85 * u1x)>>8);       /* MAT_1_b */
B1  =    y1 + (WD( 443 * u1x)>>8);        /* MAT_1_b */
end;
```

**Figure 6** SILAGE description for YUV-to-RGB conversion.

is assumed to be 27 MHz for our 1.2 $\mu$m CMOS library, 12 initial clusters can be identified for our demonstrator. This has been achieved using a simple clustering strategy, mainly based on the desired number of clusters (currently user-defined) and on their similarity in terms of operation and dependency types. In the SILAGE description in figure 6, these clusters are indicated at the right-hand side: four expressions in both FIR filters (FIR_xxx) and four expression groups in the matrices (MAT_xxx). Note that the delayed signal assignments (indicated with @) are ignored at this stage, since they are dealt with later, during the low-level mapping [8, 18].

## *ASU assignment*

This step involves allocation and assignment of clusters to ASUs. Compatibility measures between clusters are the objective function here, again within the imposed global timing- and memory-related constraints [3]. The assignment problem is formulated as a graph partitioning problem, in which the most similar clusters are assigned to the same partition. Similarity of clusters is measured by means of a cluster distance, which is defined as the difference of the hardware costs of the two clusters and the hardware costs of the set of resources which can execute both clusters: $D(a, b) = 2 \times C_{ab} - C_a - C_b$. Both the internal compatibility cost, i.e., the similarity expressed as hardware sharing overhead, and the embedding cost, i.e., an estimate of inter-ASU interconnect cost, are included. The graph partitioning problem is translated into a mixed ILP formulation, which also allows for an automatic distribution of the cycle budget over the hierarchical blocks in case of control flow hierarchy.

For the YUV-RGB application, there are 12 clusters that have to be executed in four clock cycles due to HSF requirement of 27 MHz / 6.75 MHz. This means that the clusters will have to be assigned to one of three ASUs. The internal compatibility measure shows a high degree of similarity for the eight clusters of the FIR filters ($D = 0.5 \ldots 1.0\text{mm}^2$) and for the clusters of the matrix ($D = 0.6 \ldots 0.9\text{mm}^2$), as indicated by the cluster distance entries without brackets in table 2. The FIR clusters are very different from the matrix clusters ($D = 2.2 \cdots 2.9\text{mm}^2$). This will favor an assignment of the matrix clusters to one and the same ASU. However, the embedding compatibility is higher for the clusters that belong to the same FIR filter because there are intercluster data dependencies, as indicated by the cluster distance entries within the brackets in table 2. Hence, each filter is executed on a separate ASU. The required CPU time amounts to 112 seconds, plus about five minutes for a general-purpose ILP package on a DecStation 5000. The resulting assignment is given in table 3.

| Cluster | FIR_U_a | FIR_U_b ... | FIR_V_a | FIR_V_b ... | MAT_0_a | MAT_0_b ... |
|---|---|---|---|---|---|---|
| FIR_U_a | 0.0 (0.0) | 1.0 (1.4) | 0.0 (4.2) | 1.0 (5.2) | 2.5 (6.3) | 2.9 (7.1) |
| FIR_U_b | — | 0.0 (0.0) | 1.0 (5.2) | 0.0 (4.2) | 2.4 (6.2) | 2.6 (6.8) |
| ... | | | | | | |
| FIR_V_a | — | — | 0.0 (0.0) | 1.0 (1.4) | 2.5 (6.3) | 2.9 (7.1) |
| FIR_V_b | — | — | — | 0.0 (0.0) | 2.4 (6.2) | 2.6 (6.8) |
| ... | | | | | | |
| MAT_0_a | — | — | — | — | 0.0 (0.0) | 0.9 (4.2) |
| MAT_0_b | — | — | — | — | — | 0.0 (0.0) |
| ... | | | | | | |

**Table 2** Cluster distances for the YUV-to-RGB conversion. The figures within brackets take embedding compatibility into account.

| ASU instance | Assigned clusters |
|---|---|
| ASU_A | FIR_U_a, FIR_U_b, FIR_U_c, FIR_U_d |
| ASU_B | FIR_V_a, FIR_V_b, FIR_V_c, FIR_V_d |
| ASU_C | MAT_0_a, MAT_0_b, MAT_1_a, MAT_1_b |

**Table 3** The cluster assignment for the YUV-to-RGB conversion.

## *ASU definition*

The ASU composition can now be determined, with the area cost in terms of abstract building blocks (such as adders) as the objective function. This offers early feedback on the quality of the clustering and the ASU assignment. The constraints are those imposed by the ASU assignment, and the I/O constraints imposed, for example, by memory management. The output includes the assignment of operations to building blocks. This involves a resource-sharing problem [18, 27, 4], deciding on the selection of the suitable FU types for each operation, the allocation of the desired number of FUs per type, a decision on the assignment of operations to FUs, and the interconnection of the FUs by means of a programmable switching network.

We have presented a fast ASU definition technique based on pairwise merging [4]: iterating over the set of clusters assigned to an ASU, merging two clusters in each iteration step. The merging is modeled as a bipartite matching problem. A recent version of this technique, which includes more accurate modeling of interconnect cost, leads to the ASU structures in figure 7 within a CPU time of 41 seconds. This result exhibits a very limited hardware sharing overhead, and almost all building blocks are occupied all of the time. The combined

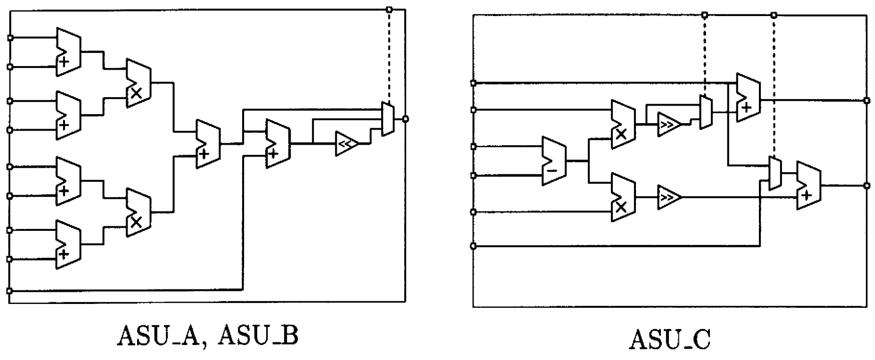


**Figure 7** Synthesised ASU structures for the YUV-to-RGB conversion.

active area occupied by these ASUs is only 13.3 mm$^2$ for a 1.2 $\mu$m CMOS library. Note that the areas occupied by memory, routing, and controller are not incorporated in this figure, since they result from other design stages in the trajectory [18]. This solution is also equivalent with the one found manually.

## 3 Evaluation of the result and design iteration

In order to evaluate the quality and feasibility of this allocation and assignment, two more subtasks are performed. These provide important feedback to the designer, and they can initiate design iterations:

- Pipeline balancing: the ASU compositions in terms of abstract building blocks are retimed, and selection of implementations of the building blocks is performed to check whether the proposed clock period can be achieved. If desired, a balancing of the pipelines can be performed to reduce the hardware cost [18].

- Cluster scheduling: this scheduling step is mainly used to evaluate the memory and interconnect cost, and to check the feasibility of the result proposed by all other steps. High-level scheduling models can be used to increase the efficiency.

  For the YUV-RGB demonstrator, it is possible to schedule the 12 available clusters on the three ASUs within the four available clock cycles. This means a functional hardware use of 100%. This is feasible by using an ILP-based scheduling technique, which can deal with very tight schedules in a close-to-optimal way. This is in contrast with the scheduling techniques for the highly multiplexed processors in chapters 8 and 9.

The final ASU architecture has very little programmability overhead and is optimal for the given throughput requirement and the available library of hardware operators. Subsequent design stages are needed in CATHEDRAL to arrive at the actual layout [8].

## 4 CONCLUSION

In this chapter, several important contributions have been described in the context of architectural synthesis for irregular high throughput applications. These are mainly situated in the domain of memory management techniques to arrive at area-efficient memory organizations and data-path mapping techniques to come up with customized data-path configurations that meet the stringent throughput requirements.

The power of the approach has been demonstrated on two realistic real-time signal processing applications, namely an auto-correlation kernel in a CD interpolator and a YUV-RGB video format converter. The results for both test vehicles are partial but nevertheless very promising. This has been achieved by adopting a very tuned synthesis script and dedicated synthesis techniques targeted to the CATHEDRAL-3 application domain. These differ considerably compared to the approaches proposed in chapters 8 and 9, which are oriented to different target domains. There are also significant differences with other techniques in the literature [7, 9, 13, 21, 25].

## REFERENCES

[1] F. Catthoor and H. De Man. Application-specific architectural methodologies for high-throughput digital signal and image processing. *IEEE Trans. on Acoustics, Speech and Signal Processing*, 37, number 2, pages 176–192, Feb 1990.

[2] F. Franssen, M. van Swaaij, F. Catthoor, and H. De Man. Modeling piecewise linear and data dependent signal indexing for multi-dimensional signal processing. In *Proc. 6th Int. Workshop on High-Level Synthesis*, Laguna Beach CA, Nov 1992.

[3] W. Geurts, F. Catthoor, and H. De Man. Time constrained allocation and assignment techniques for high throughput signal processing. In *Proc. 29th ACM/IEEE Design Automation Conf.*, Anaheim CA, pages 124–127, Jun 1992.

[4] W. Geurts, F. Catthoor, H. De Man. Heuristic techniques for the synthesis of complex functional units. In *Proc. of 4th European Design Automation Conf.*, Paris, France, Feb 1993.

[5] G. Goossens, *Optimization techniques for automated synthesis of application-specific signal-processing architectures.* PhD thesis, ESAT, K. U. Leuven, Belgium, Jun 1989.

[6] P. N. Hilfinger, J. Rabaey, D. Genin, C. Scheers, and H. De Man. DSP specification using the Silage language. In *Proc. Int. Conf. on Acoustics, Speech and Signal Processing*, Albuquerque, NM, pages 1057–1060, April 1990.

[7] K. S. Hwang, A. Casavant, C-T. Chang, and M.d'Abreu. Scheduling and hardware sharing in pipelined data-paths. In *Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara CA, pages 24–27, Nov 1989.

[8] D. Lanneer, S. Note, F. Depuydt, M. Pauwels, F. Catthoor, G. Goossens, and H. De Man. Architectural synthesis for medium and high throughput signal processing with the new CATHEDRAL environment. In R. Camposano and W. Wolf, editors, *Trends in high-level synthesis*, Kluwer, Boston, 1991.

[9] P. Lippens, J. van Meerbergen, A. van der Werf, W. Verhaegh, B. McSweeney, J. Huisken, and O. McArdle. PHIDEO: a silicon compiler for high speed algorithms. In *Proc. European Design Autom. Conf.*, Amsterdam, The Netherlands, pages 436–441, Feb 1991.

[10] D. B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24, number 1, pages 121–145, 1977.

[11] D. Mallon and P. Denyer. New approach to pipeline optimization. In *Proc. 1st ACM/IEEE Europ. Design Automation Conf.*, Glasgow, Scotland, pages 83–88, Apr 1990.

[12] T. H. Matheiss, D. S. Rubin. A survey and comparison of methods for finding all vertices of convex polyhedral sets. *Mathematics of Operations Research*, 5, number 2, pages 167–185, 1980.

[13] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proc. of the IEEE*, 78, number 2, pages 301–318, Feb 1990.

[14] D. Moldovan. Advis: a software package for the design of systolic arrays. In *Proc. IEEE Int. Conf. on Computer Design*, Port Chester NY, pages 158–164, Oct 1984.

[15] M. Moonen, P. Van Dooren, J. Vandewalle. SVD-updating for tracking slowly time-varying systems. Advanced algorithms and architectures for signal processing IV, *Proc. SPIE conf.*, Volume 1152, San Diego CA, Nov 1989.

[16] G. L. Nemhauser and L. A. Wolsey. Integer and combinatorial optimization. Wiley, New York, 1988.

[17] A. Nicolau, Loop quantization: a generalized loop unwinding technique. *Journal of Parallel and Distributed Computing*, 5, pages 568–586, 1988.

[18] S. Note, W. Geurts, F. Catthoor, and H. De Man. Cathedral III: Architecture driven high-level synthesis for high throughput DSP applications. In *Proc. 28th ACM/IEEE Design Automation Conf.*, San Francisco CA, pages 597–602, Jun 1991.

[19] D. A. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29, number 12, pages 1184–1201, 1986.

[20] K. Parhi. Algorithmic transformation techniques for concurrent processors. *Proc. of the IEEE*, 77, number 12, pages 1879–1895, Dec 1989.

[21] N. Park and A. C. Parker. Sehwa, a software package for synthesis of pipelines from behavioral specifications. *IEEE Trans. on Comp. aided Design*, CAD-7, number 3, pages 356–370, Mar 1988.

[22] N. Park and F. J. Kurdahi. Module assignment and interconnect sharing in register-transfer synthesis of pipelined data paths. In *Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara CA, pages 16–19, Nov 1989.

[23] C. Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on the architecture design. *IEEE Trans. on Computers*, 37, number 8, pages 991–1004, Aug 1988.

[24] P. Quinton and Y. Robert, editors. *Algorithms and parallel VLSI architectures II.* Elsevier, Amsterdam, 1992.

[25] J. Rabaey and M. Potkonjak. Resource-driven synthesis in the HYPER environment. In *Proc. IEEE Int. Symp. on Circuits and Systems*, New Orleans, pages 2592–2595, May 1990.

[26] A. Schrijver. *Theory of linear and integer linear programming.* Wiley, 1986.

[27] A. van der Werf, E. Aerts, M. Peek, J. van Meerbergen, P. Lippens, and W. Verhaegh. Area optimization of multi-function processing units. In *Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara CA, pages 292–299, Nov 1992.

[28] J. Vanhoof, I. Bolsens, and H. De Man. Compiling multi-dimensional data streams into distributed DSP ASIC memory. In *Proc. IEEE Int. Conf. Comp. Aided Design*, Santa Clara CA, pages 272–275, Nov 1991.

[29] M. van Swaaij, J. Rosseel, F. Catthoor, and H. De Man. High-level synthesis of ASIC regular arrays for real-time signal processing systems. In *Proc. Int. Workshop on Algorithms and Parallel VLSI Architectures*, Pont-a-Mousson, France, June 1990.

[30] M. van Swaaij, F. Franssen, F. Catthoor, and H. De Man. Modeling data and control flow for high-level memory management. In *Proc. 3rd ACM/IEEE Europ. Design Automation Conf.*, Brussels, Belgium, March 1992.

[31] M. van Swaaij, F. Franssen, F. Catthoor, and H. De Man. Automating high-level control flow transformations for DSP memory management. In *Proc. IEEE workshop on VLSI signal processing*, Napa Valley CA, Oct 1992.

[32] M. van Swaaij, F. Franssen, F. Catthoor, and H. De Man. High-level modeling of data and control flow for signal processing systems. In M. Bayoumi, editor, *Design Methodologies for VLSI DSP Architectures and Applications*, Kluwer, Boston, 1992.

[33] I. Verbauwhede, F. Catthoor, J. Vandewalle, and H. De Man. Background memory management for the synthesis of algebraic algorithms on multi-processor DSP chips. In *Proc. VLSI'89, Int. Conf. on VLSI*, Munich, Germany, pages 209–218, Aug 1989.

[34] I. Verbauwhede, F. Catthoor, J. Vandewalle, and H. De Man. In-place memory management of algebraic algorithms on application-specific ICs. *Journal of VLSI signal processing*, 3, pages 193–200, 1991.

# 8

# AUTOMATIC SYNTHESIS FOR MECHATRONIC APPLICATIONS

**Peter Pöchmüller, Norbert Wehn, Manfred Glesner**

*Darmstadt University of Technology*

## ABSTRACT

This mechatronic applications chapter presents the application of high-level synthesis techniques in connection with a rapid prototyping environment. This methodology is part of a design approach supporting the development of embedded information-processing units in complex mechatronic systems. In the following, a synthesis environment will be presented that automatically maps modules of a mechatronic system to be implemented as ASICs onto a rapid prototyping board.

## 1 INTRODUCTION

Traditional high level synthesis approaches [1, 2] start with a behavioral specification of an algorithm and proceed to derive an application-specific integrated circuit (ASIC) solution realizing the intended behavior. In contrast to this, the intention of the approach presented in this chapter is not to derive a specific ASIC architecture, but to get early estimations of overall system performance if certain components will be realized as ASICs and to derive a first prototype that permits real-time system simulations. In order to meet those goals during early design phases, we use an ASIC emulator board which is part of a larger design framework supporting the development of complex mechatronic systems. The framework supports a wide spectrum of subcomponent realizations, ranging from software implementations over digital signal processor (DSP) solutions to ASIC emulation in a heterogeneous multiprocessor environment.

## 2 SYSTEM OVERVIEW

The intention of this section is to give a brief overview on the whole framework, since the remainder of this chapter will focus exclusively on a single aspect, namely the automatic synthesis of a behavioral specification down to the ASIC emulator board. If the emulation demonstrates that an overall performance increase will be obtained through the realization of corresponding system submodules by an ASIC, other synthesis tools (such as CATHEDRAL or AMICAL, described in chapters 7 and 9) have to be used to derive a final chip solution replacing the emulator.

### 1 Context

The overall system performance of machine tools, vehicles, and aircraft can be improved significantly through the application of embedded information technology with intelligent software for the realization of core system components. Mechanical systems with embedded information-processing units are usually referred to as *mechatronic systems*. *Mechatronics* (*mecha*nics and elec*tronics*) can be described as an emerging engineering discipline dealing with the design, manufacturing, and operation of machines capable of intelligent behavior [3]. The design and implementation of corresponding systems is a highly complex task, involving problems like hardware/software tradeoff, system partitioning, final realization of components through microprocessors or ASICs, etc. Behavioral simulation is one approach to support such early design decisions; however, in this case the mechanical subsystem has to be modeled appropriately. The main problem of that approach is to determine whether false system response is due to an error in the control algorithm or in the model of the mechanical subsystem itself. A promising alternative to that approach is the application of hardware-in-the-loop simulation with real-time capabilities. In that case, the hardware simulator can be considered as a prototype implementation that can be attached directly to the mechanical process; thus, no model of the process is required. The prototype has the same functionality as the final implementation, only differing in physical dimension, power consumption, etc. One main characteristic of such a prototype is that changes can be implemented quickly; thus, the prototype allows to validate design decisions in early design phases. This strategy is called *rapid prototyping* [4].

Figure 1 gives an overview of the whole rapid prototyping system design framework [5]. The overall system specification is done in a language called RPL (rapid prototyping language). The first design step is the partitioning of this

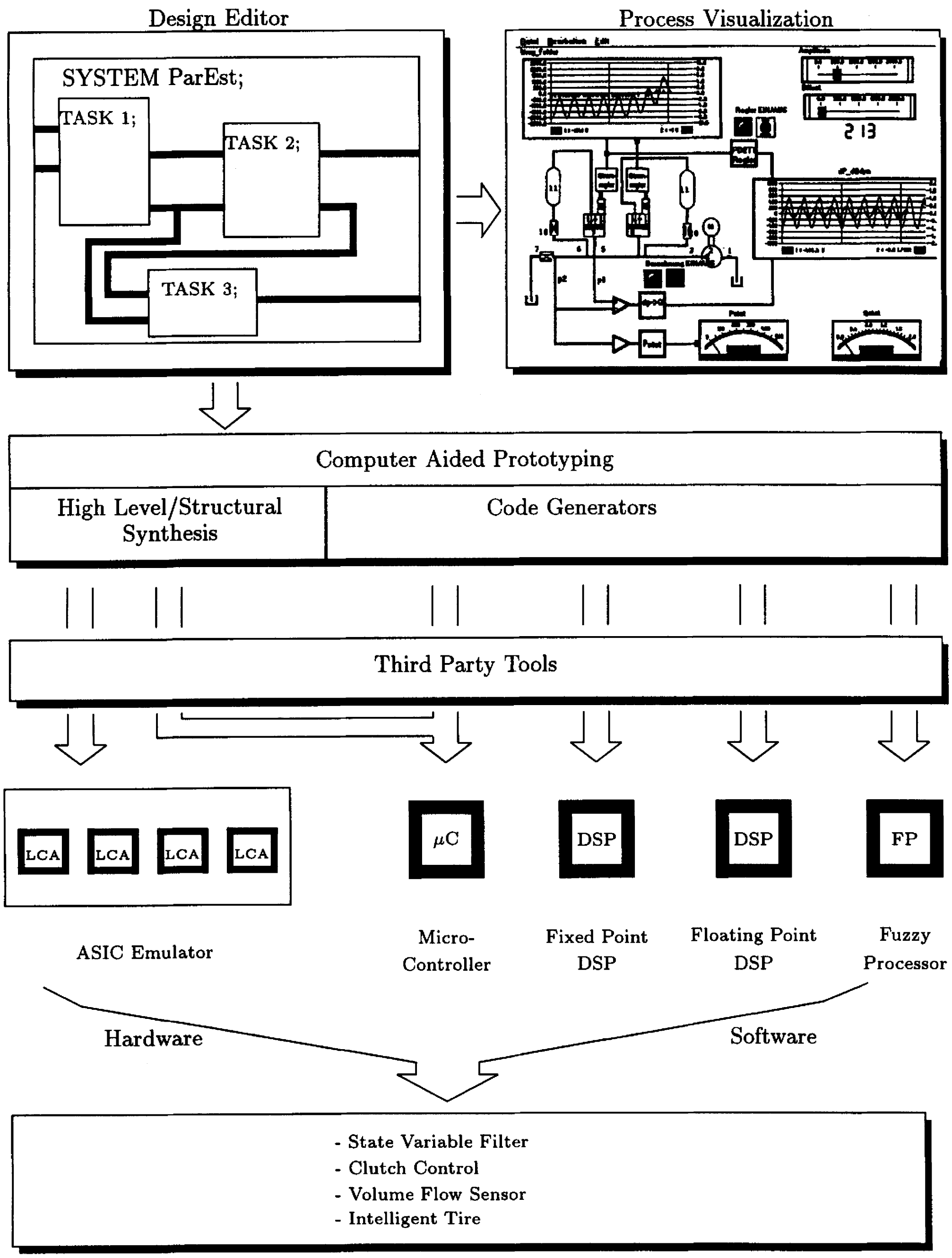


**Figure 1** Rapid prototyping system design framework.

specification into different tasks. Through a set of compiler/third party tools, a single task can be mapped to software solutions (Pascal, C) running on a host computer, to fuzzy processor software, to firmware for standard signal processors/microcontrollers, or onto an ASIC emulator. In the remainder of this chapter, emphasis will be on the ASIC emulator board. Therefore, it will be assumed that partitioning has already been performed, and a single task (process) is to be realized through the ASIC emulator.

## 2 ASIC emulator synthesis environment

As already stated above, the synthesis path toward an ASIC emulator requires as input specification a HARDWAREC description of a single task. This specification will be provided within the rapid prototyping framework depicted in figure 1. HARDWAREC as used in this system is not equivalent to Stanford HARDWAREC [6] which was found to be too restrictive to be used directly for our mechatronic applications. This is mainly due to the fact that Stanford HARDWAREC does not support complex data types and arrays of them (large integer, fix, etc.). On the other hand, many features of Stanford HARDWAREC are not required in this context, since only single tasks (including host communication) have to be specified, and no structural components are required. Figure 2 gives an example specification of a differential heat release computation (DHRC) algorithm, which in the following will be used to illustrate all main synthesis steps. This algorithm realizes a simplified computation of the heat release within a combustion engine [9].

As a first step, the HARDWAREC compiler performs a profound data/control-flow analysis. A DFG optimizer is directly coupled to the compiler. After optimization, the data/control-flow graph will be stored as a combined single flow graph which can be considered as a data base. All behavioral synthesis transformations require input from this data base and will produce output in the graph format defined in chapter 2.

Behavioral synthesis comprises several transformations [7] of the optimized flow graph. The final result of all behavioral synthesis steps will again be a flow graph, since hardly any structural information will be added through those transformations. The most important behavioral synthesis steps in our application domain are background memory management and loop folding. Both have strong effects on the final throughput. Other more common transformations are, for example, loop unfolding and tree balancing, which will not be explicitly mentioned in this chapter.

