Field-Programmable Custom Computing Technology

Architectures, Tools, and Applications

edited by

Jeffrey Arnold
Wayne Luk
Ken Pocek

Springer Science+Business Media, LLC

FIELD-PROGRAMMABLE CUSTOM COMPUTING TECHNOLOGY: ARCHITECTURES, TOOLS, AND APPLICATIONS

edited by

Jeffrey Arnold
Adaptive Silicon, Inc.

Wayne Luk
Imperial College

Ken Pocek
Intel

SPRINGER SCIENCE+BUSINESS MEDIA, LLC

المنارة للاستشارات

www.manaraa.c

Journal of VLSI
SIGNAL PROCESSING SYSTEMS
for Signal, Image, and Video Technology

Volume 24, Nos. 2/3, March 2000

Special Issue: VLSI on Custom Computing Technology
Guest Editors: Jeffrey Arnold, Wayne Luk and Ken Pocek

Guest Editors' Introduction .... *Jeffrey Arnold, Wayne Luk and Ken Pocek* 1

Pipeline Reconfigurable FPGAs .... *Herman H. Schmit, Srihari Cadambi, Matthew Moe and Seth C. Goldstein* 3

Design and Implementation of the MorphoSys Reconfigurable Computing Processor .... *Ming-Hau Lee, Hartej Singh, Guangming Lu, Nader Bagherzadeh, Fadi J. Kurdahi, Eliseu M.C. Filho and Vladimir Castro Alves* 21

Co-Synthesis to a Hybrid RISC/FPGA Architecture .... *Maya B. Gokhale, Janice M. Stone and Edson Gomersall* 39

Design-Space Exploration for Block-Processing Based Temporal Partitioning of Runtime Reconfigurable Systems .... *Meenakshi Kaul and Ranga Vemuri* 55

Application of Reconfigurable CORDIC Architectures .... *Oskar Mencer, Luc Séméria, Martin Morf and Jean-Marc Delosme* 85

A Configurable Logic Based Architecture for Real-Time Continuous Speech Recognition Using Hidden Markov Models .... *Panagiotis Stogiannos, Apostolos Dollas and Vassilis Digalakis* 97

The CAM-Brain Machine (CBM): Real Time Evolution and Update of a 75 Million Neuron FPGA-Based Artificial Brain .... *Hugo de Garis and Michael Korkin* 115

المنارة للاستشارات

www.manaraa.c

ISBN 978-1-4613-6988-2 ISBN 978-1-4615-4417-3 (eBook)
DOI 10.1007/978-1-4615-4417-3

Library of Congress Cataloging-in-Publication Data

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

Copyright © 2000 by Springer Science+Business Media New York
Originally published by Kluwer Academic Publishers in 2000
Softcover reprint of the hardcover 1st edition 2000

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, mechanical, photo-copying, recording, or otherwise, without the prior written permission of the publisher, Springer Science+ Business Media, LLC

Printed on acid-free paper.

المنارة للاستشارات

www.manaraa.c

Journal of VLSI Signal Processing Systems 24, 127, 2000.
© 2000 Kluwer Academic Publishers. Printed in The Netherlands.

Guest Editors' Introduction

This issue contains a collection of seven selected papers on custom computing technology. In particular, it describes the latest advance in architectures, design methods, and applications of field-programmable devices for high-performance reconfigurable systems. This journal is among the first to devote special issues to this topic (Volume 6, Number 2 and Volume 12, Number 5); recent special issues on configurable computing (IEEE Computer Magazine) and reconfigurable systems (IEE Proceedings), as well as the thriving conferences on similar themes such as FCCM, FPGA and FPL, demonstrate the recognition of its increasing importance.

The first two papers are focused on architectures for custom computing. Schmit, Cadambi, Moe and Goldstein explain the idea of hardware virtualisation, and show how it can be supported by a pipeline architecture. In addition to providing high performance, their architecture is designed to benefit the hardware compiler and to take advantage of improvements in silicon technology.

The second paper, by Lee, Singh, Lu, Bagherzadeh, Kurdahi, Filho and Alves describes the implementation of an architecture consisting of configurable logic resources and a RISC processor. Simulation results indicate that this system can achieve better performance than existing ones, particularly for data-parallel applications such as video compression and automatic target recognition.

The next two papers cover design methods and tools for custom computing systems. Gokhale, Stone and Gomersall report in the third paper a pragma-based approach to programming architectures similar to that of Lee et. al., which consists of configurable logic and a conventional processor. The pragma directives specify information such as location of data and computation, which can be used by a compiler to produce a program for the conventional processor and a configuration bit stream for the configurable logic.

The fourth paper, by Kaul and Vemuri, presents a method for synthesising reconfigurable designs based on temporal partitioning and design space exploration. Techniques such as block processing and iterative constraint satisfaction are used judiciously to optimise both large and small problems.

The last three papers demonstrate the use of custom computing technology for various applications. Mencer, Semeria, Morf and Delosme indicate in the fifth paper how reconfiguration enables CORDIC units to be adapted to specific application requirements. An adaptive filter example is used to illustrate the reduction in latency and in area of the resulting pipelined implementation.

The sixth paper, by Stogiannos, Dollas and Digalakis, describes an architecture for real-time continuous speech recognition based on a modified hidden Markov model. Of particular interest is the exploitation of reconfiguration for optimising the design which involves, for instance, adapting arithmetic precision and pipeline depth to run-time characteristics.

The final paper, by de Garis and Korkin, explores a reconfigurable machine for implementing a genetic algorithm which contains up to 75 million neurons. This machine has been developed to control the real-time behaviour of a robot kitten.

We thank the contributors to this special issue and the reviewers who completed the reviews under a tight schedule. The advice and encouragement of Professor Sun Yuan Kung are gratefully acknowledged. We also thank the staff at Kluwer, particularly Jennifer Evans, Carl Harris and Sharon Palleschi, for their assistance.

Jeffrey Arnold
Adaptive Silicon, Los Gatos

Wayne Luk
Imperial College of Science, Technology and Medicine, London

Ken Pocek
Intel, Santa Clara

المنارة للاستشارات
www.manaraa.c

Journal of VLSI Signal Processing Systems 24, 129–146, 2000.
© 2000 Kluwer Academic Publishers. Printed in The Netherlands.

Pipeline Reconfigurable FPGAs*

HERMAN H. SCHMIT, SRIHARI CADAMBI AND MATTHEW MOE
Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA

SETH C. GOLDSTEIN
Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

Abstract. While reconfigurable computing promises to deliver incomparable performance, it is still a marginal technology due to the high cost of developing and upgrading applications. Hardware virtualization can be used to significantly reduce both these costs. In this paper we describe the benefits of hardware virtualization, and show how it can be achieved using the technique of pipeline reconfiguration. The result is PipeRench, an architecture that supports robust compilation and provides forward compatibility. Our preliminary performance analysis on PipeRench predicts that it will outperform commercial FPGAs and DSPs in both overall performance and in performance normalized for silicon area over a broad range of problem sizes.

1. Introduction

Components in a signal processing system are typically implemented in one of two ways: (1) custom hardware or (2) software running on a processor. The advantage of implementation in hardware is that it can exploit the correct amount of parallelism in order to meet performance constraints while minimizing either the power or per-unit cost of the system. The chief problem with hardware implementations of signal processing components is the time and money consumed by the many steps of the design process and the high non-recoverable costs of fabrication. As a result of these high costs, hardware solutions are only feasible in systems that are either cost-insensitive, where the high development cost is tolerated, or systems that are produced in very high volume, where the development cost is absorbed by the lower per-unit cost of a hardware implementation.

Field-programmable Gate Arrays (FPGAs) have enabled the creation of hardware designs in standard, high-volume parts, thereby amortizing the cost of mask sets and significantly reducing time-to-market for hardware solutions. However, engineering costs and design time for FPGA-based solutions still remain significantly higher than software-based solutions. Designers must frequently iterate the design process in order to meet system performance requirements while simultaneously minimizing the required size of the FPGA. Each iteration of this process takes hours or days to complete.

Another way to reduce the effective costs of hardware design would be to frequently re-use hardware components in multiple systems. However, hardware designs are difficult to port to different process technologies. Furthermore, it is inefficient or impossible to re-use a component in a system that requires significantly more or less performance than the original component, because the parallelism exploited by a component is fixed by the original designer.

In this paper, we describe a technique, called *hardware virtualization*, that solves the problem of hardware re-use. We present techniques to virtualize pipelined applications using existing FPGA architectures. Based on the shortcomings of these techniques, we present a new method of hardware reconfiguration, called *pipeline reconfiguration*, that enables efficient hardware virtualization for pipelined applications.

*This work supported by DARPA, under contract DABT63-96-C-0083.

المنارة للاستشارات
www.manaraa.c

1.1. Hardware Virtualization

Hardware virtualization frees a designer to create a hardware design that exploits a very large amount of parallelism but also consumes a great deal of silicon area. This large hardware design can be emulated on a much smaller amount of physical hardware at a reduced level of performance. The emulation of the large design (or *virtual hardware design*) is accomplished by time-multiplexing programmable hardware.

The closest analog to the ideal of virtual hardware is virtual memory in processor systems. In virtual memory, a small physical memory is used to emulate a large logical memory by moving infrequently accessed memory into slower cheaper storage media. This has numerous advantages for the process of software development. First, neither programmers nor compilers need know exactly how much physical memory is present in the system, which speeds development time. Second, different systems, with different amounts of physical memory can all run the same programs, despite different memory requirements. A small physical memory will limit the performance of the system, but if this performance is unacceptable, the user simply buys more memory. Furthermore, since the price of memory is ever decreasing, newer systems will have more memory and therefore the memory performance of legacy software will improve until these programs fit entirely into the physical memory in the system.

Similarly, an ideal virtualized FPGA would be capable of executing any hardware design, regardless of the size of that design. The execution speed would be proportional to the physical capacity of FPGA, and inversely-proportional to the size of the hardware design. Because the virtual hardware design is not constrained by the FPGA's capacity, generation of a functional design from an algorithmic specification would be much easier than for a non-virtual FPGA and could be guaranteed from any legal input specification. Optimizing the virtual hardware design would result in faster execution, but would not be required to initially implement or prototype the application. Thus, hardware virtualization enables FPGA compilers to more closely resemble software compilers, where unoptimized code generation is extremely fast, and where more compilation time can be dedicated to performance optimization when necessary. This accompanying benefit to hardware virtualization is called *robust compilation*.

A family of virtualized FPGAs could be constructed that all share the ability to emulate the same virtual hardware designs, but that differ in physical size. The members of this family with larger capacity will exhibit higher performance because they emulate more of the virtual design at any one time. Future members of this family, built in newer generations of silicon, could emulate virtual hardware designs at higher levels of performance *without redesign*, much like the way microprocessor families run binaries from previous generations without re-compilation. This benefit, which we call *forward-compatibility*, increases the return on investment in FPGA applications. In other words, the expense of generating (or purchasing) virtual hardware designs can be amortized for many systems with different performance and cost requirements, over multiple generations of silicon.

1.2. Pipeline Reconfiguration

This paper focusses on the virtualization of hardware applications that can be formulated as a pipeline. A pipeline is a systolic array [1] where all data flow goes in one direction and there is no feedback. Figure 1 illustrates two stages of a FIR filter application transformed in such a way to meet these requirements. Transforming algorithms into pipelines is a well-understood problem. Some of the techniques for transforming algorithms into pipelined implementations are presented in [2–4]. Fortunately, a large percentage of computationally challenging applications can be implemented as pipelines, including many in the domains of three-dimensional rendering, signal and image processing,

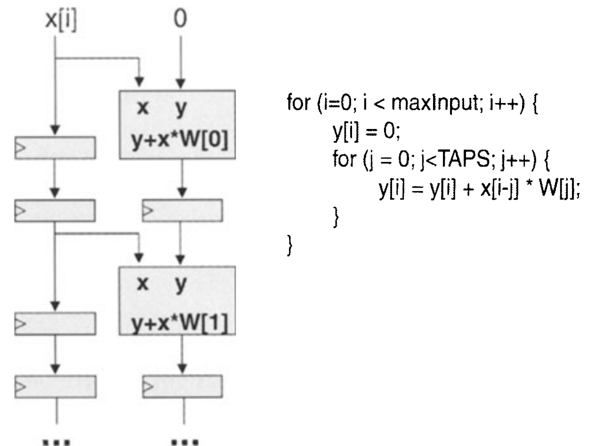


Figure 1. A pipelined FIR filter. All wires propogate in a single forward direction. One additional benefit to this design is that all multiplications have one constant operand, allowing further hardware optimization.

المنارة للاستشارات
www.manaraa.com

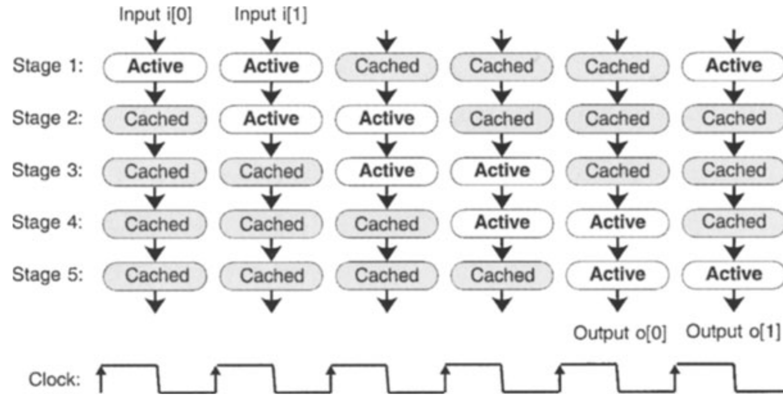


Figure 2. Pipeline reconfiguration. An example of mapping a five stage pipeline onto a FPGA with the ability to hold two stages.

and cryptography. Furthermore, extremely fine-grained pipelining is the most important technique used by reconfigurable systems to obtain high throughput [5]. If reconfigurable systems become widely used, they will be predominately applied to pipelineable applications.