```
inout int<16>  host;
void main ()
{ int<16>  a0, a1, a2, h2, h3, h5, cv, dp, h6, h7, d, h9, bvl, i;
  int<16>  V[469], p[469], dv[469], bvl[469];
  int<32>  h1, h4, s1, s2, s, h8;
  get (a0, host); get (a1, host); get (a2, host);
  get (p[339], host); get(p[340], host);
  s = cv = h7 = h6 = h4 = dp = 0;
  for(i = 340; i < 468; i++)
  { get(p[i+1], host);
    d  = p[i+1] - p[i-1];
    dp = d/2;
    h1 = a0 * (int<32>)V[i];
    h2 = (int<16>)(h1 >> 14);
    h3 = a1 + h2;
    h4 = h3 * (int<32>)p[i];
    s2 = dp * (int<32>)V[i];
    h7 = (int<16>)(s2 >> 12);
    h5 = (int<16>)(h4 >> 11);
    cv = h5 + a2;
    s1 = p[i] * (int<32>)dv[i];
    h6 = (int<16>)(s1 >> 12);
    s  = h6 + h7;
    h8 = s * (int<32>)cv;
    h9 = (int<16>)(h8 >> 11);
    bvl[i] = h9 + h6;
    put(host, bvl[i]); }
}
synthesisconstraints {  /* CBB constraints */
  time     { cycletime = 200ns; setuptime = 10ns; holdtime  = 10ns; }
  hardware { 1 barrelshifter ((<<,90ns) (>>,90ns) bit<32> bit<16>);
             1 adder         ((+,140ns) bit<16> bit<16> bit<16>);
             1 multiplier    ((*,150ns) bit<16> bit<16> bit<32>); }
}
```

**Figure 2** Differential heat release computation algorithm (DHRC).

Structural synthesis comprises another group of transformations to be executed on the submitted flow graph. The main tasks to be solved are scheduling, allocation, and binding. The flow graph format cannot be maintained during those steps, since more and more structural information will be derived. The final output after structural synthesis will be all information necessary to customize the ASIC emulator board.

All the synthesis steps mentioned above can be executed either automatically or interactively. The programmed ASIC emulator can be inserted directly into the overall mechatronic system to facilitate real-time test runs. In the following, all main synthesis steps will be described in the same top-down sequence as they are implemented through the synthesis script. Throughout the rest of the chapter, the terms *ASIC emulator (board)*, *rapid prototyping board*, and *target architecture* will be used synonymously.

## 3 APPLICATION DOMAIN AND TARGET ARCHITECTURE

The application domain and ASIC emulator architecture strongly influence the overall synthesis script as well as the actual algorithms for solving single synthesis tasks. In order to achieve efficient solutions, our whole approach has been tuned toward real-time system applications as they are found in mechatronic systems. To support the understanding of the implemented approach/algorithms, this section will describe the intended application domain and ASIC emulator architecture in more detail.

### 1 Application domain

This synthesis approach is tuned toward real-time applications appearing in mechatronic systems, such as an adaptive shock absorber, a combustion engine controller, intelligent tire friction control, volume stream measurement, etc. A large number of microelectronic subcomponent designs have been studied [8]. As a result, it was found that the designs typically share the following common characteristics:

- The data/control-flow graphs are large and complex. Moreover, the applications require clock frequencies of 10–20 MHz, whereas the intermediate data throughput and sample frequencies remain far below 1 MHz. As a result of these factors, the hardware-sharing factor (HSF) is much larger than 1, resulting in highly multiplexed architectures.

| Benchmark | Nodes | Edges | Background memory | Mutually exclusive operations |
|---|---|---|---|---|
| State variable filter 1 | 307 | 598 | yes | no |
| State variable filter 2 | 521 | 945 | yes | no |
| DHRC | 122 | 211 | yes | no |
| Volume stream measurement | 99 | 188 | no | yes |
| Combustion engine control | 1047 | 1953 | yes | yes |

**Table 1** Benchmark set of mechatronic applications.

- Some applications require floating-point arithmetic (which is not supported by the emulator board and module library).
- There are only one- and two-dimensional data streams (vectors and matrices) of complex data types (large integer, fix). Typical vector sizes are 10–500 elements. Arrays with two dimensions have less than 10 entries in each dimension.
- Due to the presence of these arrays and vectors, a considerable part of the area of the final ASIC implementations (not the ASIC emulator) is occupied by memory.
- Single processes have to be realized, which only communicate with a flexible host and the mechanical process.
- There is a large dynamic range of values to be processed.

The target architecture described below provides a high degree of parallelism and is defined in such a way that the characteristics listed above are met. In contrast to this, the application domains as defined in chapter 7 (data-flow-dominated applications where a hardware-sharing factor closer to 1 implies a lowly multiplexed architecture handling complex multidimensional data streams) and chapter 9 (control-flow-dominated applications) differ in most regards. To develop a suited synthesis script and corresponding tools, an appropriate benchmark set was compiled, as shown in table 1. The number of nodes and edges refer to flow graphs after compilation without optimization. In order to permit a comparison with traditional high-level synthesis approaches [1, 2], the well-known fifth order elliptic filter benchmark has been compiled, too. With 68 nodes and 131 edges after compilation, the filter is a quite small

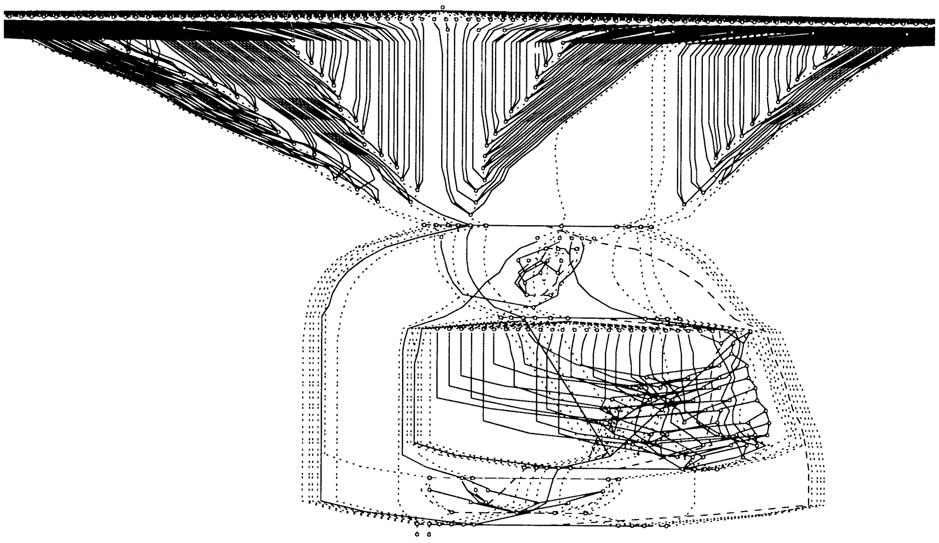


**Figure 3** Flow graph of state variable filter 2 after compilation and optimization.

synthesis example. In contrast to this, figure 3 shows the flow graph of the "state variable filter 2" benchmark algorithm (which has constant matrices) after compilation *and* optimization. The DHRC example of table 1 is small but nevertheless realistic; it will therefore be used throughout this chapter to illustrate the different synthesis steps. At this moment, there is no way to map the combustion engine control benchmark onto the emulator board since it involves floating point arithmetic. However, it is a realistic example that has been realized as an ASIC [9], and it is therefore a very good test vehicle to study effects of behavioral transformations (especially background memory management).

## 2 Target architecture

The principal target architecture of the emulator is given in figure 4. It comprises a host interface, four configurable building blocks (CBBs), and a process interface (8 analog input channels, 1 analog output channel, 16 digital I/O channels, and 2 programmable timers). Those blocks are connected via a programmable routing network, consisting of four busses. The fifth bus is used for debugging purposes only. All data communication busses are split into independent bus segments which can be connected via programmable switching

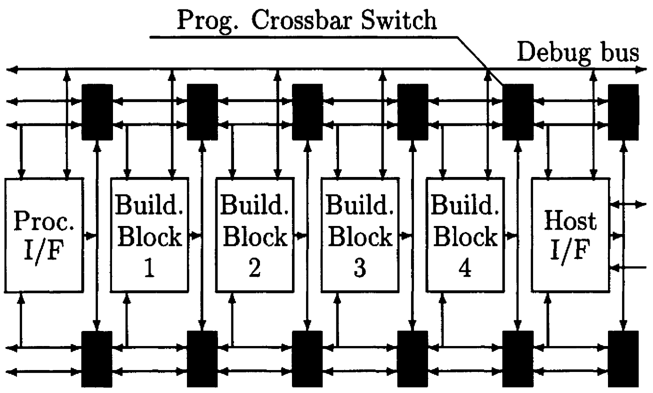


**Figure 4** ASIC emulator target architecture.

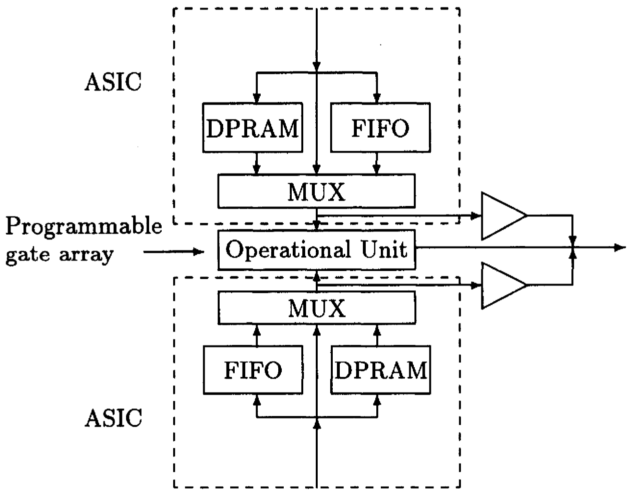


**Figure 5** The architecture of a configurable building block (CBB).

matrices. The matrices are full crossbar switches; each matrix can establish all possible connections between its inputs at any time. A CBB consists of two inputs, one output, and one operational unit (see figure 5). Operands are buffered in dual-port memories or FIFOs. The switching matrix and operand memories with their multiplexors are realized as ASICs in a 1.5$\mu$m CMOS technology. When the architecture of figure 4 is used as an ASIC emulator board, the operational units are implemented by means of field-programmable gate array (FPGA) devices (XILINX XC4005). A customization of the board is achieved by down-loading configuration files from a library. The global board controller includes a sequencer (realized by an FPGA). To minimize interconnection between controller and data-path, all micro-instructions are stored locally in an instruction memory on each CBB.

This architecture is justified by the following reasons:

- The use of FPGAs as operational units allows a large flexibility in customizing the ASIC emulator.
- Sufficiently high parallelism can be achieved (four CBBs).
- The highly multiplexed data-path architecture supports mapping of computationally intensive algorithms with a large hardware sharing factor.
- FIFOs as operand memories are well suited for vector/matrix operations.
- The number of active switching elements is a good estimation for the routing complexity of a corresponding ASIC implementation.

## 4 BEHAVIORAL SYNTHESIS TRANSFORMATIONS

A large number of behavioral synthesis transformations have been implemented in the system. However, this chapter will only focus on the more important and interesting transformations, especially background memory management and loop folding. More common behavioral synthesis steps, like loop unfolding or tree balancing, are not presented due to the limited space available. In this approach, compiler optimizations are also regarded as behavioral transformations. The different steps will be illustrated using the DHRC example of figure 2.

### 1 Compiler optimizations

The first synthesis step is a graph optimization, as is familiar in software compiler construction. Figure 6 shows the flow graph of the DHRC example directly after compilation. The goal of the optimization process is to execute transformations on the flow graph which will improve final synthesis results or at least speed up synthesis through simpler graphs (without influencing the efficiency of the final solutions negatively). Those transformations do not include any structural synthesis steps. This means that all those graph transformations are regarded as optimizations where it is possible to predict an improvement

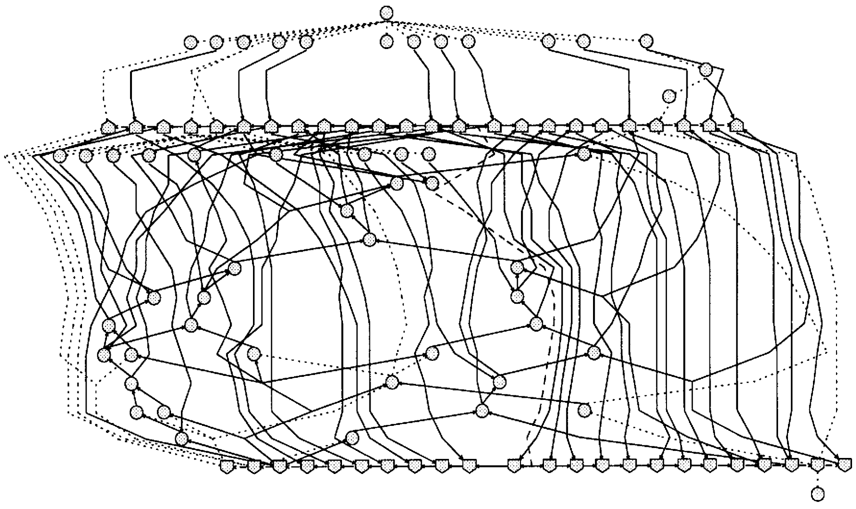


**Figure 6** Flow graph of unoptimized DHRC example directly after compilation.

exclusively based on flow graph information. Some typical tasks that can be solved are dead-code elimination, removal of redundant expressions, and constant propagation. Furthermore, the HARDWAREC compiler produces redundant edges/nodes in connection with variables. During parsing, it is not possible to know if there will be any other future reference to a written variable. Therefore, in the case of loops, variables will be led out for flow graph connections in future references. This is also the prime source of redundancy in figure 6. Figure 7 shows the same graph after optimization. This transformation reduced the numbers of nodes/edges from 123/210 to 82/131.

## 2 Background memory management

In this context, background memory is defined as storage for all data explicitly defined as arrays in the initial HARDWAREC specification. Foreground memory denotes all registers that are generated during structural synthesis to hold the values of single variables. After lifetime analysis, several variables with nonoverlapping lifetimes might share a single register.

The goals of background memory management (BMM) are to minimize background memory (arrays in the HARDWAREC specification) both in number and size as much as possible, and to simplify address generation taking into ac-

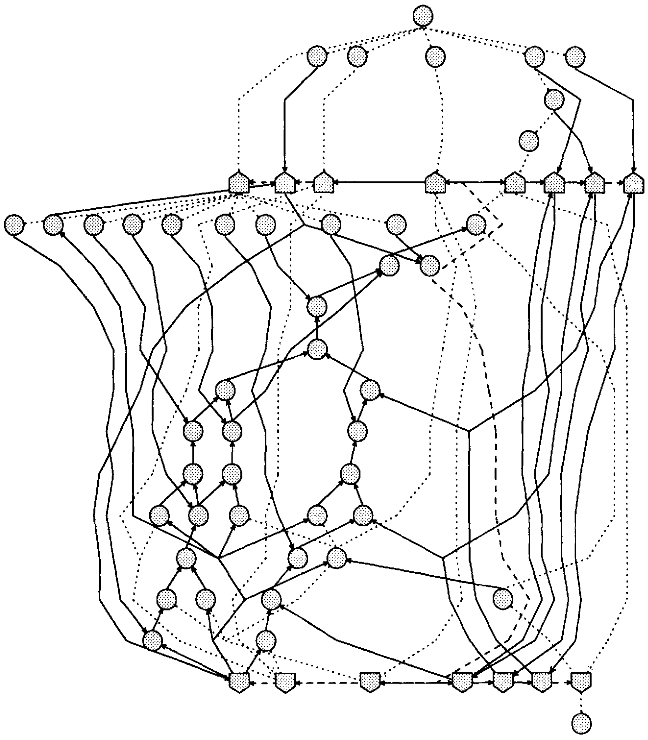


**Figure 7** Flow graph of optimized DHRC example.

count user-defined constraints. Background memory management has proved to be an important behavioral synthesis transformation for specifications with a considerable number of arrays and vectors [10]. In the investigated examples, it was so critical that without BMM, no final ASIC solution was possible. Therefore, BMM was selected to be the first behavioral synthesis transformation (after optimization) so that the design space is still as little restricted as possible. This is in agreement with research results on high-level memory management presented in chapter 7. However, our mechatronic applications motivate a different memory management approach due to their distinct characteristics. There are no multidimensional data with huge index spaces, but only vectors and two-dimensional matrices of complex types with a limited index space (considerably less than 100 in each dimension). Generally, there are no large off-chip memories, and efficient realizations require the size of the onchip background memories to be minimized. Furthermore, vector and matrix updating is already organized to produce compact arithmetic computations in the initial HARDWAREC specification (e.g., complete pressure and volume vectors are processed to produce a new resulting vector). Hence, there is less need for polyhedral analysis to improve the global loop structure (see chapter 7). Instead, the emphasis lies on I/O access management and memory size reduction based on data flow analysis. To meet these characteristics, the following background memory management steps have been defined and implemented.

### *I/O management*

In the intended application domain, ASIC subcomponents are generally used as interfaces between a flexible host computer and the mechanical process to be controlled. In most cases, host-to-ASIC communication is restricted to an occasional transfer of parameters, which can be provided by the host in any desired sequence. This transfer is not time-critical since the time spent in computations executed on those data is several orders of magnitude larger than the transfer time. Therefore, there is a choice of directing $n$ streams of data through $m$ physical ports. Furthermore, several data streams can be interleaved: for example, vectors $v_1 = (n_1, n_2, \ldots, n_k)$ and $v_2 = (m_1, m_2, \ldots, m_l)$ with $l = 2k$ can be read through one physical port in the sequence $n_1, m_1, m_2, \ldots, n_k, m_{l-1}, m_l$. This saves one port and might not delay data-dependent operations too much. Since the current version of the emulator supports only a single port to the host computer, I/O management has to decide how all data are to be optimally sequenced into the ASIC emulator board.

### *Replacement of accesses and arrays*

The removal of unnecessary arrays is one of the core processes during memory management, replacing many redundant memory write- and read-operations with data edges. This is equivalent to shifting background memory into foreground, since values on data edges might later be assigned to registers during structural synthesis if they are crossing clock cycle boundaries. Figure 8 shows an example for such a transformation taken from the combustion engine control benchmark. The value `Qb[i]/Qmax` is computed and assigned to the array `Su[i]`. The same value `Su[i]` is directly accessed in the next statement for further processing. Human programmers are frequently using arrays in this way to define data flow without the intention that `Qb[i]` and `Su[i]` have to be realized as explicit memories.

If there would not be any other access to the value `Su[i]` in the example above, then the complete update/retrieve-pair could be replaced through a single data edge, as depicted in figure 8. This means that the value transported on this data edge will be assigned to a register (foreground memory) during