Pipeline reconfiguration is a new way to use the reconfigurability of FPGAs to virtualize pipelined applications. In pipeline reconfiguration, the configuration bits corresponding to each pipeline stage are brought into the executing FPGA fabric, one stage every cycle. When the FPGA fabric is fully populated by active pipeline stages, older pipeline stages are replaced by newer pipeline stages.

Figure 2 shows an example of pipeline reconfiguration for a five stage pipeline running on an FPGA with a capacity of two active pipeline stages. In this example, there are two results produced every five cycles. The FPGA "scrolls" through the pipelined application, and each run through the application takes five cycles and produces two results. Therefore the throughput of this implementation is two-fifths of the throughput possible without virtualization. The input and output behavior of this implementation is modified from the non-virtualized implementation. Input and output from the virtualized pipeline occurs in two-cycle bursts that repeat every five cycles. Ideally, virtualized FPGA should accomodate this burstiness without requiring the involvement of the pipeline designer.

In Section 2, we first present ways to virtualize pipelines using traditional FPGA reconfiguration techniques. We quantify the latency and throughput of these techniques based on system parameters such as FPGA capacity and reconfiguration time. Then we compare these techniques to pipeline reconfiguration. We show that reconfiguration time is the most important factor in the performance of all these systems. We also show that pipeline reconfigurable devices avoid many of the other problems with traditional reconfiguration, including pipeline fill and empty penalties and memory capacity problems.

In the remainder of the paper, we address a number of architectural challenges for pipeline reconfiguration FPGAs. Each section addresses one of the three significant problems for these architectures. The first problem is reconfiguration time. For maximum performance, a pipeline reconfigurable FPGA should be able to configure a computationally significant pipeline stage in one cycle. Section 3 describes the PipeRench architecture, which is designed to minimize the impact of reconfiguration time on performance. The second problem is how to control the pipeline reconfiguration at run-time in order to accurately virtualize hardware. Section 4 presents the PipeRench configuration controller, which controls the movement of configuration data between storage and active FPGA fabric. The third problem, as illustrated in Fig. 2, is that the schedule of inputs and outputs to the pipeline is dependent on the virtualization and must be determined at run-time. Section 5 presents the PipeRench data controller, which performs these functions. Section 6 presents the estimated performance of PipeRench for a set of pipelined FIR filters and compare to both commercial FPGAs and DSPs, and Section 7 presents some comparisons of pipeline reconfiguration to related work in FPGA and computer architecture.

2. Pipeline Virtualization

In this section, we evaluate methods to virtualize pipelined applications on standard FPGAs using conventional reconfiguration, and we compare it to pipeline reconfiguration in terms of throughput and latency.

المنارة للاستشارات

www.manaraa.com

2.1. Component-Level Reconfiguration

Popular commercial FPGAs such as the Xilinx 4000 family [6] and the Altera FLEX family [7] have exclusive operational and configuration modes. There is no mechanism to allow simultaneous operation and configuration or even partial modification of a configuration that is already loaded. The atomic unit of reconfiguration is the whole chip, therefore the chip is only capable of *component-level reconfiguration*. Configuration data itself is fed into these FPGAs through a small number of I/O pins. Configuration times can therefore be thousands or hundreds of thousands of times longer than the operating cycle time of a design. As we will show, this long configuration time hurts throughput, latency and memory requirement for pipelined application. *Dynamic partial reconfiguration* is a variation on component-level reconfiguration that allows the configuration memory to be written simultaneously with the operation of the chip. It was present in the Xilinx 6200 family [8]. This mode needs to be very carefully used, as it does not prevent the reconfiguration from interfering with the computation on the device.

Another type of configuration mechanism is the *multiple-context configuration*, as discussed in [9–11]. This mechanism is similar to that in a standard FPGA, except that instead of having one configuration stored in the FPGA, n complete configurations are loaded into the FPGA. A global selection bus determines which one of the n configuration should be used during the current cycle. Logical reconfiguration of the entire FPGA can be accomplished in a time comparable to the execution cycle time of the design, but the atomic unit of reconfiguration remains the whole chip.

While multiple-context configuration solves configuration speed problem, it does have limitations. First, the process of switching contexts moves a large amount of configuration data in a short period of time. Context-switching is therefore a power-intensive operation. Furthermore, the amount of "virtual" hardware emulated by a multiple-context FPGA is limited to n times the physical hardware in that FPGA. Reconfiguration beyond n contexts must take place on a low-speed, narrow configuration bus. Finally, as we shall demonstrate, component-level reconfiguration has significant disadvantages for virtualization of pipeline designs.

2.1.1. *The Application.* The application we will examine is a very deeply pipelined application, such as a high-order FIR filter implemented as shown in Fig. 1. Assume that this application has S identically-sized pipeline stages. Further assume that there are D bytes of data flowing between each stage of the filter every cycle, and between the filter input and output. This last assumption rarely holds in real pipelined applications. In most pipelines, the intermediate data between any two stages is much greater than D. This assumption greatly simplifies the following analysis, however.

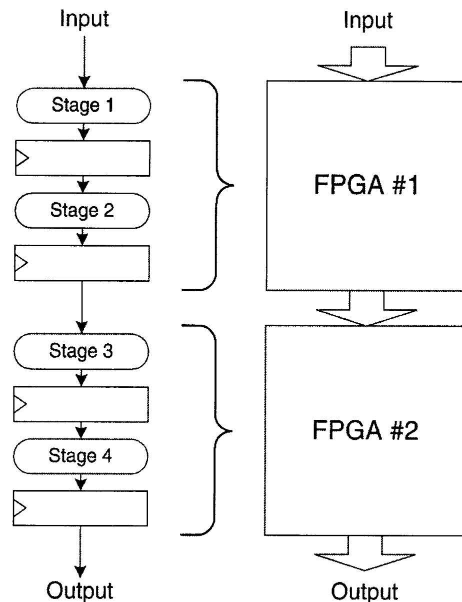


Figure 3. Example pipeline application: Four stages implemented on two FPGAs. $S = 4$ and $N = 2$.

To implement this application, we have FPGAs with a fixed logic capacity. Assume that in order to statically implement the whole filter we would require N of these FPGAs, as illustrated in Fig. 3. If the clock cycle time of the FPGA, as determined by the most complex pipeline stage is T, then the throughput of the static, N-FPGA implementation of this filter is D/T bytes per second. To simplify the analysis, we will not consider the time that it takes to configure this application initially, or to swap between different applications.