```
for (i=0; i<128; i++)
{
  /* normalization */
  Su[i] = Qb[i]/Qmax;
  /* air number */
  r[i]  = Su[i] * ...;
}
```

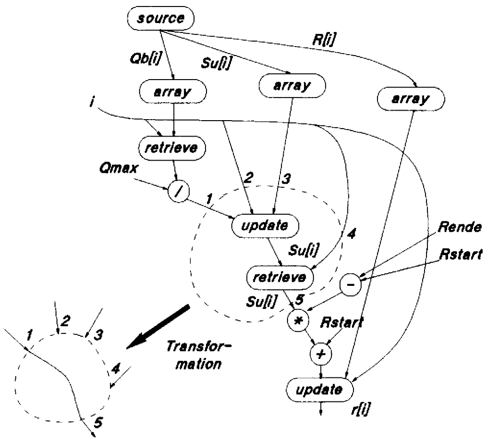


**Figure 8** Replacement of memory access operations. The *update/re-trieve*-node represents a *write/read*-operation (see chapter 2).

structural synthesis. The corresponding register will probably be shared with other variables, which is why such a pushing into foreground memory does not necessarily generate an extensive number of additional registers. Now, the index values (edges 2, 3, and 4 in figure 8) are not required any more, and they are subject to further optimizations since all involved computations can be eliminated. If there are no more update/retrieve operations, then even the whole array could be eliminated (pushed into foreground).

However, the replacement of update/retrieve-operations and eventually complete arrays is rarely as simple as depicted in the example. The comparison of update/retrieve-indexes is problematic, since the index $i$ generally does not originate from the same node for different update/retrieve operations (e.g., `a[i+1] = x[k-3] + 4`). This problem has been solved through tree matching algorithms which are used to compare flow graph structures that are generating the indexes. A new call of the optimizer will remove obsolete index computations that will further minimize the flow graph.

## *Compression, merging, reorientation, and type selection*

Frequently, only a small "window" of an array is used in a loop iteration to compute the next entry. A typical expression is the statement `a[i] = a[i-1] + 3*a[i-2]` located within a large loop with index $i$. If that array is not used for other computations, the whole structure can be pushed into foreground, requiring only three registers. Should the additional number of registers exceed a user-defined constraint, the array will be compressed to the window size if the addressing scheme does not become more complicated.

Array merging is the process of forming a new set of $n$ arrays out of $m$ previous arrays, where $n < m$. It can be very advantageous to merge arrays, especially when the lifetimes of the stored values are different (overwriting of old values). One critical factor in array merging is memory access conflicts. If too many arrays are merged into one array, and these arrays have data accesses at the same time, the overall algorithm execution can be prolonged. Nonoverlapping (in time) array operations are exploited during array merging. Another very critical factor in array merging is again address computation. If several arrays are merged into a single array and some of these arrays were already submitted to array compression, address computation can become quite complex and irregular. In such cases no merging will be executed.

All array indices are analyzed over time through a fast simulation to support the above-mentioned steps. This information will also be used for actual array-type selection. For the investigated applications, arrays could be realized through FIFOs or ring buffers in most cases. If possible, array entries will be moved to other locations to realize monotonous access orders supporting such efficient realizations. A fast and very simple list scheduler is used to estimate background memory access conflicts.

Background memory management can result in strong improvements of efficiency if the specified background memory is complex. The DHRC example is not much affected since the background memory is simple. Only the four arrays `V[469]`, `p[469]`, `dv[469]`, and `bvl[469]` are specified explicitly. `V` and `dv` represent static values (volume and differential volume over the engine crank angle) to be read once at the beginning of the algorithm. (This part is not included in the HARDWAREC specification of figure 2.) The pressure $p$ has to be read continuously for each working cycle. In this case, background memory management only eliminates the lower parts of the arrays that are never addressed (e.g., $0 \leq i \leq 339$ for `V`). Furthermore, the memory manager proposes to map `V` and `dv` onto the FIFOs of the ASIC emulator, since their values are addressed sequentially, whereas `p` and `bvl` have to be stored in the interface section.

As already mentioned earlier, the combustion engine control algorithm is an excellent test vehicle for behavioral transformations, especially since the background memory is both complex and intricate. The initial HARDWAREC specification uses 37 arrays of 128 float values each. After background memory management, the number of arrays could be reduced to 4 (90% reduction), the number of array write operations from 48 to 6 (87.5% reduction), and the number of array read operations from 123 to 26 (79% reduction). Furthermore, many index computations became obsolete, and a new call of the optimizer resulted in a reduction of overall flow graph nodes from 1,047 to 765 (29.9% reduction) and edges from 1,953 to 1,425 (27% reduction). It was also proposed that the remaining four arrays should be realized as FIFOs.

## 3 Loop folding

Loop folding is another behavioral synthesis transformation that proved to be crucial for the applications under investigation [12]. This is because a straightforward mapping of the graphs onto data-path modules results in relatively poor hardware utilization. Data dependences prohibit parallel execution in most cases, if parallelism via loop boundaries (iterations) is not taken into account. However, through loop folding, the potential parallelism of operations located in different loop iterations (software pipelining) can be exploited.

Most previously published approaches include loop folding into the scheduler [11]. This can be problematic for large examples, since schedulers are already overloaded with solving several crucial tasks simultaneously (operation chaining, multicycle operations, mutual exclusiveness, etc). Therefore, in this approach folding was defined as a purely behavioral graph partitioning transformation, not affecting the scheduler directly. The goal of folding is to transform, for given hardware resources, the flow graph of a loop in such a way that throughput will be maximized in the consecutive structural synthesis step.

The partitioning is based on the mobility $\beta(op)$ of operations, which is limited through ASAP- and ALAP-scheduling. A probability *p(op,t)* can be associated with each operation, indicating the probability that operation *op* will finally be found in clock cycle *t*. If an operation *op* has, for example, the mobility $\beta(op) = 3$ then it is assumed that it will be finally scheduled with a probability of $p(op,t) = \frac{1}{3}$ into clock cycle $t$, $ASAP(op) \leq t \leq ALAP(op)$. To estimate the distribution of all hardware resources, the probability of a single operation type can be summed up within a clock cycle to derive the so-called distribution graph $D$:

$$D(op\_type, t) = \sum_{op \in op\_type} p(op, t)$$

where *op_type* denotes a single operation type, such as *adder*. This approach is very similar to the well-known force-directed scheduling [13]. On the left side, figure 9 shows the distribution graph of the DHRC benchmark's mul-

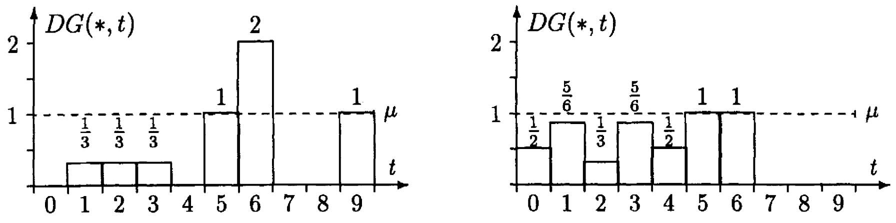


**Figure 9** Distribution graphs of multiplier resources before (left) and after loop folding.

tiplier resources (the corresponding flow graph is in figure 7). As depicted by the distribution graph, multiplier utilization is not as smooth as would be desired. The graph partitioning is realized through a shifting of single operations into the next loop iteration. The shifting is based on a cost change $\Delta F = C_{after} - C_{before}$. Costs are computed before and after shifting of an operation, and as long as cost changes are negative, operations will be shifted into the next iteration. The actual cost function computations are based on the distribution graphs, and the more the distribution graphs are deviating from the average value given through hardware resources, the higher the involved cost values become. Through this approach, the loop body of figure 7 will be partitioned into two iterations, resulting in a final flow graph as shown in figure 10. Obviously the critical path length has been decreased (by four operations), and the multiplier distribution graph on the right side of figure 9 shows a much more efficient utilization, indicating an increased data throughput.

## 5 STRUCTURAL SYNTHESIS

After behavioral synthesis, a transformed and optimized flow graph is available, which still does not include any explicit structural information (except module-type proposals for background memory and constraints specified in the input description). The goal of the consecutive structural synthesis process is to transform this flow graph information into an actual implementation that can be executed on the ASIC emulator board. The main steps to be carried out are scheduling, allocation, and binding. The results will again be illustrated by means of the DHRC-benchmark.

### 1 Scheduling

Scheduling receives the optimized data/control flow graph, a fixed CBB allocation, and a maximum timing constraint as input. The goal of scheduling

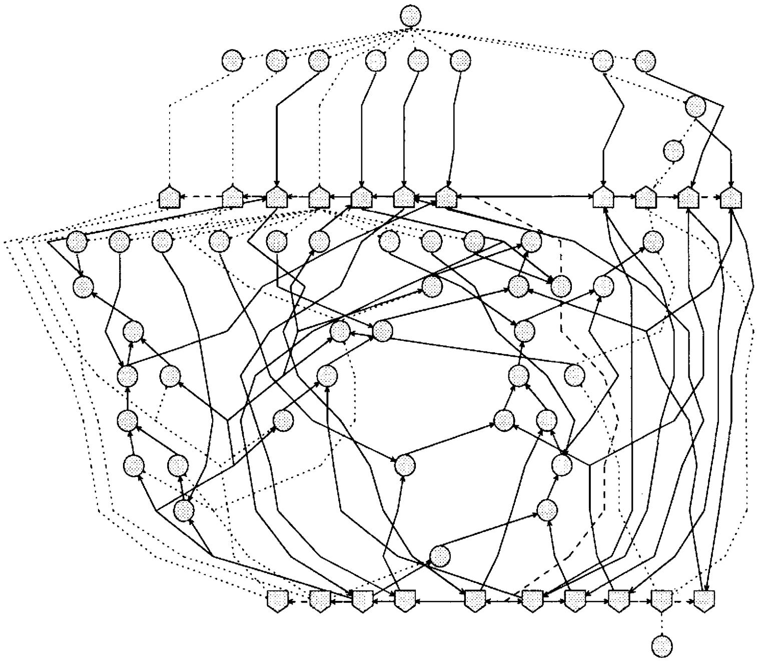


**Figure 10** Flow graph of DHRC example after loop folding.

in the presented approach is to assign flow graph operations to time slots, so that the potential parallelism of the algorithm is exploited as much as possible by taking into account timing and hardware constraints. This is essentially the classic microcode scheduling problem. The hardware constraints are the fixed CBB allocation (as specified in the HARDWAREC description) and the restrictions imposed by the emulator board. The applications are data-flow-dominated with medium throughput. Thus, heavy time multiplexing on the few CBBs results. In order to run the board with a moderate frequency (since CBBs are realized by FPGAs), operation chaining is not allowed. On the other hand, CBB operations can require more than one clock cycle. The architecture of the emulator board does not support pipelining. To take into account these requirements and restrictions, a list scheduling technique was preferred.

Due to the constructive nature of list scheduling, hardware constraints can be checked easily whenever operations of the ready-list (this list keeps all operations that can be scheduled in the actual control step, i.e., all their predecessors are already scheduled) are mapped into the current cycle. In order to

```
a1=c1+c2;           /* A1 */
a2=a1+c3;           /* A2 */
if (condition)
  { a3=a1+c4;       /* A3 */
    a4=a1+a3; }     /* A4 */
else a5=a1+a2;      /* A5 */
```

**Figure 11** An example for mutual exclusiveness.

get a global view, the priority function (which determines the operations to be deferred if hardware constraints are violated) of the scheduler must reflect the consequences on all operations not yet scheduled. Instead of using a static priority function such as the mobility of operations, a dynamic technique is used that calculates lower bounds on the hardware necessary to execute the operations not yet scheduled within the timing constraint [15]. This guarantees a global view on decisions to be made in the actual control step. Some relaxation techniques are used in order to have a moderate complexity of the calculation of the lower bounds. Note that these bounds must be calculated for each candidate of the ready-list.

Many scheduling techniques are restricted to basic blocks, i.e., the schedulers are not able to schedule across the border of a loop body or a conditional blocks. There are only a few exceptions, for example, path-based scheduling [16]. However, path-based scheduling uses operation chaining extensively, which makes it ill-suited in this case. Since loop folding is done prior to scheduling as a behavioral transformation, scheduling beyond basic blocks is only required for conditional blocks. A vector concept [17] has been implemented, which dynamically calculates a condition vector for each operation. This vector reflects the condition under which an operation is executed. The vectors permit calculating the exact minimum number of CBBs in each control step, taking into account all types of mutual exclusiveness, both implicit and explicit.

The example in figure 11 illustrates this concept. Assume that each addition needs one cycle and that only one adder-CBB is available. At first glance it seems that four cycles are necessary to execute this piece of code, since there are data dependencies (A1, A2) and (A3, A4). However, a more detailed analysis of data dependencies shows that a solution with three cycles is possible: since the results of A2 and A3 are never used simultaneously, the two operations are mutually exclusive. The same is true for A4 and A5. Under the assump-

```
cycle 1:  A1
cycle 2:  (A2,A3)
cycle 3:  (A4,A5)
```

**Figure 12** Scheduling result of example given above.

tion that the value of *condition* is already known, the schedule of figure 12 results. Obviously the scheduler is able to make decisions beyond basic blocks; thus, throughput is further increased. The result of this phase is a scheduled microcode, the vector information, and lifetime values which are fed into the binder.

## 2 Binding

In addition to the scheduling results, the binding phase receives loop information and information on arrays to be mapped into the CBB-FIFOs. The goal of binding is to map operations onto CBBs, transfers onto bus-segments/switching matrices, and assignment of variables to actual memory locations. The bottleneck in the binding phase is the programmable routing network, which imposes hard restrictions. Binding is separated into five steps:

- First, operand collisions are resolved. Operand collisions emerge if two CBB results are written into the same operand memory at the same time. If collisions cannot be resolved by exchanging operands (through exploitation of operation commutativity), it is guaranteed that the corresponding operations are mapped onto different CBBs.
- In the next phase, the scheduled flow graph is partitioned, i.e., all operations belonging to one partition are mapped onto the same CBB. In this step, the vector information of the scheduler is used to take into account mutually exclusive operations.
- After partitioning, interconnection binding is carried out. This process tries to avoid interconnection binding deadlocks. The goal is to transfer CBB results on one horizontal bus, i.e., the corresponding variable should be used at the same operand position by all successor operations. In addition, the number of transfers on each bus system is balanced in every control step. A heuristic has been developed, which is similar to the

Kernighan-Lin algorithm [18]. This heuristic exploits operation commutativity in order to align operands.

- Next, a linear CBB assignment is done, i.e., the CBBs are assigned to the different slots of the ASIC emulator board. This assignment is crucial for reducing the total interconnection cost. Since the number of CBBs is limited to four, an enumeration process is acceptable. The host- and process-interface CBBs are fixed in their positions; thus, only 24 permutations result. For each permutation, the costs of the switching matrix and multiplexors are calculated. In a post-optimization phase, switching matrix costs are reduced by moving groups of transfers between cliques. Many of these permutations fail in the target architecture, i.e., the request for switching matrices exceeds the hardware of the ASIC emulator. The feasible permutation with the lowest cost function is selected as final assignment.

- Finally, *memory binding* is done, i.e., the variables and arrays are mapped onto the DP-RAMs and FIFOs.

Partitioning, linear CBB assignment, and memory binding are modeled as weight-directed clique partitioning problems. An improved version of the Tseng heuristic [19] has been implemented to solve them. The edge weights of the compatibility graphs are tuned towards the target architecture as much as possible.

When binding is finished, all information is available for programming CBBs and state machines. The ASIC emulator can be customized and control code can be generated. In order to keep the controller complexity as low as possible, all loops are unrolled at the end of binding. This procedure is justifiable, since the size of the ASIC emulator has no relation to the final ASIC implementation. Note that the goal is to get some performance indications with this board.

## 6 RESULTS

The presented approach has been successfully tested on some high-level synthesis benchmarks. Although the fifth order elliptic filter benchmark is not representative for the investigated application domain, it was submitted to the synthesis system, mainly for testing the synthesis flow. The example could be mapped successfully onto the emulator board, and the scheduling results are comparable to the best reported in literature.

| Allocation | |
|---|---|
| CBBs | 1 ALU, 1 multiplier, 1 shifter |
| Interface | `p[i], bvl[i]` |

| Results | |
|---|---|
| $\#SM_{static}$ | 4 |
| $\#SM_{dynamic}$ | 8 |
| Bus segments | 14 |
| Registers | $16 + 2 \cdot 128$ |
| Cycles | 1,294 |
| Evaluation time | 0.258 ms |
| Multiplier occupation | 100% |
| ALU occupation | 50% |
| Shifter occupation | 60% |
| Parallelism (2 CBBs) | 90% |
| Parallelism (3 CBBs) | 20% |

**Table 2** Results of DHRC example after mapping onto the ASIC emulator board. $SM_{static}$ are static connections which do not change during runtime. $SM_{dynamic}$ are dynamic connections.

Table 2 shows the results achieved for the DHRC benchmark. The total execution time on the emulator board is 1,294 cycles. This throughput could only be achieved for the folded graph. Without folding, the algorithm required 1,810 cycles, since hardware utilization was very poor.

Both versions of the state variable filter as well as the volume stream measurement benchmark could be mapped onto the board.

As already mentioned, the combustion engine benchmark could only be submitted for testing high-level synthesis transformations, which was very interesting since the background memory is complex and intricate. The results achieved had been very good. The initial specification comprised 37 arrays, which were finally reduced to 4, realized as FIFOs. This is comparable to a solution found by a skilled designer.

The volume stream measurement benchmark could also be mapped onto the board. Mutually exclusive operations are available in this example and were exploited by the scheduler.

## 7 CONCLUSION

In this chapter, the application of-high level synthesis techniques within a rapid prototyping environment for real-time medium-throughput systems has been presented. A synthesis scheme has been discussed, which supports the automatic mapping of single process algorithms specified on a behavioral level in HARDWAREC onto a highly multiplexed ASIC emulation board. New efficient techniques have been presented to solve the corresponding synthesis tasks. Examples demonstrate the efficiency of the approach.

## REFERENCES

[1] P. Michel, U. Lauther, and P. Duzy, editors. *The synthesis approach to digital system design.* Kluwer Academic Publishers, 1992.

[2] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-level synthesis.* Kluwer Academic Publishers, 1992.

[3] G. Rzevski. Mechatronics at the Open University. In *Proc. of the 24th International Symposium on Automotive Technology and Automation (ISATA)*, Florence, pages 15–22, May 1991.

[4] M. Srivastava, J. Sim, and R. Brodersen. Hardware and software prototyping for application-specific real-time systems. In *Proc. of the 2nd Int. Workshop on Rapid System Prototyping*, Research Triangle Park, June 1991.

[5] H. Herpel, N. Wehn, and M. Glesner. RAMSES—a rapid prototyping environment for embedded control applications. In *Proc. of the Second Int. Workshop on Rapid System Prototyping*, Research Triangle Park, June, 1991.

[6] G. DeMicheli and D. Ku. HERCULES—a system for high-level synthesis. In *Proc. of 25th DAC*, Anaheim, pages 483–488, Jun 1988.

[7] M. Potkonjak and J. Rabaey. Optimizing resource utilization using transformations. In *Proc. of the ICCAD*, Santa Clara, pages 88–91, 1991.

[8] P. Windirsch, H. Herpel, A. Laudenbach, and M. Glesner. Application-specific microelectronics for mechatronic systems. In *Proc. of EURO-DAC'92*, Hamburg, pages 194–199, Sep 1992.

[9] A. Laudenbach and M. Glesner. VLSI system design for automotive control. *IEEE Journal of Solid-State Circuits*, JSSC-27, number 7, pages 1050–1056, Jul 1992.

[10] P. Pöchmüller and M. Glesner. Memory management as a high level synthesis transformation. In *ASIC 92*, Rochester, pages 166–169, Sep 1992.

[11] G. Goossens, J. Vandewalle, and H. De Man. Loop optimization in register-transfer scheduling of DSP systems. In *Proc. of 26th DAC*, Las Vegas, pages 862–831, Jun 1989.

[12] P. Pöchmüller and M. Glesner. Force directed loop folding. *ITG-Fachbericht 122*, VDE-Verlag, pages 135–146, Nov 1992.

[13] P. Paulin and J. Knight. Force-directed scheduling for the behavioural synthesis of ASICs. *IEEE Transactions on CAD of Integrated Circuits and Systems*, CAD-8, number 6, pages 661–679, Jun 1989.

[14] N. Wehn, H. Herpel, T. Hollstein, P. Pöchmüller, and M. Glesner. High-level synthesis in a rapid-prototype environment for mechatronic systems. In *Proc. of EURO-DAC'92*, Hamburg, pages 188–193, Sep 1992.

[15] M. Potkonjak and J. Rabaey. A scheduling and resource allocation algorithm for hierarchical signal flow graphs. In *Proc. of the 26th DAC*, Las Vegas, pages 7–12, Jun 1989.

[16] R. Camposano. Path-based scheduling for synthesis. *IEEE Transaction on CAD*, CAD-10, pages 85–93, Jan 1991.

[17] K. Wakabayshi and T. Yoshimura. A resource sharing and control synthesis method for conditional branches. In *Proc. of the ICCAD*, Santa Clara, pages 62–65, 1989.

[18] K. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graph. *Bell System Technical Journal*, 49, number 2, pages 291–307, Feb 1970.

[19] C. Tseng and D. P. Siewiorek. Automated synthesis of data paths in digital systems. In *IEEE Transactions on CAD*, CAD-5, number 3, pages 379–395, Jul 1986.

9

# SYNTHESIS FOR CONTROL-FLOW-DOMINATED MACHINES

**Kevin O'Brien, Inhag Park**
**Ahmed A. Jerraya, Bernard Courtois**

*TIM3/INPG, Grenoble*

## ABSTRACT

This chapter presents AMICAL, an advanced, high-level synthesis system targeted toward control-flow-dominated machines. It interfaces well with existing design environments and methodologies. Starting with a pure VHDL input, AMICAL produces a full specification for existing logic and RTL synthesis tools. It also allows the re-use of existing components within new designs (external library of macros). In addition, AMICAL uses a powerful scheduler and accepts a comprehensive VHDL subset (multiple waits, nested loops, exits, procedures, functions, etc).

The combination of automatic and manual synthesis allows a quick and broad exploration of the design space in real time. The response time of AMICAL is very short, making it a genuine interactive system. Several large examples have already been used for AMICAL evaluation with excellent results, including a telephone answering machine controller, which is used as an illustrating example in this chapter.

## 1 INTRODUCTION

After the success and the widespread acceptance of logic and register-transfer-level (RTL) synthesis, the next step is micro-architecture synthesis, commonly called behavioral synthesis or high-level synthesis. This chapter presents an advanced, high-level synthesis system called AMICAL that is targeted toward

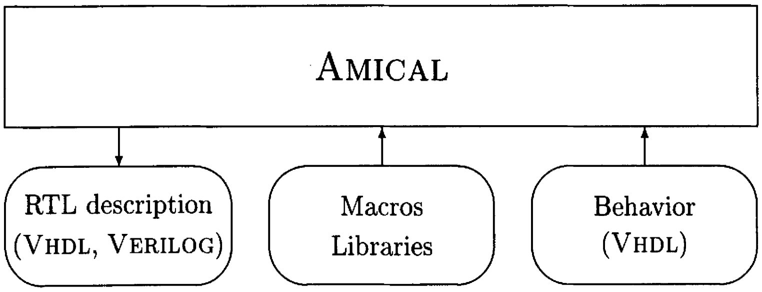


**Figure 1** A global view of AMICAL.

control-flow-dominated machines. AMICAL starts with a functional specification, given in VHDL, and generates an architecture, composed of a data-path and a controller, that may feed existing synthesis tools acting at the logic and register-transfer level. It also allows the re-use of existing components within new designs (through an external library of macros).

As shown in figure 1, behavioral synthesis starts with two kinds of information: a behavioral description and an external library of functional units (FUs). The external library of FUs may include standard execution units (adders, multipliers, ALUs, etc.) as well as more complex units defined by the designer. These may be large, complex blocks such as cache memories, I/O units, and so on.

AMICAL is organized as an interactive environment where automatic and manual synthesis can be mixed. The combination of automatic and manual synthesis allows a quick and broad exploration of the design space in real time. Furthermore, AMICAL provides many facilities for analyzing the generated architecture (such as statistics, evaluation, and links between the VHDL behavior and the architecture). Several large examples have already been used for AMICAL evaluation with excellent results.

The rest of this chapter concentrates on the methodologies and the underlying concepts behind the development of AMICAL rather than the individual algorithms. Details about the algorithms used by AMICAL can be found in literature [13, 12, 14].

The chapter is divided into six sections. The next section deals with interfacing high-level synthesis tools and existing CAD environments. Section 3 gives a global presentation of AMICAL. The input specification, the synthesis environment, and the design flow are presented. Facilities for mixing manual and automatic design are explained in section 4. Section 5 discusses the need for specific algorithms in order to synthesize control-flow-dominated machines. We also show the application of this approach to the design of a large circuit, namely a telephone answering machine controller. Section 6 provides an evaluation of the system and some conclusions.

## 2 INTEGRATION WITH EXISTING DESIGN ENVIRONMENTS

Although behavioral silicon compilation has made large strides toward the automation of VLSI design [3, 9], the use of such systems is currently very limited in industrial environments. The main problems are:

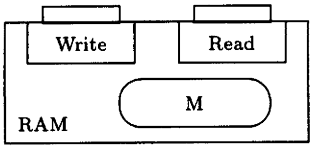
- The lack of integration within existing design methodologies.
- The lack of integration within existing CAD environments.

AMICAL's solution to the first problem consists of mixing manual and automatic design within an interactive, high-level synthesis environment. This approach will be detailed in the following sections. The rest of this section explains how the use of VHDL eases the integration of AMICAL within existing design environments. The use of VHDL allows a combination of behavioral and other description styles and levels for the specification of complex real-time systems. The use of VHDL packages and procedures also allows us to import existing macro-blocks into behavioral descriptions.

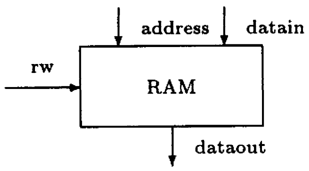
### 1 Using existing hardware

Functional units (FUs) are introduced in order to allow the mixing of behavior and structure. FUs allow the use of existing macro-blocks in the behavioral specification.

An FU may execute a set of standard operations (which is the case for adders, multipliers, ALUs, etc.) or new customized operations introduced by the user. An FU may be a large complex block such as a cache memory or an I/O unit. It can be called from within a behavioral description in order to perform a



```
PACKAGE RAM_fu is
  Type memory is array (0 to 3199) of integer;
  Signal RAM : memory;
  Procedure read(A : in integer;
                 RAM : in memory;
                 B : out integer);
  Procedure write(A,B : in integer;
                  RAM : in memory);
end RAM_fu;
```