2.1.2. *Virtualization.* We will use component-level reconfiguration to implement this filter in one FPGA of similar capacity. The theoretical maximum throughput of the 1-FPGA implementation of this filter using Run-Time Reconfiguration (RTR) is $D/(NT)$ simply due to the reduction in computing hardware. We will examine the implementation of this filter using component-level reconfiguration in terms of its performance characteristics and memory requirements.

المنارة للاستشارات
www.manaraa.com

Using component-level reconfiguration, the N different FPGA configurations from the N-FPGA design sequentially configure a single FPGA. This level of reconfiguration has also been called *Global RTR* [12]. The configuration controller loads one configuration, and allows the FPGA to perform operations on X words of data. It takes S/N cycles to get the first result from this configuration, and $X - 1$ cycles to get the remaining results. Therefore, the time required to complete these computations, in seconds, is:

$$T(X - 1 + S/N) \quad (1)$$

After this computation is complete, the system controller reconfigures the FPGA with the next configuration in the sequence as illustrated in Fig. 4. If it takes C cycles to reconfigure the FPGA, then the throughput of this implementation can be described using the formula:

$$\frac{DX}{NT(X - 1 + S/N + C)} \quad (2)$$

$$= \frac{D}{T\left(N + \frac{S-N}{X} + \frac{NC}{X}\right)} \quad (3)$$

Throughput falls short of the ideal due to the pipeline penalty and a reconfiguration penalty. The pipeline penalty, which is expressed in the $(S - N)/X$ term, is the penalty suffered for having to repeatedly fill and empty the pipeline between reconfigurations. The reconfiguration penalty, which is expressed in the $(NC)/X$ term, is caused by the non-zero reconfiguration time of the device. The relationship of C, X and throughput is shown in Fig. 5. In this graph, $S = 100$ and $N = 10$. The ideal performance of this implementation is $D/(10T)$. The value on the y-axis indicates how actual throughput compares to this ideal. For the moment, assume X is small. (Note: when $X = 10$ the pipeline is just filled, and then emptied.) When C is large, as in the case of standard FPGAs, the throughput is unacceptably low. When C is small, as is the case with the multiple-context FPGAs, the pipeline penalty limits throughput.

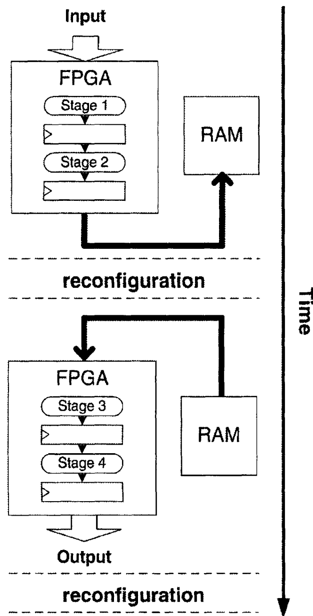


Figure 4. Component-level reconfiguration: Virtualization of pipelined application through reconfiguration of one FPGA with RAM to store intermediate results.

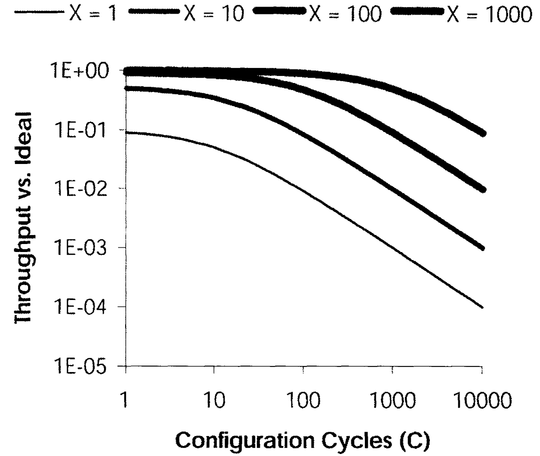


Figure 5. Throughput versus configuration time: Component-level configuration for various values of X. $S = 100$ and $N = 10$. This is a log-log plot.

Increasing X will increase the throughput of the implementation regardless of C, but by increasing X the latency of the implementation is also increased. The latency for this implementation is:

$$NT\left(X + \frac{S}{N} - 1 + C\right) \quad (4)$$

The second problem with increasing X is that it is necessary to have enough memory to store all the data output from one block during reconfiguration so that it can be used as the input to the next block of the pipeline. The required amount of memory is DX.

In addition, assuming input data arrives as a rate not greater than the throughput rate, any virtualized implementation will require a buffer to store inputs while the

www.manaraa.com

Table 1. Commercial FPGA configuration times (clock frequency: 33 MHz).

Part	Config. time	C
XC4028EX [6]	8.35 ms	275,000
XC6216 [8]	92 μs	3036

lower stages of the pipeline are being executed. This buffer would also need to have a capacity of DX bytes.

If X is very large, it will be difficult to meet these memory requirements on the same chip as the FPGA. If this is the case, the time required to access off-chip memory may increase T, degrading performance of the whole system.

Table 1 shows typical values of C for two available Xilinx components using the fastest configuration mode available for that component. These results were computed assuming a modest operating frequency of 33 MHz. Obviously, reconfiguration time is going to play a critical role in determining throughput, latency and memory requirements for applications which use these components.

Multiple-context FPGAs have a C value of one cycle or less. While this effectively eliminates the configuration penalty, it does not reduce the effect of the pipeline penalty. In addition, multiple-context FPGAs only have a low C if the number of contexts held in the device is greater than N for the particular application. If a multiple-context FPGAs can have inactive configurations modified while simultaneously executing another configuration, then it would be possible to extend the virtualization. But this would require X to be large enough to hid the reconfiguration time of the inactive configuration.

2.2. *Pipeline Reconfiguration*

Pipeline reconfiguration is a restricted form of local RTR [12], in which the pipeline is separated into S components, each corresponding to one pipeline stage. The FPGA can hold P of these pipeline stages, and the reconfiguration happens in an incremental manner. In order to normalize the capacity to the previous analysis, $P = S/N$. During each stage of the computation, we add one additional stage to the configuration, and remove (or overwrite) a stage if necessary to keep the amount of configuration within the capacity of the FPGA. Figure 2 illustrates this procedure. Reconfiguration in this manner can be visualized as the scrolling of a window through the computation.

2.2.1. Virtualization. As with component-level reconfiguration, we will assume that the time it takes to execute a pipeline stage is T in seconds, and the number of FPGAs required to hold the whole application is N. The number of execution cycles required to reconfigure the entire FPGA is C. Therefore, the time required to substitute a new pipeline stage into the configuration is, ideally, TC/P. For one complete sweep through the application, S stages must be configured, requiring $TCS/P = TCN$ seconds. Execution of the entire pipeline will take S cycles for the first element of data, and $P - 1$ cycles to process the remaining data in the pipeline. Therefore, the throughput of this implementation is equal to:

$$\frac{DP}{T(S + P - 1 + CN)} \tag{5}$$

$$= \frac{D}{T\left(N + 1 - \frac{1}{P} + (N^2C)/S\right)} \tag{6}$$

The best-case latency of this implementation is:

$$T(S + CN) \tag{7}$$

Figure 6 shows the relationship of throughput to the configuration cycles, C. For comparison, two curves for component-level reconfiguration with $X = 100$ and $X = 10$ are shown. Figure 7 shows the plots of latency for the same three implementations. These graphs show that when C is small, the pipeline reconfigured implementation exhibits both high throughput and low

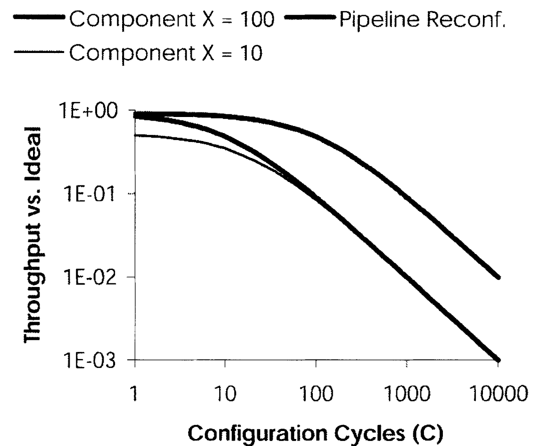


Figure 6. Throughput versus configuration time: Pipeline reconfiguration compared to component-level configuration. $S = 100$ and $N = 10$.

المنارة للاستشارات
www.manaraa.com

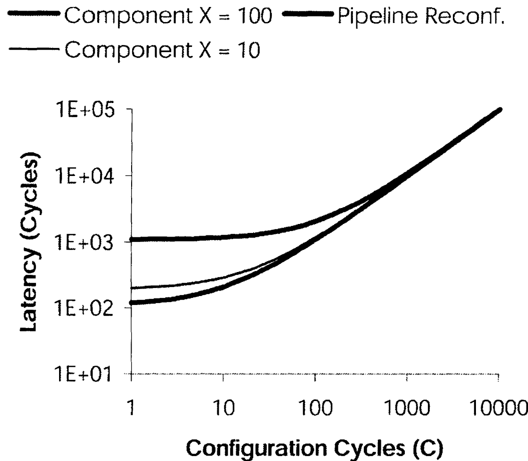


Figure 7. Latency versus configuration time: Pipeline reconfiguration compared to component-level configuration. $S = 100$ and $N = 10$.

latency. When C is large, the throughput exhibited by the pipelined reconfigured design exhibits behavior very similar to the component-level reconfigured implementation with $X = S/N$.

These graphs again demonstrate the importance of configuration cycles, C, in the throughput and latency equations. As C approaches zero, the throughput and latency of the pipeline reconfigured FPGA approach their respective theoretical optima. Component-level reconfigured implementations can only trade throughput for latency, and can therefore never optimize both quantities simultaneously.

Another advantage of pipeline stage reconfiguration is that all intermediate results remain stored in the appropriate pipeline stage. There is no need for supplemental storage. The front-end storage to buffer arriving inputs must still be present, but it needs only store DS/N bytes, as opposed to DX bytes.

The most important characteristic of incremental pipeline reconfiguration is that the presence of more hardware transparently results in higher throughput.

Pipeline reconfiguration requires the ability to modify only a portion of the FPGA at a time. Therefore it is only possible using dynamically reconfigurable FPGAs, such as the Xilinx 6200 [8]. Using the Xilinx 6200 to virtualize pipelines was described in [13]. The primary problem with using pipeline reconfiguration on an on-line reconfigurable FPGA like the XC6200 series is that the relatively low bandwidth of the configuration bus may make the effective value of C quite large. This limitation could be fixed by incorporating an on-chip configuration cache and widening the connection between the memory and the FPGA fabric.

For these reasons, we have designed an FPGA architecture specifically for pipeline reconfiguration, which we call PipeRench. PipeRench is capable of configuring a stage of the pipeline concurrently with the execution of the rest of the pipeline. Because of this concurrency, C effectively equals zero (even though the entire device still requires TP to be configured), and the performance approaches the theoretical maximum. The architecture and operation of PipeRench will be described in the following section.

3. PipeRench Architecture

In order to achieve high-performance and forward-compatibility, a pipeline reconfigurable device must have two architectural features. First, the architecture must support the configuration of a computationally significant pipeline stage every cycle, while concurrently executing all other pipeline stages in the FPGA, i.e. $C = 0$. Second, the architecture must allow different pipeline stages to be placed in different absolute locations in the physical device at different times. Only relative placement constraints should need to be observed, so that a pipeline stage can get its inputs from the previous stage and send its outputs to the subsequent stage. No existing FPGA has these features. This section describes how these features are provided in PipeRench.

In order to configure a pipeline stage every cycle, a pipeline-reconfigurable architecture requires a very high-throughput connection to the configuration memory that stores the virtual hardware design. Configuration storage in PipeRench is on-chip and connected to the FPGA fabric with a wide data bus, so that one memory read will configure one pipeline stage in the fabric. This wide configuration word is written into one of many physical blocks in the FPGA fabric. We call these blocks *stripes*, and they define the basic unit of reconfiguration in the architecture. We use the word stripe to describe both the physical structures to implement the functionality of a pipeline stage (a *physical stripe*), and the configuration word itself (a *virtual stripe*), which may or may not be resident in a physical stripe. Since a virtual stripe can be written into any physical stripe, all physical stripes must have identical functionality and interconnect.

Designing the stripe to provide adequate functionality for a wide range of applications with a limited

المنارة للاستشارات
www.manaraa.com

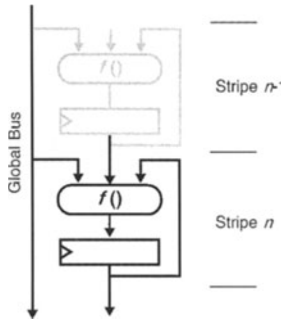


Figure 8. Generalized stripe functionality.

number of configuration bits is a critical and complex task, the description of which is beyond the scope of this paper. In general, the functionality within a stripe can be described as a combinational function of three inputs: the registers within that stripe, the registers from the previous stripe, and a set of global interconnects, as shown in Fig. 8. The combinational function $f()$ is defined by the configuration bits in the virtual stripe.

Note the feedback path from the register in a stripe back into the combinational function $f()$. If this path is used by an application, the register bits that are fed back contain state information that must be maintained by the device. We call this information *stripe state*.

PipeRench is currently envisioned as a coprocessor in a general-purpose computer (see Fig. 9). It is a memory mapped device, and has access to the same memory space as the primary processor. All the virtual stripes for all the applications that are to run on PipeRench are stored in main memory. A PipeRench "executable" consists of configuration words, which control the fabric, and data controller parameters, which determine the application's memory read/write access pattern. The processes of loading the configuration memory and data controllers from off-chip, and configuring the fabric from the configuration memory, are the responsibilities of the configuration controller, described in Section 4.