```
(FU RAM
  (Area 25000) (Width 250) (Height 100)
  (Parameter (DataIn a b) (DataOut c))
  (Connector
    (DataIn address datain (Bit 0 8))
    (DataOut dataout (Bit 0 8))
    (ControlIn rw (Bit 0 2)))
  (opType read
    (Cycle 1
      (Active rw 1)
      (Transfer a address))
    (Cycle 2
      (Transfer dataout c)))
  (Optype write
    (Cycle 1
      (Active rw 2)
      (Transfer a address)
      (Transfer b datain))))
```

**Figure 2** Different FU abstractions. From top left: the conceptual view, the behavioral view, the RTL view, and the high-level synthesis view.

given operation. It can accept and return parameters. In terms of hardware, it allows us to create partial designs that can easily be introduced into new systems [14, 15].

An FU can be specified at different levels of abstraction, as shown in figure 2. In this example, a register file called RAM is described. From a conceptual point of view, the register file is an object able to execute two operations (Read, Write) which share some data (M). At the behavioral level, the FU is described with a VHDL package that includes two procedures specifying the execution details of the operations. This may be a complex behavior. The FU may correspond to an existing macro-block that has already been synthesized or described at the RT level. The RTL view is an external view of a possible realization of the register file. It is connected to two inputs (address, datain), one output (dataout), and one command (rw) that selects the procedure to execute.

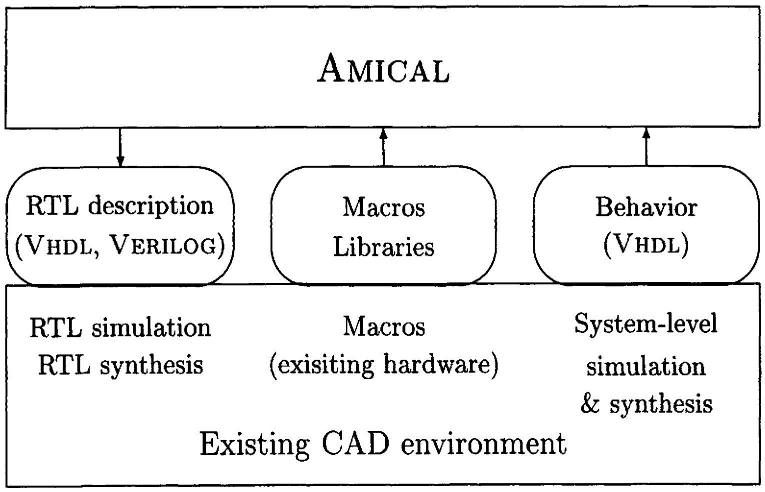

AMICAL
RTL description (VHDL, VERILOG)
Macros Libraries
Behavior (VHDL)
RTL simulation
RTL synthesis
Macros (exisiting hardware)
System-level simulation & synthesis
Existing CAD environment

**Figure 3** Integrating AMICAL within existing CAD environments.

The high-level synthesis view of the FU summarizes the behavioral and RTL views. It includes:

- The interface of the FU.
- The FU's call parameters (which correspond to procedure parameters).
- The operation set executed by the FU.
- The micro-schedule for each operation.

The synthesis environment can contain a library of such FUs. The designer invokes an FU through a simple procedure call. FUs can be of any degree of complexity and can themselves be the result of a synthesis process.

## 2 Linking AMICAL to existing tools and methodologies

The interaction between AMICAL and existing CAD environments is shown in figure 3. Starting from a behavioral description, AMICAL performs the normal high-level synthesis steps (scheduling, allocation, etc.) and generates an RTL structure composed of two subsystems: a data-path and a controller. Such a design can be realized by the following four steps:

- System level specification and verification.
- Architectural exploration and behavioral partitioning.
- Micro-architecture generation.
- RTL synthesis (RTL and layout synthesis environments).

Only the second and third steps are performed by Amical. The first and the fourth steps must be performed by the environment.

For the first step, the design is specified at the system level. This specification may be part of a complex system or a complete design. The specification should be verified through system-level simulation. The results of the simulation will be used during subsequent steps for architecture exploration. Another verification option would be to use formal techniques such as those described in chapter 2. Although these techniques have primarily been developed for the Ascis DFG and Amical concentrates on control flow graph descriptions, a large subset of Vhdl (ProcVhdl, described in section 3 of chapter 2) can be modeled by the Ascis DFG. The comparison of formal verification and exhaustive simulation is not relevant to this chapter. Through the Ascis project, we have both options open to us.

Once the behavioral description and the FU library are ready, Amical can be used for architectural exploration and synthesis. This step includes scheduling and allocation. The behavioral description is partitioned into a data-path and a controller. The synthesis can be carried out automatically, manually, or by a combination of both approaches. The automatic synthesis produces the fastest architecture according to the input description. This architecture may be manually modified in order to reduce the number of allocated FUs and busses (see section 4).

The third step produces an RTL specification. It proceeds in two steps: the first produces an abstract architecture, and the second customizes this output for a given RTL synthesis and logic environment. The customization process is programmable. It provides facilities such as handling different clocking schemes and different Vhdl styles.

Once the new Vhdl specification of the architecture is generated, a validation step is needed. This step mainly involves simulating the generated description. This post-architecture-synthesis simulation may be used to carry out a detailed performance analysis of the architecture (for example: the number of clock cycles needed to perform a given computation). Such a simulation can also be used to check the correctness of the synthesis process. This step also includes RTL and logic synthesis. For the controller synthesis and verification, techniques such as those described in chapter 10 can also be useful.

## 3 AN OVERVIEW OF AMICAL

This section gives a global view of the AMICAL system. More details about the algorithms used by AMICAL can be found in the literature [13, 12, 14].

### 1 User interaction

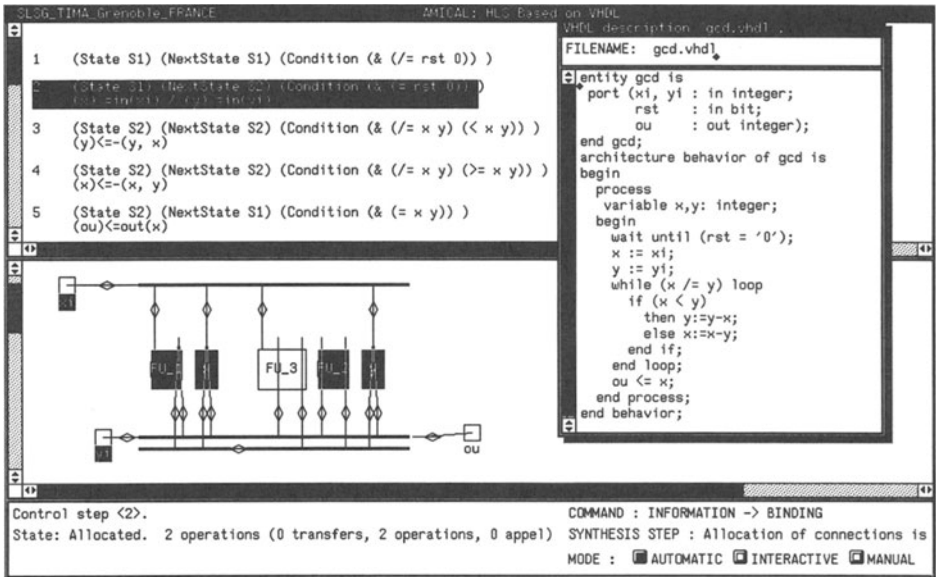
AMICAL is organized as an architectural synthesis environment. It works as a design assistant, combining automatic, manual, and interactive synthesis.

The designer interacts with the system through a mouse and a graphical interface. AMICAL's user interface consists of three windows:

- Control window (top): generally used to show and edit the controller.
- Data-path window (middle): used to show and edit the data-path.
- Information window (bottom): used to print information and error messages. It also provides information about the progress of the synthesis process (last command, synthesis step, contents of the other windows, etc). This information is needed in order to help the user during long synthesis sessions. This window also shows the synthesis mode (automatic, interactive, or manual).

As we shall see in the rest of this chapter, one of the main strengths of AMICAL is its ability to maintain the coherence between the information included in these three windows, in other words, the different aspects of the design.

Figure 4 shows a screen dump that gives a flavor of AMICAL at work. The right-hand window shows a VHDL description of the algorithm being synthesized. The top window contains the transition table generated by the scheduler. The middle window shows the data-path, as synthesized by AMICAL. The bottom window provides information on the current status of AMICAL. We make use of concepts similar to those used in CORAL II [1] in order to link the behavior and structure. In this way, AMICAL is able to maintain the coherence


SLSG_TIMA_Grenoble_FRANCE
AMICAL: HLS Based on VHDL
VHDL description 'gcd.vhdl'.
FU_2
FU_3
FU_1
ou

```
1    (State S1) (NextState S1) (Condition (& (/= rst 0)) )
2    (State S1) (NextState S2) (Condition (& (= rst 0)) )
     (x)<=in(xi) / (y)<=in(yi)
3    (State S2) (NextState S2) (Condition (& (/= x y) (< x y)) )
     (y)<=-(y, x)
4    (State S2) (NextState S2) (Condition (& (/= x y) (>= x y)) )
     (x)<=-(x, y)
5    (State S2) (NextState S1) (Condition (& (= x y)) )
     (ou)<=out(x)
```

```
FILENAME:  gcd.vhdl

entity gcd is
 port (xi, yi : in integer;
       rst    : in bit;
       ou     : out integer);
end gcd;
architecture behavior of gcd is
begin
  process
   variable x,y: integer;
  begin
    wait until (rst = '0');
    x := xi;
    y := yi;
    while (x /= y) loop
      if (x < y)
        then y:=y-x;
        else x:=x-y;
      end if;
    end loop;
    ou <= x;
  end process;
end behavior;
```

```
Control step <2>.
State: Allocated.  2 operations (0 transfers, 2 operations, 0 appel)
```

```
COMMAND : INFORMATION -> BINDING
SYNTHESIS STEP : Allocation of connections is
MODE :  ■ AUTOMATIC  □ INTERACTIVE  □ MANUAL
```

**Figure 4** Screen dump of AMICAL at work.

between the information contained in the three main windows. For example, the designer may require information concerning the correspondence between the controller and the data-path. In figure 4, we have asked for information about the resources used for the execution of the two parallel operations in transition 2 of the controller representation (highlighted in top window). AMICAL has highlighted the appropriate data-path components in the middle window. The resources highlighted by AMICAL include not only the registers and functional units necessary for storing the variables and executing the operations, but also the busses and switches used in transferring the variables to and from the functional units. More information about the control step is given in the bottom window. The interaction with the user is similar to the MIES system [10].

Starting from a behavioral-level VHDL description, the designer has the option of synthesizing entirely automatically, proceeding manually, or combining automatic and manual steps in a true interactive mode. This philosophy of allowing the designer to influence the synthesis process as much as desired makes AMICAL a true computer-*aided* design tool. The various synthesis tasks in the AMICAL design flow are briefly outlined in the following paragraphs.

## 2 Input specification

As stated above, the compilation process starts with a behavioral specification given using a VHDL subset. The specification consists of an entity/architecture pair, with the architecture limited to a single process statement. The VHDL subset used is slightly different from that used by PROCVHDL in order to accommodate properties of control-flow-dominated circuits. The main difference is that some restrictions of PROCVHDL concerning loops, loop exit statements, and wait statements have been removed.

The system uses a library of FUs similar to the application-specific units (ASUs) used in CATHEDRAL-3, which is described in chapter 7 [11]. In CATHEDRAL-3, the ASUs are extracted automatically from the behavioral description. This is important for high-throughput real-time signal processing applications that require complex application-specific data-paths, as targeted by CATHEDRAL-3. Within AMICAL, the FUs are provided by the user. Even if this approach is less automatic, it provides more flexibility when using existing hardware.

Accesses to arrays and ports are automatically converted to functional unit calls. For example, suppose we have an FU named *ram*, described in VHDL as an array (thus it is represented as a memory device in hardware, the array index being the address). Assignments to *ram* in the initial description would then be replaced by WRITE requests, and if *ram* was used as an operand in some operation, this would be replaced by a READ function. For our control-flow-dominated target domain, we have chosen to realize the memory access as implied by the VHDL description. No memory-oriented loop optimizations, as addressed in chapter 7 and 8, are foreseen here.

## 3 Scheduling

AMICAL uses a dynamic loop scheduling algorithm [13, 12]. This algorithm is adapted to control-flow-dominated designs written in VHDL, and is a development of the path-based approach proposed by Camposano [2]. Essentially, the scheduler reads in a VHDL description and produces a behavioral FSM in the form of a transition table. Each transition (macro-cycle) corresponds to the execution of a control step under a given condition. A macro-cycle may need several basic cycles (clock cycles) for its execution. The top left window in figure 4 shows the transition table composed of two states and five transitions.

The default option is that all the operations of a transition are executed in parallel. This schedule may be modified manually during an extra chaining step.

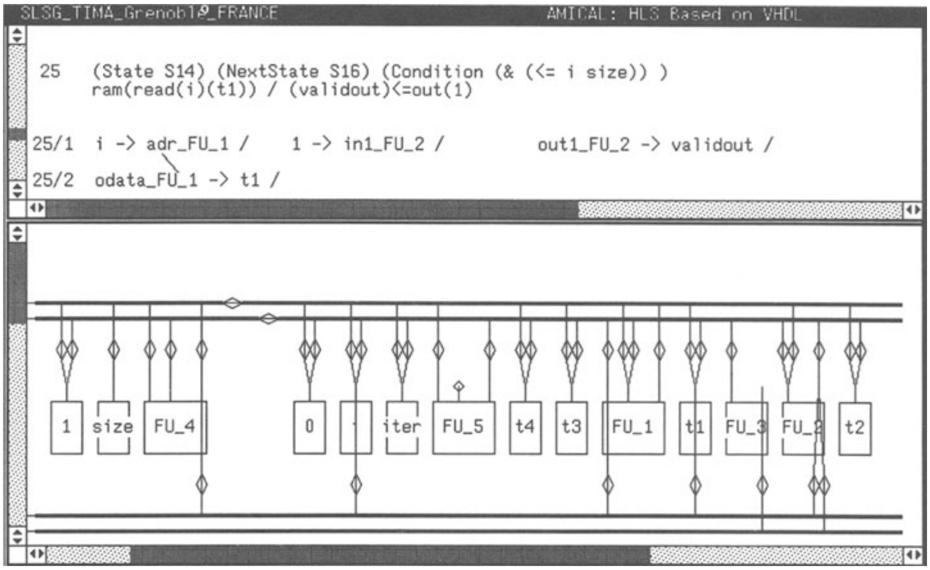
Chaining implies the serialization of operations in the same control step. It is used, for example, when the initial schedule generates a control step containing too many parallel operations, requiring a lot of FUs to execute them. Chaining is carried out interactively.

## 4 Architecture synthesis

After scheduling, architecture synthesis starts with two kinds of information, namely the scheduled description and an external library of functional units. The resulting architecture is bus-based. The default option is that there is no limitation on the number of busses. The architecture synthesis involves four steps [15]:

- Functional unit allocation: associates an FU with each operation in the state table.
- Micro-scheduling: generates the micro-schedule according to the execution scheme given for each operation (given in the FU specification). Each operation is decomposed into a set of transfers. Each micro-cycle contains a set of parallel transfers that take one basic clock cycle to execute.
- Component placement: places the registers and FUs in order to reduce the connections. Because AMICAL uses a bus-based target architecture, the placement of registers and functional units is very important. Optimal placement will minimize the number of busses necessary.
- Connection allocation: produces the bus structure. For each transfer, a set of connections containing wires, switches, and busses must be allocated in such a way that parallel transfers do not use the same resources. The efficiency of the result of this step largely depends on previous synthesis steps.

FU allocation is based on an EMUCS-like algorithm [5]. Micro-scheduling uses an ASAP algorithm. Component placement and connection allocation make use of improved constructive approaches similar to those used in APOLLON [6]. The originality of the approach is that all of these algorithms are implemented in order to allow mixed manual and automatic design. With the exception of micro-scheduling, they all use a constructive approach. At each step, a new element is allocated or placed. The four steps may be sequenced automatically or performed step by step. Each step may be executed automatically, interactively, or manually.



```
25    (State S14) (NextState S16) (Condition (& ((<= i size)) )
      ram(read(i)(t1)) / (validout)<=out(1)

25/1  i -> adr_FU_1 /    1 -> in1_FU_2 /        out1_FU_2 -> validout /

25/2  odata_FU_1 -> t1 /
```

**Figure 5** The data-path generated automatically for the bubble-sort example.

Figure 5 shows the data-path generated automatically for a bubble-sort example [15]. The top window shows the detailed micro-scheduling of control step 25. According to the corresponding FU description, each operation is split into a set of transfers related by precedence constraints (lines in the the top window). In this case, the READ *ram* operation needs two micro-cycles for execution. The micro-scheduling of this control step resulted in two micro-cycles (basic clock cycles).

## 5 Architecture generation

The output of AMICAL is a structure composed of two subsystems: a data-path and a controller. This RTL specification is generated in two steps. The first produces an abstract architecture coded in an intermediate form called SOLAR [7]. In order to reach silicon, this abstract architecture needs to be refined in order to include a synchronization scheme (clocks, resets) and other characteristics such as testing. This refinement produces a detailed architecture specification. During the last step, glue cells may be inserted. For example, a synchronization block may be included anywhere in the circuit hierarchy, if such a personalized scheme is needed. This may be useful in the case of a circuit that uses a single external clock signal and includes a clock generator that produces internal clock signals.

The final output of AMICAL is a netlist composed of a control unit and a data-path. The data-path is itself a netlist. The controller is an FSM description that can be fed to an FSM synthesizer such as that described in chapter 10.

## 4 MIXING MANUAL AND AUTOMATIC DESIGN

The classic behavioral synthesis design methodology consists of three fundamental steps: scheduling, allocation, and architecture generation. There is much controversy over the ordering of the first two steps. AMICAL compromises by providing an initial schedule and allowing the designer the freedom to mix manual and automatic modes in order to complete the design. This section describes the facilities provided by AMICAL in order to ease the mixing of manual and automatic design.

### *Interaction modes*

Mixing manual and automatic design can be performed through two modes: a true interactive mode and a manual modification of automatic synthesis mode. All allocation algorithms are iteratively constructive, allowing the user to intervene at each iteration, to modify the results or to cancel them completely. The scheduling tasks (scheduling and micro-scheduling) are performed automatically. The designer can manually modify the results of the schedule.

### *Linking the controller and the data-path*

An important feature of AMICAL is maintaining links between the initial VHDL description, the transition table, and the data-path. These links simplify the control of the design process when manual and automatic design is mixed. Through the *information* submenu, the user may ask for relationships between the synthesized structure and the scheduled description (see also section 3).

### *Verification functions*

For each task there are, of course, certain rules that must be adhered to. For all manual and interactive interventions, a verification is automatically carried out by AMICAL. Only functionally correct changes are accepted. For instance,

```
(Evaluation of the synthesized data-path
   ...
   (Allocated FUs (number :  5) (area 33900.00)
      (Allocated FU <FU_1> == <ram> (area :  25000.00))
      (Allocated FU <FU_2> == <IO> (area :  2000.00))
      (Allocated FU <FU_3> == <IO> (area :  2000.00))
      (Allocated FU <FU_4> == <ALU3> (area :  3100.00))
      (Allocated FU <FU_5> == <SUB> (area :  1800.00))
      (MAX_NUMBER 10) (WEIGHT 1)
      (ESTIMATION on FU allocation :  SUCCESS))
   (Allocated connections (area 113120.00)
      (Allocated busses (number :  4)
         (BUS_1 :  <BUS_1_1> <BUS_1_2>)
         (BUS_2 :  <BUS_2_1> <BUS_2_2>)
         (BUS_3 :  <BUS_3_1>)
         (BUS_4 :  <BUS_4_1>)
         (MAX_NUMBER 3) (WEIGHT 10)
         (ESTIMATION on bus allocation :  FAIL)))
   (Scheduled micro-cycles (number :  33)
      (MAX_NUMBER 100) (WEIGHT 1)
      (ESTIMATION on micro_cycle scheduling :  SUCCESS)
   ...
   (FINAL_ESTIMATION : FAIL))
```

**Figure 6** Evaluation file for bubble-sort data-path.

during allocation steps, only correct bindings are accepted. During manual modification of the scheduling, transformations that violate data dependencies are not accepted.

## *Evaluation functions*

AMICAL also provides several evaluation functions aimed at on-line evaluation of the synthesis tasks already executed. This evaluation summarizes the hardware allocated as well as verifying that certain constraints have been met. Figure 6 shows an extract of the evaluation file corresponding to the architec-

```
(Statistics of the synthesized data-path
   (FUs (Total Number :  5)
      (Name FU_1 (Active Cycle 19 (Rate 57.58%)))
      (Name FU_2 (Active Cycle 6 (Rate 18.18%)))
      (Name FU_3 (Active Cycle 2 (Rate 6.06%)))
      (Name FU_4 (Active Cycle 9 (Rate 27.27%)))
      (Name FU_5 (Active Cycle 1 (Rate 3.03%))))
   (Bus (Total number :  6)
      (Name BUS_1_1 (Active Cycle 13 (Rate 39.39%)))
      (Name BUS_1_2 (Active Cycle 33 (Rate 100.00%)))
      (Name BUS_2_1 (Active Cycle 7 (Rate 21.21%)))
      (Name BUS_2_2 (Active Cycle 11 (Rate 33.33%)))
      (Name BUS_3_1 (Active Cycle 9 (Rate 27.27%)))
      (Name BUS_4_1 (Active Cycle 3 (Rate 9.09%)))))
```

**Figure 7** The statistics file corresponding to figure 5.

ture shown in figure 5. We can easily see that this architecture includes five FUs and four busses organized into six segments. The controller corresponding to this architecture includes 33 micro-cycles. Statistics about the use of resources are also provided (figure 7). These are the results of static analysis. The statistics file provides the number of cycles where each resource is used.

For complex designs, these feedback files provide useful indications that may guide the designer's decisions. For example, in figure 6, we note that the number of busses allocated has exceeded the imposed maximum. In order to reduce the number of busses, we can perform some manual micro-scheduling to remove some of the parallel transfers.

## 5 A DESIGN EXAMPLE

This section deals with the design of a complex control-flow-dominated circuit, namely a controller for a telephone answering machine. The main characteristics of this design include real-time constraints and synchronization, nested loops, and emergency exits from loop hierarchies. Figure 8 shows a block diagram of the answering machine. This example is based on the answering machine proposed by Vahid and Gajski [16].

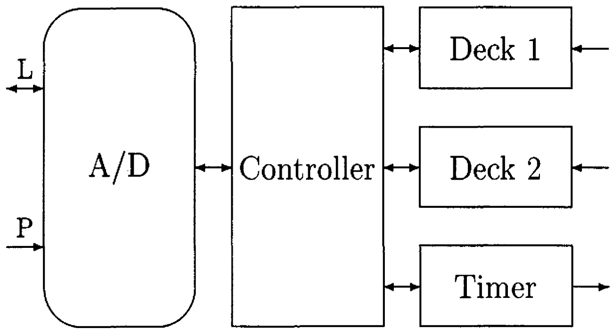


**Figure 8** Telephone answering machine block diagram.

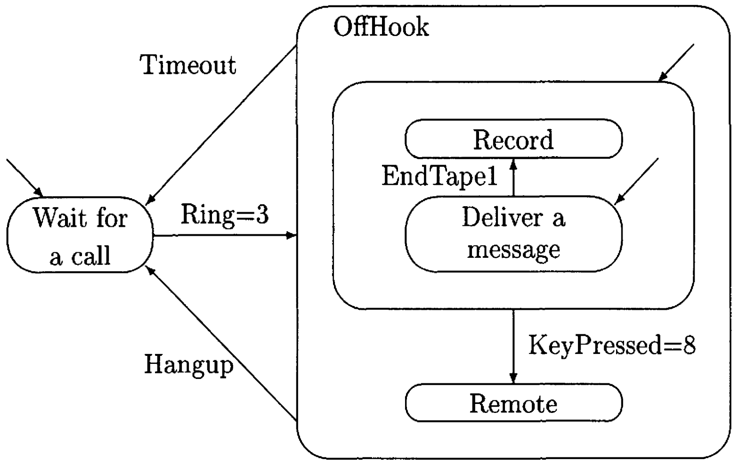


**Figure 9** StateChart-like model of answering machine controller.

The answering machine consists of five principal blocks: an A/D converter, two tape decks, a timer, and a controller. It is this last block that will be used to demonstrate the capability of AMICAL to deal with large control-flow-dominated designs. Figure 9 shows a StateChart-like model [4] of the telephone answering machine controller. The entry state of the system is the state *Wait_For_A_Call.* When three rings have been received from the telephone exchange, a transition to the state *OffHook* is made. In this state, a pre-recorded message is delivered and the caller can continue or hang up. If the caller hangs up at any stage during the process, an immediate transition to the state

| Synthesis operations | FUs | Buses | Bus segments | Control signal switches | Micro-cycles |
|---|---|---|---|---|---|
| Automatic synthesis | 6 | 6 | 8 | 83 | 50 |
| After re-scheduling | 3 | 4 | 7 | 61 | 73 |
| After re-scheduling and micro-schedule optimization | 3 | 3 | 3 | 44 | 89 |

**Table 1** Summary of different solutions for the answering machine controller obtained using AMICAL.

*Wait_For_A_Call* must be made. A similar transition is made if any of the timeout restrictions is not adhered to.

The controller is modeled as a single VHDL process containing seven nested loops organized in the hierarchy shown in figure 10. The VHDL code required to implement the *Get_3_Digits* loop is shown in figure 11. During the execution of any of these loops, if there is a global exception (the caller hangs up, for example), the loop hierarchy must exit and control be passed to the *Wait_For_A_Call* loop. The VHDL subset used permits us to use loop exit statements to implement global exceptions. After each *wait* statement, all global exceptions are tested.

The full description contains 180 lines of VHDL code. An initial scheduling produces a transition table containing 22 states and 67 transitions. Some of these transitions include up to six parallel operations (transfers, arithmetic operations, and memory accesses), implying eight parallel transfers. After a full automatic synthesis session, the solution produced is summarized in row 1 of table 1. This is the fastest solution in terms of the number of micro-cycles. For this application, however, we can afford to trade off some of the speed against data-path resources. Therefore, to improve this design, it is necessary to perform scheduling and micro-scheduling modifications.

An evaluation showed that three of the FUs allocated were I/O units. The initial schedule produced 17 transitions containing two parallel I/O operations and six transitions containing three parallel I/O operations. A re-scheduling step was performed in order to eliminate two I/O units. The resulting solution

```
(LOOP behavior --implicit process loop
   (LOOP Wait_For_A_Call
      (LOOP Get_3_Rings))
   (LOOP OffHook
      (LOOP Remote
         (LOOP Get_3_Digits)
         (LOOP ManualControl))))
```

**Figure 10** Loop hierarchy of the controller.