Figure 10 illustrates two possible layouts for physical stripes. In Fig. 10(a), the virtual stripes move every cycle into a different physical stripe. This has two advantages: the interconnect between adjacent virtual stages is very short, and new virtual stripes are written into only one physical stripe (on the bottom). The chief disadvantage with this layout is that all the configuration data must move every cycle. This is a tremendous power sink, and it reduces performance because now the clock cycle must include the time it takes for the configuration data to move and settle.

An alternative layout is illustrated in Fig. 10(b), which shows the physical stripes arranged in a ring, allowing the configuration to remain stationary. There are two disadvantages to this approach. First, it requires configuration data to be loaded anywhere in the fabric. Second, there is a longer worst-case interconnect between adjacent stripes (at the bottom and the top). But because only one stripe needs to be reconfigured, it is possible to configure that one stripe while simultaneously executing the application in the remaining stripes. In Fig. 10(b), five stripes are computing, despite the

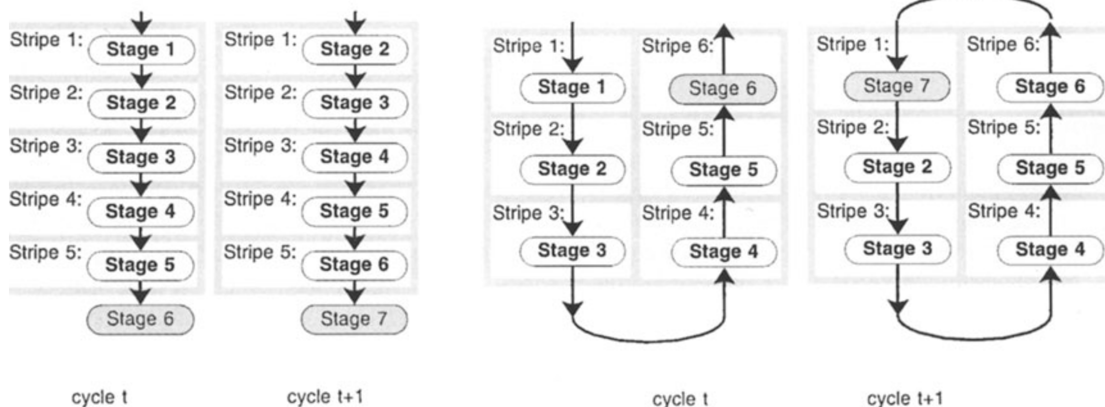


Figure 9. Shifting and stationary configuration.

المنارة للاستشارات
www.manaraa.com

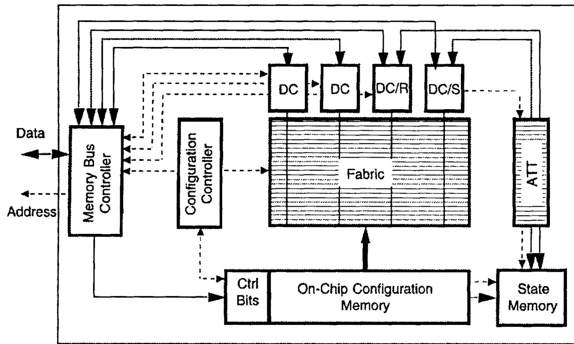


Figure 10. Architecture overview. Sold lines are data paths, dashed lines are address and control paths.

fact that there are six physical stripes in the fabric. We believe that the disadvantages of this approach are outweighed by the power and performance advantages.

There are three types of interconnect necessary for a stripe: intra-stripe, local inter-stripe and global inter-stripe. Intra-stripe routing is used to interconnect the elements of a stripe to create the functionality of the pipeline stage.

Local inter-stripe interconnect receives inputs from the previous stripe and sends outputs to the next stripe in the pipeline. Since this is a pipelined application, and each stripe contains a pipeline stage, there is no need for non-registered interconnect between non-adjacent stripes. It is essential that all local inter-stripe interconnects be registered, and that the configuration bits from one stripe cannot change anything in the path between that stripe's registers and its interconnection to the following stripe. For example, in Fig. 10, the computation in stage 2 at cycle $t + 1$ requires the result of the computation in stage 1 at cycle t. But in cycle $t + 1$ the configuration for stage 1 is being removed from the fabric or overwritten. If a change to the configuration effects the ability of stage 2 to see stage 1's last computation, the results can not be guaranteed.

Global inter-stripe interconnect is used to get operands to any input stripe, get results from any output stripe, and to save and restore the stripe state when it is removed or inserted from the FPGA fabric. The stripe state may also be initialized using the restore functionality.

At the end of each global data bus is a data controller, which handles processing of the inputs and outputs from the application. Because the sequence of data writes and reads from the fabric depends upon the number of physical stripes in the FPGA and the number or virtual stripes in the application, the data controller must do run-time scheduling of memory accesses. In order to provide the necessary memory bandwidth, the data controllers may contain memory caches to take advantage of data locality, or FIFOs to deal with the "bursty" memory traffic that is caused by virtualizing the application. All the data controllers access off-chip memory through a shared memory bus control unit. This unit arbitrates access to a single memory bus. The memory bus control unit is also the path used to load the configuration memory.

Two of the data controllers have additional functionality that allow them to deal with the problem of saving and restoring a stripe state when it is removed and later returned to the FPGA fabric. The physical stripes in PipeRench are constructed to have a special path from a global bus into and out of the registers on that stripe. This path is enabled when the stripe contains state that would be lost if that stripe was removed from the fabric. The state information for each stripe is stored in an on-chip state memory. This memory has one location for each location in the configuration memory, and can therefore hold the state for any application that can fit into the configuration memory. In order to keep track of which virtual stripe is placed in each physical stripe, there is an Address Translation Table (ATT in Fig. 9) with one entry per physical stripe.

4. Configuration Management

In this section we describe how the virtual stripes of an application are mapped to the physical stripes of the hardware fabric. Since pipelined reconfigurable architectures can map an application of any size to a given physical fabric, the configuration controller must handle the time-multiplexing of the application's stripes onto the physical fabric, the scheduling of the stripes, and the management of the on-chip configuration memory. Additionally, the controller is the interface between the host, the configuration memory, the fabric, and the data controllers.

We assume the interface between the FPGA and the CPU host resembles a typical slave co-processor, like a floating-point unit. After a general description applicable to all pipelined reconfigurable architectures, we present the controller used by PipeRench. The interaction between the configuration controller and the data controller is discussed in Section 5.

4.1. Characteristics of a Configuration Controller

We break down the tasks of managing the configurations into four sub-tasks: interfacing (between the host

المنارة للاستشارات
www.manaraa.com

and the fabric), mapping (the configuration words to the fabric), scheduling (time-multiplexing and managing virtualization), and managing the on-chip configuration memory.

The controller manages the interface between the CPU host and the fabric. At the very least the interface must allow the host to initiate execution of a particular configuration, and allow the FPGA co-processor to indicate that it has completed execution. If the configuration information is stored in main memory, it is possible to specify the application by giving the main-memory address of the first configuration word of an application, the number of iterations to be performed, and the main-memory addresses for data input and output. The co-processor could signal the completion of the application through an interrupt or a status register that is polled by the CPU.

The mapping task involves loading the virtual stripes into the on-chip configuration memory and the fabric itself. If the application fits in the fabric, the task is greatly simplified. If, however, the application is larger than the available hardware, stripes need to be swapped out during execution. Therefore, given an application, the controller must detect the case when virtualization is required and time-multiplex the application appropriately.

The controller schedules individual stripes of an application to ensure that each virtual stripe is present in the fabric long enough to process all the data: if a virtual stripe needs to be swapped out prematurely, it is reloaded later. Figure 11 shows the extent of time that the first and last virtual stripe spend in the fabric for the virtualized and the non-virtualized case. In the virtualized case, i.e., $V > P$ where V is the number of virtual stripes and P is the number of physical stripes, the number of active cycles for each stripe has a plateau of length $(V - P + 1)$ which occurs when the stripe is swapped out of the physical fabric. Each time a virtual stripe is loaded into the fabric it remains there for at most $P - 1$ active cycles. The controller thus has to swap stripes in and out at regular intervals. Points F0 and F1, and L0 and L1 in Fig. 11 indicate the initial loading and completion points of the two stripes; the stripes are swapped out at the points F2 and L2, and swapped back in at F3 and L3 respectively.

Finally, the controller must use the on-chip configuration memory efficiently, since going off-chip to fetch a configuration word is time-consuming, and may lead to pipeline stalls. If an application or multiple applications have common configuration words, these may be shared; shared configuration words need appear only once in the on-chip memory. Thus space utilization is enhanced as are the chances of fitting an application in the on-chip memory.

4.2. PipeRench's Configuration Controller

Here we present our implementation of a configuration controller for PipeRench. For the sake of simplicity, we omit discussion of pipeline stalls and present a controller that loads the entire application into the on-chip memory before beginning execution.

The CPU initiates the execution of an application on PipeRench by loading a set of control registers with the starting address of the executable in main memory, and the number of iterations to perform using this executable.

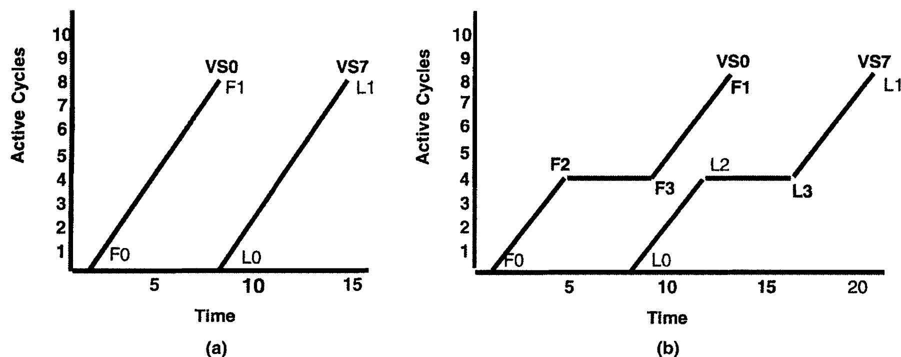


Figure 11. Active cycles example: Variation of the active cycles with time for (a) the non-virtualized and (b) the virtualized case. (a) Shows the case for 8 virtual stripes on 8 physical stripes while (b) shows the case for 8 virtual stripes on 5 physical stripes. The two curves represent the first and the last virtual stripes (VS0 and VS7).

المنارة للاستشارات

www.manaraa.com

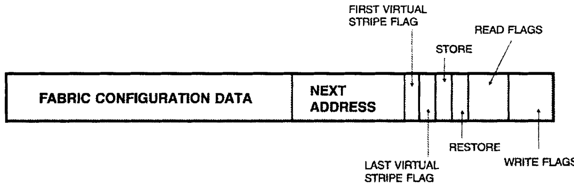


Figure 12. Configuration word. The structure of a configuration word consisting of the configuration data that goes to the fabric, the next address field, and a set of flags. The flags comprise indicators for the first and the last virtual stripes, and other fields described in Section 5.

In PipeRench, an "executable" is composed of a series of configuration words each of which includes three fields: fabric configuration bits, a next-address field, and a set of flags used by the configuration and data controllers (see Fig. 12). The flags relevant to the configuration controller are the first- and the last-virtual-stripe flags. The controller uses these to determine the iteration count and the number of stripes in the application.

The general architecture of the controller is shown in Fig. 13. When the *IDLE* line is asserted, the host can start a new application by specifying a start address and the number of iterations. The controller then deasserts the *IDLE* line until the application has completed the number of iterations specified.

4.2.1. Mapping the Configuration. Each virtual stripe in an application includes a next-address field which is used by the controller to find and then load the next stripe in the application. When the stripe is placed in the on-chip configuration memory, the next-address field is translated to an address in the on-chip memory. A record of this translation is maintained in a fully-associative on-chip Stripe Address Translation Table (SATT), shown in Fig. 13. Fortunately, the number of entries in the SATT is small compared to the size of the application, therefore it will not be on the critical path.

A counter is used to maintain the number of virtual stripes in the application. If the number of virtual stripes is larger than the number of physical stripes in the fabric, the controller will time-multiplex the application onto the fabric.

4.2.2. Configuring Physical Stripes. On every cycle the controller enables a specific physical stripe to be reconfigured. PipeRench uses a counter modulo the number of physical stripes to sequentially generate physical stripe addresses. This simple method automatically ensures that if the application is too big to fit in the fabric, configured stripes are overwritten and the hardware is virtualized over the entire physical fabric.

4.2.3. Tracking the Iterations. Once stripes are overwritten, they may need to be reloaded since all the requested iterations may not have been performed (i.e., each stripe may not have processed all the data required). In order to do this and execute an application

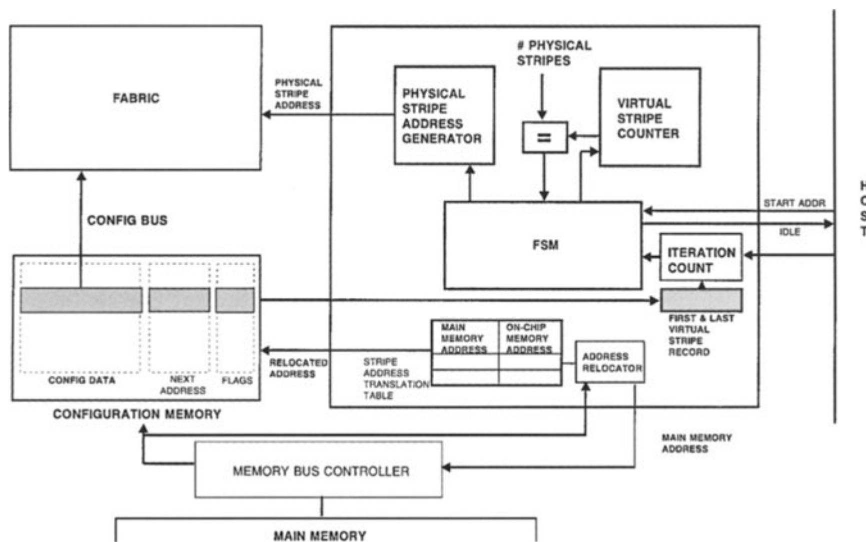


Figure 13. Configuration controller architecture: The configuration controller, and its interface to the host, main memory, on-chip memory, and the fabric.