```
Get_3_Digits :  LOOP
  WAIT UNTIL (KeyPressed = 0);
  Timeout := functional unit_REM(ElapsedTime,20) ;
  WAIT UNTIL ((ElapsedTime=Timeout) OR (KeyPressed/=0)
     OR (HangUp = '1'));
  IF ((HangUp = '1') OR (ElapsedTime = Timeout)) THEN
    EXIT OffHook;
  END IF;
  NextDigit := PasswdROM(DigitCount);
  IF (KeyPressed /= NextDigit) THEN
    FalsePasswd := '1';
  END IF;
  DigitCount := DigitCount + 1;
  IF (DigitCount = 3) THEN
    EXIT Get_3_Digits;
  END IF;
END LOOP Get_3_Digits;
```

**Figure 11** VHDL code segment of the controller.

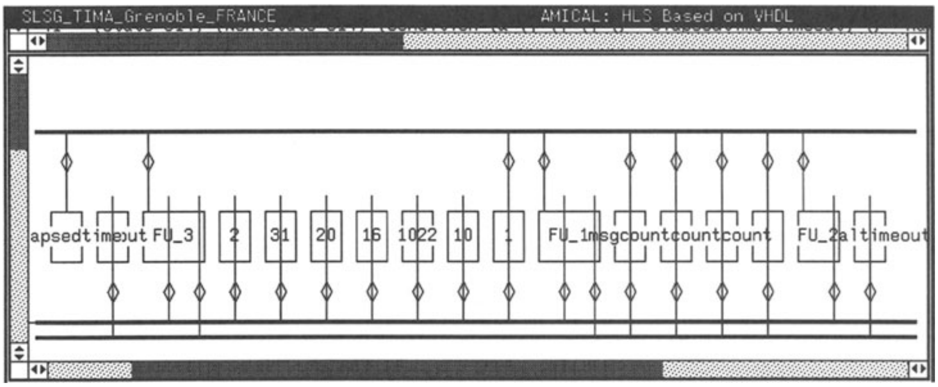


**Figure 12** Final optimized data-path for the answering machine.

is summarized in row 2 of table 1. This transformation caused the number of micro-cycles to increase by 23, which we have decided is an acceptable side effect. The number of busses, however, is still very high (four busses organized into seven segments).

In order to save connections, a manual modification of the micro-scheduling was performed. The resulting solution is shown in row 3 of table 1. The synthesized data-path is shown in figure 12. The final data-path included only three busses with no cuts.

In table 1, the fifth column shows the number of switches necessary for the control of the data-path. These switches correspond to control bits coming from the controller. We can note that resource saving has implied a reduction in the number of switches (from 83 to 44). This reduction partly offsets the increase in the controller size due to the increase in the number of micro-cycles (from 50 to 89).

The response time of AMICAL is reasonably short, making it a genuine interactive system. The synthesis speed is a key issue if the designer is to be able to try different alternatives and obtain the best architecture. None of the compilation steps corresponding to the examples used in this chapter needed more than 10 seconds on a Sun SPARCstation 2.

## 6 CONCLUSION

In this chapter, the Amical compiler, targeted to the synthesis of control-flow-dominated applications, has been described. Most of the common high-level synthesis benchmarks have already been compiled with Amical. Experiments have shown that for most benchmarks, the resulting solutions are as good as or better than those produced in other ways [13, 14]. These solutions can be obtained by mixing automatic and manual design or by using an automatic exploration of the design space. These performances also apply to large designs.

Several real-life examples have also been used for Amical evaluation. The most complex is a programmable waveform generator. The design is composed of four modules. The largest module results in an FSM with 548 states and 1,474 transitions. The compilation of this block takes less than one minute on a Sun SPARCstation 2.

The short response time of Amical is crucial for design productivity. The synthesis speed has been considered as a key issue to let the designer try different alternatives and to get the best architecture. The combination of automatic and manual synthesis allows a quick and broad exploration of the design space in real time.

Apart from the integration of Amical with an existing logic synthesis tool, future work includes the generation of other styles of architecture (e.g., mux-based) and to extend the interaction with Amical in order to allow micro-architecture simulation [10].

## REFERENCES

[1] R. L. Blackburn et al. Coral II: linking behavior and structure in an IC design system. *Proc. 25th DAC*, 1988.

[2] R. Camposano. Path-based scheduling for synthesis. *IEEE Trans. on CAD*, CAD-10, number 1, pages 85–93, Jan 1991.

[3] R. Camposano and W. Wolf, editors. *High-Level VLSI Synthesis*. Kluwer Academic Publishers, 1991.

[4] D. Harel et al. Statecharts: a working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, 16, number 4, 403–413, Apr 1990.

[5] C. Y. Hitchcock and D. E. Thomas. A method of automatic data-path synthesis. *Proc. 20th DAC*, paper 31.3, 1983.

[6] R. Jamier and A. A. Jerraya. APOLLON: a data-path silicon compiler. *IEEE Circuits & Devices*, May 1985.

[7] A. A. Jerraya and K. O'Brien. SOLAR: an intermediate form for system-level design and specification. *CoDes Workshop*, Grassau, Germany, 1992.

[8] A. A. Jerraya, I. Park, and K. O'Brien. AMICAL: an interactive high-level synthesis environment. *Proc. EDAC'93,* Paris, Feb 1993.

[9] M. C. McFarland, A. C. Parker, and R. Camposano. The high-level synthesis of digital systems. *Proc. of the IEEE*, 78, number 2, Feb 1990.

[10] J. Nestor, B. Soudan, and Z Mayet. MIES: a micro-architecture design tool. *Proc. 22nd International Workshop on Microprogramming and Microarchitecture*, 1989.

[11] S. Note et al. Cathedral III: architecture-driven HL synthesis for high-throughput DSP applications. *Proc. 28th DAC*, 1991.

[12] K. O'Brien, M. Rahmouni, and A. A. Jerraya. A VHDL-based scheduling algorithm for control-flow dominated machines. *6th Intl. High-Level Synthesis Workshop*, Dana Point Resort CA, Nov 1992.

[13] K. O'Brien, M. Rahmouni, and A. A. Jerraya. DLS: a scheduling algorithm for high-level synthesis in VHDL. *Proc. EDAC'93*, Paris, France, Feb 1993.

[14] I. Park, K. O'Brien, and A. A. Jerraya. An interactive data-path allocation algorithm. *IFIP Workshop on Control Dominated Synthesis from a Register Transfer Level Description*, Grenoble, France, 1992.

[15] I. Park. *AMICAL: Un assistant pour la synthèse et l'exploration architecturale des circuits de commande.* PhD thesis, INPG, Grenoble, France, 1992.

[16] F. Vahid and D. D. Gajski. Specification partitioning for system design. *Proc. 29th DAC*, 1992.

10

# CONTROLLER SYNTHESIS AND VERIFICATION

**Kenny Ranerup[1], Lars Philipson[1]**
**Jan Madsen[2], Ole Olesen[2], Geert Janssen[3]**

[1] *Lund University*
[2] *Technical University of Denmark*
[3] *Eindhoven University of Technology*

## ABSTRACT

This chapter focuses on synthesis and verification of control units. One of the key issues in synthesis is the ability to explore the design space. One step toward design space exploration is the results presented here in control architecture synthesis that enables exploration of a range of control architectures. Another step is the use of a compiled cell approach to the technology mapping problem in control unit logic synthesis. The verification of the synthesized control unit is also an important issue. A new approach is presented that, using a combination of propositional temporal logic verifier and sequential logic extraction, has made it possible to verify formally the layout of a control unit against the specification.

## 1 INTRODUCTION

One of the goals of building synthesis systems for VLSI design is to allow the designer to explore a large part of the design space in a short time. The previous chapters of this book (chapters 7, 8, and 9) have addressed the exploration for data-path synthesis, but nothing has been said about control unit (CU) design.

The process of designing a CU typically starts with choosing an architecture suitable for the particular control program at hand. The choice of architecture can be greatly simplified by the use of algorithms for mapping a control program onto many CU architectures. This enables an exploration of the architectural design space in a way not possible before. The results described in section 2 from this approach show that significant area gains can be achieved.

The second step in CU design is implementation of the architectural components in an efficient way. Several implementation methods are available today for the combinational part of a CU, for example, PLA and standard cells. Accompanied by modern logic synthesis methods, the design space can be explored in both area and time directions. There are, however, disadvantages with both PLA and standard cell solutions. The standard cell solution maps onto a fixed library of cells which restricts the logic synthesis from finding an optimal solution. As an alternative to the fixed library, an appealing approach is the use of cell compilers which are able to generate *any* logic function as a single composite gate, i.e., reducing the number of transistors needed to implement the function. This approach can result in significant area gains compared to other methods, as described in section 3.

Last but not least, the function of the CU has to be verified. Verification is divided into two processes: validating the control program against the designer's intentions, and verifying that the resulting layout is a correct implementation of the control program. The complexity of the control program often makes this a difficult task, and the verification also becomes even more complex as more and more tools are used that potentially can introduce errors in the resulting layout. In the left part of figure 1, a typical design flow is depicted, each step involving complex algorithms.

The only tools traditionally available for functional verification are simulators. The current practice is to simulate the circuit for a large number of cases. Unfortunately, it is usually not feasible to simulate exhaustively, i.e., to simulate for all possible input sequences; hence, only partial correctness of the design is typically established.

A fundamentally different method of checking correctness is the use of mathematical proof techniques for proving the correctness of a design. These methods do not have to resort to exhaustive simulation to achieve full confidence of correctness. One such method based on propositional temporal logic is described in section 4. In principle this method can be used to verify a transistor netlist extracted from the layout against the control program, but in practice this is not computationally feasible. The problem is that there is too large a gap of abstraction between the transistor netlist and the control program. One way of bridging this gap, described in section 5, is to extract an equivalent sequential logic description from the transistor netlist. Such a description is much more suitable for verification against the control program.

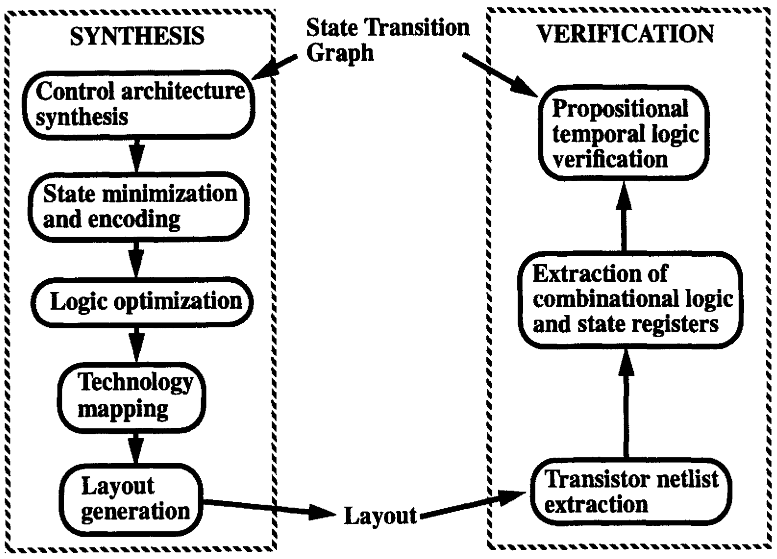

SYNTHESIS
Control architecture synthesis
State minimization and encoding
Logic optimization
Technology mapping
Layout generation
State Transition Graph
Layout
VERIFICATION
Propositional temporal logic verification
Extraction of combinational logic and state registers
Transistor netlist extraction

**Figure 1** Controller design and verification process.

## 2 ARCHITECTURE SELECTION

Automatic synthesis of control architectures from a program description is a very important task in order to allow the designer to explore a large part of the design space with little effort. Until now, such approaches have mainly been addressed for data-path architectures, as described elsewhere in this book.

Certain parts of the synthesis and optimization of control logic, such as state assignment, two-level logic optimization, and multilevel logic minimization [11, 6, 5], have been well studied. The general approach for CU synthesis has been to generate and optimize logic for given, fixed control architectures. For a given control program, different control architectures can, however, result in implementations that differ widely in terms of size and speed. Therefore, a CU synthesis system should be able automatically to map a single CU program onto a number of architectures and eventually automatically choose an architecture, optimal for the particular control program at hand, given certain performance constraints.

Research into such control architecture synthesis has mainly been along two different approaches:

- One approach is based on design experience from CU design and microprogramming, where characteristics of different control architectures are well known. Algorithms for mapping control programs onto some of these architectures has been developed, using, for example, microprogram counters [2] and subroutines [23].
- Another approach involves more general decomposition methods, where a single CU is decomposed into a set of communicating finite state machines [12, 14].

The results presented in this section follow the first approach. The resulting system allows the designer to explore a range of CU architectures.

## 1 A range of CU architectures

CU programs are usually expressed as state transition graphs (STGs). In many CU programs, it is possible to find subsequences that occur in several places in the STG. Two sequences are identical if and only if they produce the same output sequences for all possible input sequences. The basic idea of a range of different architectures is to re-use these state sequences so that only one copy is needed in the implementation, thereby reducing the area of the CU.

### *Stack architecture*

Subroutines in micro-programmed CUs is one example of how common subsequences can be re-used. The subroutine calling mechanism can be implemented in many ways. Saving the calling state on a stack is one possibility. The traditional FSM built with a PLA and D-flipflops can easily be extended with a subroutine mechanism by replacing the state registers with a stack of state registers (figure 2).

The return operation in this subroutine mechanism differs from the usual microprocessor type of subroutines in that it returns the FSM to the calling state, not the state after the calling state. Traversing the STG in figure 2 would therefore lead to the sequence *1,2,a,b,c,d,2,3.* An extra flip-flop is needed to distinguish between the two outgoing edges of state 2. The extra registers for implementing the stack increases the register area, but in many cases the corresponding decrease in the combinational logic area gives a total area decrease.

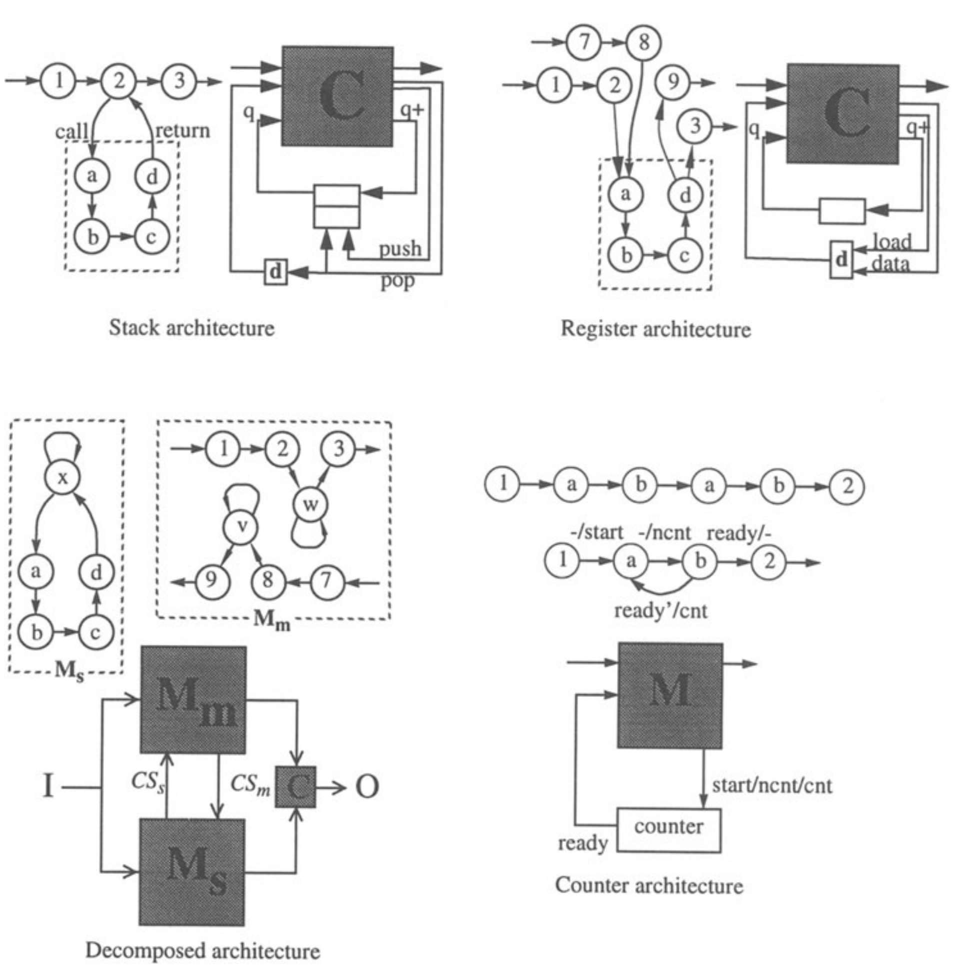


**Figure 2** Target architectures.

### *Register architecture*

A similar architecture is based on the observation that it is unnecessary to save the complete state in a subroutine call. If a subroutine is called from $n$ different states, only $\log_2 n$ bits need to be saved to later return to the correct state. This is accomplished with a register loaded with an identifier when a subroutine is called. The value of this register determines which state to return to from a subroutine call. In the STG in figure 2, the subroutine $a,b,c,d$ is called twice, and therefore one bit is needed to select the correct return state. This architecture is in many cases more area efficient than the stack architecture, since the number of registers needed frequently is smaller. However, when the number of calls to one subroutine is large, the stack architecture is more efficient.

### *Decomposed architecture*

Another possible way of taking advantage of common subsequences in the STG is to decompose the STG into two STGs, where one contains subroutines ($M_s$) and the other ($M_m$) contains the main part of the STG (see figure 2). The subroutine call/return is signaled between the machines by the current state ($CS_m$, $CS_s$). It is unnecessary to communicate the entire state between the machines. As in the register architecture, only a coding of the subroutine states is needed. In this case a unique coding of the different subroutines is needed from $M_m$ and only a return signal is needed from $M_s$. Note that when the CU is decomposed into two separate machines, some of the logic will be duplicated. This can, however, be overcome by optimizing the combinational logic of the two machines together (see "Decomposed separate" and "Decomposed combined" in table 1).

### *Counter architecture*

Repetitions of a subsequence is a special case of identical sequences. If a subsequence is repeated a large number of times (i.e., in the sequence $a,b,a,b,a,b,$ the sequence $a,b$ is repeated three times), it can be advantageous to replace the repetitionstate repetition with a loop over the sequence and an external counter (see figure 2). The counter only counts a fixed number of cycles and then generates a ready signal. More than one repetition can be replaced in this way if the counter is extended to allow a number of different cycle counts.

| FSM | Orig area $\mu m^2$ | Stack | Register | Decomposed | |
|---|---|---|---|---|---|
| | | | | separate | combined |
| brain | 221328 | 0.86 | 0.91 | 1.66 | 1.48 |
| e84tr_utan | 380016 | 0.60 | 0.66 | 1.12 | 0.99 |
| gammal | 335008 | 0.53 | 0.50 | 0.85 | 0.76 |
| lee2601c | 20416 | — | 0.52 | — | — |
| newpuls | 137808 | 0.58 | 0.60 | 0.60 | 0.54 |
| sand | 568864 | — | 0.84 | — | — |

**Table 1** Comparison of areas for the different controller architectures. The areas of the stack, register, and decomposed architectures are relative to that of the original architecture.

## 2 A mapping algorithm

To allow the designer to explore these architectures, an algorithm has been developed that maps a control program expressed as an STG onto the different architectures [23, 20]. This algorithm is divided into several steps. The first step is architecture independent and consists of finding all factors in the STG. A factor is a set of states (subsequences) that occurs at several places in a STG. The second step is to filter out the factors that are compatible with the chosen architecture. The last step is a heuristic algorithm for choosing the best factors (in respect to area gain) from the set of compatible factors.

## 3 Results

Experiments have been performed on a number of examples [22] in order to compare the area of the original FSM implementation (combinational logic and state register) with the areas of implementations onto the four target architectures (including the area for registers and counters). In these experiments, the combinational parts are implemented as multilevel logic using MIS [5] to optimize and map the logic onto the 2 $\mu$m standard cell library Lib2 from MCNC. The area of the multilevel implementation excludes routing. The results are shown in table 1. The results are expressed relative to the original implementation. As can be seen in the table, significant gains in area can be realized by using different architectures for different CU programs. For a more detailed study of implementation aspects, see section 3 [23, 20, 22].

The results for the different architectures can also be analyzed. There is a tendency that the register architecture is good in many cases while the stack architecture is less useful. This is explained by the low overhead of the register subroutine mechanism, which makes it suitable to a large range of CUs. The stack mechanism has a larger overhead, but in CUs with many subroutines the advantage of this mechanism overcomes the larger overhead, and this architecture outperforms the others (see e84tr_utan).

The decomposition architecture also has a relatively large overhead which makes it less useful, but still, examples can be found where this architecture outperforms the others (e.g., the example newpuls). There is also a clear benefit in optimizing the combinational parts of the two machines together.

The choice of architecture also influences the maximum clock frequency of the CU. This has not been analyzed in detail, but there is a clear tendency that the delay is decreased as the area decreases. This is particularly true for PLA implementations, but also standard cell implementations become faster as the area decreases.

There is, of course, always a question about how good the heuristic algorithm is compared to the optimal solution. The only result so far is an indication that the mapping algorithm performs favorably compared to a human designer in exploiting these architectures [23].

The CPU time of the mapping algorithm is fairly small and can be divided into two parts. Finding the factors takes less than one second for all examples on a Sun SPARCstation 2. The heuristic algorithm for selecting good factors takes from under one second up to two minutes. This time is dependent on the target architecture; the register architecture is the slowest.

## 3 ARCHITECTURE IMPLEMENTATION

Automatic synthesis of control units has been dominated by PLA implementations, where boolean functions in the sum-of-product form are mapped onto an array structure, or standard cell implementations, where the optimized logic functions are mapped onto a predefined library of cells. As already mentioned, an alternative to the fixed library is the use of cell compilers, which permit full exploitation of the advantages of the optimized logic.

| FSM | PLA | | | Standard Cells | | |
|---|---|---|---|---|---|---|
| | Orig area ($\mu m^2$) | Stack | Register | Orig area ($\mu m^2$) | Stack | Register |
| dk512 | — | — | — | 174616 | 1.08 | — |
| e84cd_utan | 513131 | 1.05 | 0.91 | 2114444 | 0.69 | — |
| ex451 | 77480 | — | 0.90 | 78720 | 1.26 | — |
| jorgen_utan | 633866 | 0.75 | 0.69 | 2247594 | 0.49 | 0.52 |
| lee2601c | 59300 | — | 0.76 | 62832 | — | — |
| newpuls | 124624 | 1.00 | 0.85 | 232560 | 0.86 | 1.19 |
| sse | — | — | — | 322524 | 0.96 | — |
| visa_utan | — | — | — | 116272 | 0.98 | — |

**Table 2** Results of PLA and standard cell implementation using a 2$\mu$m CMOS process. Both solutions have been implemented in the GDT environment of Mentor Graphics. For the stack and register architectures, the area relative to that of the original architecture is given.

## 1 PLA synthesis

Implementing the combinational part of a control architecture using a PLA is very common. The synthesis onto PLAs is efficient and well understood; however, the PLA structure is not very flexible, usually resulting in fixed speed and an aspect ratio that cannot be varied. Furthermore, a multilevel implementation will in many cases result in smaller and faster circuits.

The examples of table 1 have been synthesized through the two-level logic optimizer Espresso [6], and implemented using a custom PLA generator built in the GDT environment [9]. The results for the three architectures presented in table 2 include the area for state registers, etc. For the stack and register architectures, only the area relative to the original architecture is presented.

## 2 Synthesizing for standard cells

Using standard cells for implementation of the control architecture allows for the implementation of both combinational and clocked elements. However, the combinational part that is implemented as multilevel logic is separated in the optimization step, and then afterwards merged with the clocked elements before placement and routing.

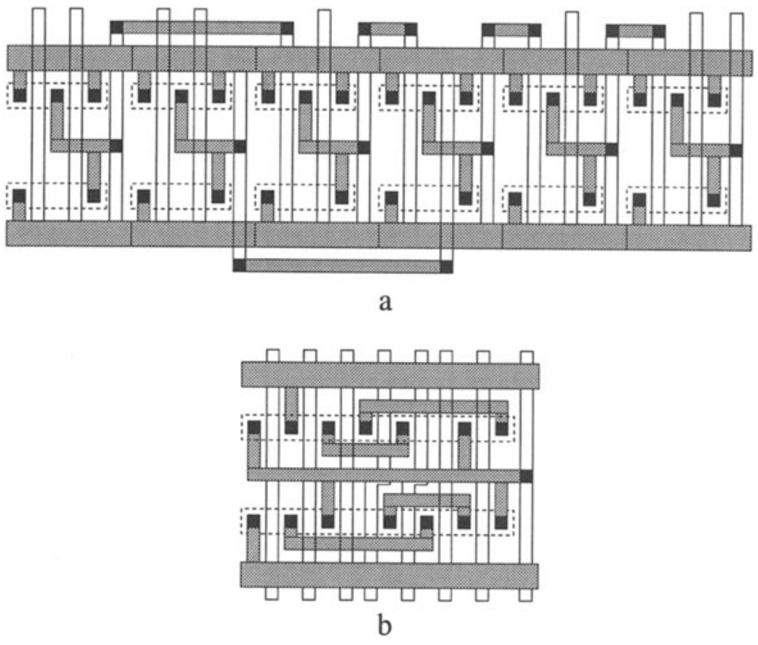


**Figure 3** Layout of the logic function $f = \overline{g \cdot (a + (b + c) \cdot (e \cdot f + d))}$: a) using two-input NAND gates; b) using a complex gate.

The examples of table 1 have been synthesized through the multilevel logic optimizer MIS [5]. The targets have been to optimize for area and map onto a $2\mu$m standard cell library. The combinational logic together with registers are then placed and routed using the Mentor Graphics tool AutoCell. The results are included in table 2.

## 3 Synthesizing for compiled cells

Synthesis for compiled cells allows for generating *any* logic function as a single composite gate. Thus, the advantages of such an approach is the ability to combine several simple logic functions into one complex gate, thereby reducing the area as well as fine-tuning each cell to meet both local and global constraints such as timing aspects and interconnection considerations. An example can be seen in figure 3.

Although a cell compiler is in principle capable of realizing an infinite library of cells, it is in practice limited by the technology, i.e., the stack size restriction (N,P). The stack size is a parameter that indicates the maximum number of series-connected NMOS and PMOS transistors, respectively. The motivation for using compiled cells is partly the large number of different cells, even for small stack size restrictions. For instance, a stack size restriction of (4,4) may create 3.503 different cells, and one of (5,5) may produce 425.803 different cells, while a stack size restriction of (6,6) results in 154.793.519 different cells. Furthermore, each logic function may be implemented by several different cells, i.e., these cells are representing the same logic function but are topologically different, which influences both area and performance. Even though a logic function may be mapped onto a series/parallel transistor network using a straightforward mapping, this mapping may not result in the best solution, since the mapping of a logic function onto a series/parallel network is a one-to-many mapping. Thus, in order to fully exploit the capabilities of using cell compilers, advanced mapping techniques have to be used. Such techniques for technology mapping have only been addressed by a few researchers [4, 1].

An experimental system capable of synthesis for compiled cells has been built by combining the flexible mapper FM [17] with the cell compiler CELLO [15]. The basic idea behind the mapper is to integrate the global transistor sizing for performance with the step in multilevel logic synthesis that generates the actual gate-level implementation, i.e., the technology mapping step. While this integration is only to a limited degree possible in traditional semi-custom implementation of multilevel logic, it seems obvious to combine these two steps when implementing multilevel logic using cell compilers. Such layout compilers for full custom layout not only allow geometric level optimizations for area efficiency; they can also size transistors individually as well as change the topology of the netlist. Changing the topology is only possible if no timing optimization has been performed by the mapper, i.e., selecting transistor positions according to the arrival of critical signals and individually sizing transistors. These aspects may be handled by the cell compiler through specifying the transistor netlists as *fixed* or *dynamic*.

All the examples of table 1 have been synthesized through this experimental synthesis system. A detailed description can be found elsewhere [16].

An important question about the compiled cells approach is whether it is likely to find a sufficient number of complex gates to make the approach worthwhile. The results from synthesizing the examples of table 1 using a stack size restriction of (3,3) have shown that using an advanced mapper does result in a high number of complex gates, i.e., gates representing a logic depth larger than 2.

The number of complex gates were in the range of 11%–38% (22.8% on average). Also, savings of 30% in both gate and transistor count using a stack size restriction of (3,3) have been reported [1].

### *Influence of stack size restriction*

The result produced by the compiled cell approach is very sensitive to the stack size restriction (N,P). Results of synthesizing the examples of table 1 have shown that the stack size restriction has a significant impact on the resulting area [16]. The results from synthesis using three different stack size restrictions—(3,3), (6,3), and (6,6)—have shown area savings in the range of 6%–27%. However, all of these examples were generated using fixed netlists, i.e., netlists that had been optimized by the mapper. As argued previously, this imposes severe restrictions on the cell compiler, resulting in cell areas that may be far from optimum. This impact is of course dependent on the complexity of the cells.

### *Dynamic netlists*

In order to explore the effect of having dynamic netlists, some of the examples were recompiled using the dynamic netlist option, allowing the cell compiler to rearrange the netlist topology to obtain better area solutions. This results in area savings in the range of 10%–20% compared to a fixed netlist. Thus, selecting the right stack size restriction and being able to choose the right transistor netlist topology may give area savings of 33%.

### *A comparison with the standard cell approach*

Another important issue is how good these results are compared to the standard cell solution. Table 3 shows a comparison of the best results from the different (N,P) settings and a standard cell solution. Since different logic synthesizers have been used, it is difficult to make a fair comparison between the two approaches. However, it seems to be clear that the different methods produce quite different results. There is a tendency that compiled cells produce better results for the small benchmark examples. This may be because the cell compiler used for generating results using fixed netlists does not allow feed-throughs over the cells, an aspect that is very important in the larger examples. Another aspect is that the place-and-route system is targeted toward standard cells; thus, it is not able to utilize fully the advantage of the compiled cells, e.g., pin swapping among all pins of a cell. Finally, it has to be noted that the

| FSM | Area Standard cells | Area Compiled cells | |
|---|---|---|---|
| | | fixed | dynamic |
| dk512 | 174616 | 1.04 | 0.87 |
| e84cd_utan | 2114444 | 1.11 | 1.12 |
| ex451 | 78720 | 0.89 | — |
| lee2601c | 62832 | 0.76 | — |
| newpuls | 232560 | 1.08 | 1.06 |
| visa_utan | 116272 | 0.98 | 0.88 |

**Table 3** Comparison of results from both standard cell and compiled cell synthesis.

results produced by the place-and-route system in general are very sensitive to the selected aspect ratio.

## 4 FORMAL VERIFICATION OF FINITE STATE MACHINES

Even within an automatic synthesis environment, the role of verification should not be underestimated: the synthesis programs are often rather complex and likely to still contain some bugs. Moreover, when at some point a designer interferes manually, the "correct-by-construction" claim might be violated. It is therefore standard practice to check the outcome of synthesis against its input. Until recently, simulation was the only means to establish, at least partially, the correctness of a design. New approaches are offered by formal verification methods. In contrast to simulation, these methods are able to prove mathematically the equivalence of a specification and its implementation in a reasonable time. Here, we will examine the possibilities of verifying finite state machine (FSM) designs using a linear-time temporal logic called PTL (propositional temporal logic).

A somewhat related approach that has been applied successfully is the so-called implicit enumeration technique [10] based on exploring the state space in a breadth-first manner and thereby treating sets of states collectively. In such a tool, sets of states are conveniently represented by a binary decision diagram (BDD) data structure. However, it has been shown [8] that a proof checker for PTL can be implemented using basically the same techniques, and

```
<PTL-formula> ::= <formula> { <impl-part> | <equiv-part> }.
<impl-part>   ::= "->" <formula>.                      /* Implication */
<equiv-part>  ::= "<->" <formula>.                     /* Equivalence */
<formula>     ::= <term> { "V" <term> }.                        /* Or */
<term>        ::= <factor> { [ "^" ] <factor> }.               /* And */
<factor>      ::= <primary> { "U" <primary> }.               /* Until */
<primary>     ::= <atom> [ "'" ]                               /* Not */
               |  "~" <primary>                                /* Not */
               |  "@" <primary>                               /* Next */
               |  "<>" <primary>                          /* Sometime */
               |  "[]" <primary>.                           /* Always */
<atom>        ::= <variable>
               |  "True" |  "False"
               |  "(" <PTL-formula> ")".
<variable>    ::= C-style identifier.
```

**Figure 4** PTL program input syntax.

we therefore feel that PTL is somewhat more versatile. We will show how FSMs can be specified in terms of temporal logic formulas. In this way, a satisfiability checker for temporal logic can be used to provide the answers.

## 1 Notational preliminaries

A finite state machine is a quintuple $(Q, \Sigma, \delta, q_0, F)$ with $Q$ being a finite nonempty set of states, $\Sigma$ the finite nonempty input alphabet, $\delta$ the transition mapping, $q_0$ a start state, and $F$ the set of accepting states. In terms of a logic circuit, we usually let the input symbols correspond to data bits on a number of input lines and the states correspond to the values contained in the registers. In hardware applications, it is customary to introduce two derived machine concepts, known as the Moore and Mealy type machines. A (deterministic) Moore machine is described by a six-tuple $(Q, \Sigma, \delta, Init, \Gamma, \phi)$, where $Init$ is a set of initial states, $\Gamma$ the output alphabet, and $\phi : Q \longrightarrow \Gamma$ the output function. In a Mealy machine, outputs are associated with the edges in the state diagram; so we have $\phi : Q \times \Sigma \longrightarrow \Gamma$. We will call a machine incompletely specified if the $\delta$ function is not fully defined over its domain of states and symbols.

Figure 4 presents the syntax of the propositional temporal logic we use in Backus-Naur form, and also hints at the intended semantics [13]. The truth of a temporal formula is defined with respect to a so-called model, which is an infinite sequence of states. Each state is characterized by a subset of atomic propositions to be true in that state. The basic method to define a FSM in temporal logic is to associate the states of the machine with the states in the model, and let a transition coincide with a step in time. We will use `typewriter` font to denote temporal formulas in the syntax accepted by our satisfiability checker program. Names for states and symbols of a FSM will be written in *italic*, using subscripts when appropriate. We prefer to leave the `^` (and) operator of PTL implicit. An operator applied to a set of operands is to be understood as the reduction of the operator over the operands: for example, $\bigvee\{v_i\} = v_1 \vee v_2 \vee \cdots \vee v_n$.

## 2 Transformation from FSM to PTL

To describe a fully specified (deterministic) finite state machine without output in PTL, we first introduce a propositional variable for each input symbol and one for each state. Our interpretation for the variables is such that when a variable is assigned true, that symbol (state) is the machine's current symbol (state); the symbols (states) are mutually exclusive. This may be compared with a one-hot encoding scheme. In PTL, the mutual exclusion of variables `v0, v1, v2, ..., vn` can be expressed by `[]( v1 v2' ...vn' V v1' v2 ...vn' V` $\cdots$ `V v1' v2' ...vn)` or in short-hand notation: `[] E! ( v1, v2, v3, ..., vn )`.

Our $\delta$ mapping in this case is a total function $\delta : Q \times \Sigma \longrightarrow Q$. We write a clause for each state/symbol pair: `(Qi Ik -> @<`$\delta(q_i, i_k)$`>)`, where `Qi` is the PTL variable corresponding to the state $q_i$, `Ik` is the variable associated with the input symbol $i_k$, and `<`$\delta(q_i, i_k)$`>` stands for the PTL variable associated with the result of the $\delta$ function when applied to the arguments $q_i$ and $i_k$. All these clauses are and-ed together and put within an always operator. We are also required to state explicitly that only one state variable is true at any time. As above, we include a clause for the mutual exclusion of a set of variables.

The initial state of the machine ($q_0$) can be expressed by the clause `q0 q1' q2' ...qn'`. If necessary, we can introduce a PTL variable, say `Accept`, to denote the fact that we are in a final state: `[]( Accept <->` $\bigvee$`<`$F$`>)`.

To avoid having to explicitly specify the mutual exclusivity of the states, we can use a predecessor approach in representing the $\delta$ function, in the sense that we define clauses `(@Qi <->` $\bigvee_{i_k}$ `(` $\bigvee$`<`$\delta^{-1}(q_i, i_k)$`> Ik))`, where `<`$\delta^{-1}(q_i, i_k)$`>` denotes the set of PTL variables corresponding to the states that have transitions labeled $i_k$ ending in the state $q_i$.

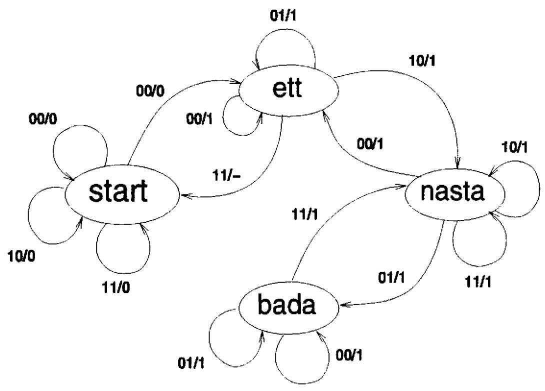


**Figure 5** Lion cage state diagram.

Incompleteness of a machine is resolved by introducing a special state. Whenever $\delta$ is undefined in a state, we add edges labeled with the missing symbols and directed toward that special state. The special state itself has an outgoing transition for each input symbol ending on itself. In an incompletely specified Mealy machine, the output function depends both on the current state and the current input. We then need to add a clause for each binary output signal stating for what state/input-conditions it is true.

Now we present some examples illustrating the method described above.

## *Lion cage*

The lion cage machine is a simple four-state example [7]. Note that the `I1=1`, `I2=0` transition for state **bada** is not specified. Also two transitions have output don't cares (see figure 5).

```
/* State transition table (not fully specified):  */
[]( (@start <-> start (i1' i2' V i1 i2' V i1 i2) V ett i1 i2)
    (@ett   <-> start i1' i2 V ett (i1' i2' V i1' i2) V  nasta i1' i2')
    (@nasta <-> ett i1 i2' V nasta (i1 i2' V i1 i2) V bada i1 i2)
    (@bada  <-> nasta i1' i2 V bada (i1' i2' V i1' i2)) )
/* Output function (not fully specified):  */
[]( VARNING' <- start (i1' i2' V i1 i2' V i1 i2) )
[]( VARNING  <- ett (i1' i2' V i1' i2 V i1 i2')
          V nasta V bada (i1' i2' V i1' i2 V i1 i2) )
/* Initial state:  */  start ett' nasta' bada'
/* Input restriction:  */  []~(bada i1 i2')
```

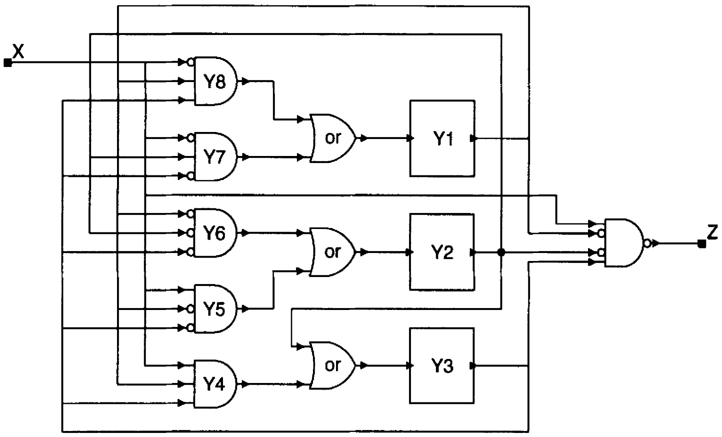


**Figure 6** BCD recognizer.

## *BCD recognizer*

The example of the four-bit BCD recognizer is taken from Pierre [19]. A possible implementation is depicted in figure 6. It can be expressed in PTL as follows:

```
/* Input :  X; 4 bits form BCD sequence, most-significant last.  */
/* Output:  Z. */
[]( /* The logic:  */                    /* The registers:  */
    (Y8 <->  X' Y1  Y3 )                (@Y1 <-> Y7 V Y8)
    (Y7 <->  X' Y2  Y3')                (@Y2 <-> Y5 V Y6)
    (Y6 <-> Y1' Y2' Y3')                (@Y3 <-> Y2 V Y4)
    (Y5 <->  X  Y1' Y3')
    (Y4 <->  X  Y1  Y3 )
    ( Z <-> (X  Y1' Y2' Y3)') )
/* Initial state of registers:  */ (Y1 <-> 0) (Y2 <-> 0) (Y3 <-> 0)
```

It is verified against a Moore state machine [19]:

```
/* States:  S0, S1, S2, S3, S4, S5, Success, Failure.  */
/* Inputs:  X */
/* Transition table (fully specified):  */
[]( (@S0 <-> False)              (@S1 <-> S0 V Success V Failure)
    (@S2 <-> S1 X')              (@S3 <-> S2 X')
    (@S4 <-> S1 X)               (@S5 <-> S2 X V S4)
    (@Success <-> S3 V S5 X')  (@Failure <-> S5 X) )
/* Initial state:  */ S0 S1' S2' S3' S4' S5' Success' Failure'
```

The equivalence test in this case is somewhat complicated; in fact, we are comparing a Mealy machine implementation (the circuit) against a Moore machine specification. In the circuit, `Z` is valid after the first three clock ticks and from then on after every fourth tick. In the state machine, `Success` might be reached after every fourth clock tick. We therefore introduce a simple modulo-four ring counter to serve as a time base:

```
[]( (@T0 <-> T3) (@T1 <-> T0) (@T2 <-> T1) (@T3 <-> T2) )
/* Initial state:  */  T0 T1' T2' T3'
```

And-ing all PTL descriptions together must imply the following test: "*At all times when* `Z` *is valid, the machine's next state is either* `Success` *or* `Failure`, *and it is* `Success` *when* `Z = 1`, *otherwise* `Failure`," which is expressed as: `[](T3 <-> Z @Success V @Failure)`.

It took our program only 0.1 seconds (on an HP9000/S750 workstation) to verify this statement. A 52 CMOS-transistor + 2 flip-flops circuit (not discussed here) was verified against an 11-state Mealy machine in 1.1 seconds.

## 5 VERIFICATION OF IMPLEMENTATION

Verification of CU implementation against specification consists of proving that the layout of the CU is functionally correct with respect to the specification. The specification can be a state graph, and the implementation can be the layout of a PLA and a set of state registers.

To be able to use mathematical proof techniques, some formalism for describing both specification and implementation has to be used. PTL, described in section 4, is one such formalism. In PTL, the specification is naturally described as an STG, but the implementation cannot be described directly as layout. Instead, transistor netlist extraction is typically used to get a switch-level description of the circuit layout. This type of description still has the disadvantage of containing too many electrical properties. A verification tool starting from this description must be able to understand domino logic, pseudo NMOS, dynamic storage elements, asynchronous circuits, etc.

A description more suitable for formal verification is a combination of boolean equations and state registers (see the BCD recognizer example in section 4). To derive such a description from layout is not a trivial task, however, and is the topic of this section. A more extensive description of the algorithms described can be found elsewhere [3].

## 1 Extracting logic equations and memory elements

A transistor network can be seen as consisting of clocked memory elements and combinational logic, which can be described as logic equations. Extracting these equations can be done by constructing a boolean expression for each node in the transistor network that fully describes the state of that node. This expression can be found by traversing all transistor paths from the node to *Vdd* and *Gnd,* thereby finding the conditions for the node being driven high or low.

If a small number of restrictions are put on the implementation, it is also possible to identify storage elements using the extracted logic equations. The restrictions are basically that the transistor network should be synchronously clocked and free from race conditions, restrictions that are met in most designs.

The memory elements are identified in the equations by characteristics common to all memory elements. The first criterion is that the loading of a value into a memory element must be controlled by the clock; the second, that the element must retain its value when not clocked. Whether a node is controlled by the clock is easily determined in the node equation. The second criterion is true if the node equation contains a reference to the node itself, and this self reference is enabled by the clock. Having identified clocked memory elements, a partition of the net into memory elements and strictly combinational parts can be found. From this information, a description in PTL can easily be derived.

## 2 Closing the design loop

The method described above was implemented and verified on a variety of test examples. These examples were implemented with different module generators and cell libraries, representing a variety of design styles, thereby showing the feasibility of the method using real design examples [3]. Using this tool in combination with the PTL verifier, it is possible to close the design loop as indicated in figure 1. Automatic verification of control unit implementation from STG specifications all the way to layout is a reality [21].

## 6 CONCLUSION

Exploring the design space both at a high level, such as the presented architecture exploration, and at a lower level, such as in the compiled cell approach, has been shown to give significant gains in implementation area as well as in timing. Combined with the formal verification method, a *reliable* control unit synthesis system can be built that better meets the demands of high-level synthesis systems.

## REFERENCES

[1] P. Abouzei, R. Leveugle, and G. Saucier. Logic synthesis for automatic layout. In *Proceedings of the WG 10.5 IFIP Workshop on Synthesis, Generation and Portability of Library Blocks for ASCIS Design*, Grenoble, France, pages 139–147, Mar 1992.

[2] R. Amann and U. G. Baitinger. New state assignment algorithms for finite state machines using counters and multiple-PLA/ROM structures. In *Proc. of the IEEE International Conference on Computer-Aided Design*, ICCAD-87, Santa Clara, CA, 1987.

[3] P. Andersson and K. Ranerup. *Sequential logic extraction from CMOS transistor networks.* Technical report, Esprit project BRA 3281, LU/m24/E1/2.

[4] M. R. C. M. Berkelaar and J. A. G. Jess. Technology mapping for standard cell generators. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, ICCAD-88, Santa Clara, CA, 1988.

[5] R. K. Brayton et al. Multiple-Level Logic Optimization System. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, ICCAD-86, Santa Clara, CA, 1986.

[6] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. L. Sangiovanni-Vincentelli. *Logic minimization algorithms for VLSI synthesis.* Kluwer Academic Publishers, 1984.

[7] B. Breidegard. Private communication. Lund University, Sweden, Oct 1989.

[8] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. *1991 International Workshop on Formal Methods in VLSI Design.*

[9] Misha R. Burich. Design of module generators and silicon compilers. In D. Gajski, editor, *Silicon Compilation*, Addison-Wesley, 1988.

[10] O. Coudert, C. Berthet, and J. C. Madre. Formal boolean manipulations for the verification of sequential machines. In *Proc. European Design Automation Conference*, Glasgow, Scotland, pages 57–61, Mar 990.

[11] G. De Micheli. Synthesis of control systems. In G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti, editors, *Design systems for VLSI Circuits, Logic Synthesis and Silicon Compilation*, Martinus Nijhoff, pages 327–364, 1987.

[12] S. Devadas and A. R. Newton. Decomposition and factorization of sequential finite state machines. In *Proc. of the IEEE International Conference on Computer-Aided Design*, ICCAD-88, Santa Clara, CA, 1988.

[13] G. L. J. M. Janssen. Hardware verification using temporal logic: a practical view. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification, VLSI Design Methods-II*, Elsevier Science Publishers B.V. (North-Holland), pages 159–168, 1990.

[14] J. Kukula and S. Devadas. Finite state machine decomposition by transition pairing. In *Proc. of the IEEE International Conference on Computer-Aided Design*, ICCAD-91, Santa Clara, CA, 1991.

[15] J. Madsen. A New Approach to Optimal Cell Synthesis. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, ICCAD-89, Santa Clara, CA, pages 336–339, 1989.

[16] J. Madsen and B. Hald. Controller Synthesis using Compiled Cells. In *Proceedings of the Tenth NORCHIP Seminar*, Helsinki, Finland, pages 36–43, Nov.1992.

[17] H. Pallisgaard. Principles of technology mapping for CMOS functional cell generators. In *Proceeding of the ASCIS Open Workshop on Controller Synthesis*, Technical University of Denmark, Lyngby, Denmark, Sep 1991.

[18] H. Pallisgaard, O. C. Andersen, L. Linqvist, and J. Madsen. Controller synthesis in Gaia, a VHDL RT-level framework. In *Proceedings of the Second European Conference on VHDL Methods*, EUROVHDL-91, Stockholm, Sweden, pages 78–85, Sep 1991.

[19] L. Pierre. The formal proof of sequential circuits described in CASCADE using the Boyer-Moore theorem prover. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification, VLSI Design Methods-II*, Elsevier Science Publishers B.V. (North-Holland), pages 309–328, 1990.

[20] K. Ranerup. Control architecture selection from state graph characteristics. In *Proceeding of the ASCIS Open Workshop on Controller Synthesis*, Technical University of Denmark, Lyngby, Denmark, Sep 1991.

[21] K. Ranerup. *Proving functional correctness of control unit layout in silicon compilation environments.* Technical report, Esprit project BRA 3281, LU/m30/E1/4.

[22] K. Ranerup and J. Madsen. Comparision of logic synthesis methods in control unit architecture synthesis. Technical report, Esprit project BRA 3281, CD/m30/E1-E2/1.

[23] K. Ranerup and L. Philipson. *Optimization of finite state machines using subroutines.* Technical report, Department of Computer Engineering, Lund University, Sweden, 1989.

# INDEX