www.manaraa.com

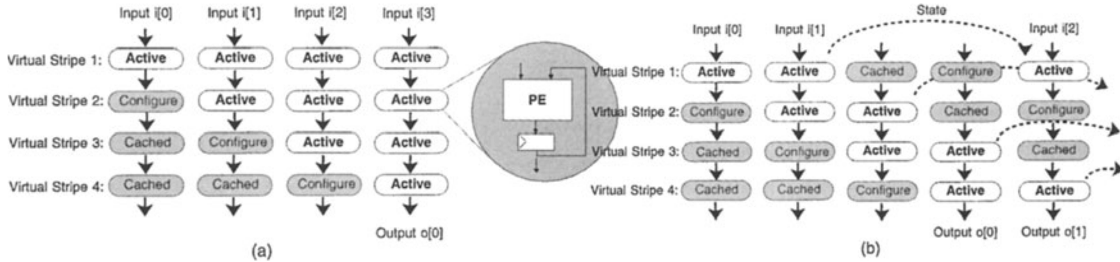


Figure 14. Virtualization I/O: Comparing input/output and state management with no virtualization and virtualization. (a) With enough hardware (no virtualization) there is no need to save state and input/output timing remain unchanged, (b) with less than enough hardware (virtualization) a stripe's state must be saved and input/output timing changed.

for a certain number of iterations, we use two of the flag bits: the first-virtual-stripe flag and the last-virtual-stripe flag.

When the first virtual stripe is loaded into the fabric, the controller records the cycle it was loaded. By monitoring this record during loading and swapping stripes, it can ascertain the number of cycles the first virtual stripe has spent in the fabric (i.e., the number of iterations it has executed). In addition to monitoring the first stripe, the controller also monitors when the last virtual stripe is swapped into the fabric.

Using the first and the last stripe, the iteration count may be managed in the following manner: when the first virtual stripe completes its required number of iterations, it does not need to be reloaded ever again. Hence the loading of the application can now stop (and a new application may be started) after loading the last virtual stripe.

4.3. Summary

In this section, we analyzed and described the four main sub-tasks of configuration management for pipelined reconfigurable architectures: interfacing, mapping, scheduling and memory utilization. In our implementation of the configuration controller for PipeRench, we use a next-address field to access configuration words from memory, use a counter (modulo the number of physical stripes) to generate the physical stripe addresses, and identify the first and last stripes by flags in order to keep track of iterations. This simple configuration controller can map an application with any number of virtual stripes onto a fabric with a given physical size.

5. Data Management

Managing the flow of data for virtualized pipelines is one of the main challenges in designing a pipelined reconfigurable architecture. Virtualization can cause disruptions in the flow of data, requiring the explicit management of execution state. The design goal in PipeRench's to make these disruptions transparent to the designer. This section presents our data controller architecture and shows how it manages the virtualization of a convolution kernel.

When there is no virtualization, there is no need to store and restore state or change input/output timing. Figure 14(a) shows the execution of a simple pipeline with no virtualization. Though PEs may contain functions of their own registered outputs, there is no need to save state because all the configurations remain in the fabric. Also, inputs and outputs are needed every cycle since the stripes that need input and output remain in the fabric.

However, when such pipelines are virtualized, the stripe state may need to be remembered and the input/output timing changed. Figure 14(b) shows the execution of the same pipeline, which now requires virtualization since there are only three physical stripes for the four virtual stripes. When stripes are functions of their own registered outputs, the state of that stripe must be stored while its configuration is not in a physical stripe and restored when it is returned to the fabric. Furthermore, input and output are only needed when the stripes that consume or produce data are in the fabric. In the example in Fig. 14, input (output) is only needed when the first (last) stripe is in the fabric.

5.1. Data Controller Architecture

The data controller architecture consists of four separate data controllers (see Fig. 9). Each controller manages one global bus that is dedicated to either state storing, state restoring, data input or data output per application. When dedicated to storing or restoring state, the data controller interfaces between the fabric and the state memory. When a controller is dedicated to

www.manaraa.com

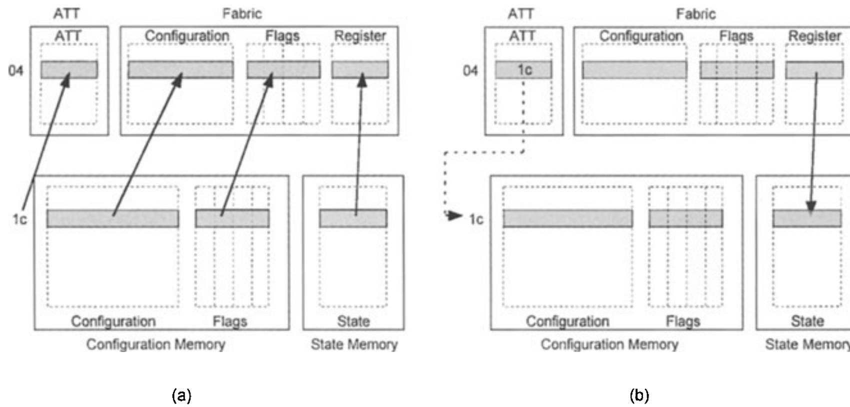


Figure 15. Store/Restore. Restoring (a) and storing (b) state between fabric, configuration memory, state memory and ATT.

data IO, the controller interfaces between the fabric and the memory bus controller. To determine which task each data controller performs, controllers contain control registers which describe functionality. The control registers specify the beginning data address, stride, and whether that bus is used for input, output, store, or restore.

5.1.1. Managing Stripe State. When needed, a stripe's state is kept in the state memory (see Fig. 16), which is addressed differently for stores and restores. During a restore, which takes place in the configuration cycle, the state memory address is the same address as that used to access the configuration memory. As Fig. 15(a) shows, when a stripe's configuration is written into the fabric, that stripe's state and flags are also written. In order to remember the address in the state memory for that stripe's state, the configuration memory address is written into the Address Translation Table (ATT). When storing state, the ATT supplies the state memory address, as shown in Fig. 15(b).

5.1.2. Managing Data IO. When managing Input/Output, configured stripes communicate with the input and output controllers through flags, and these controllers communicate via address and control logic with the memory bus controller. Each controller receives the flag bits that show the read and write data requests for its corresponding bus (Read Flags and Write Flags in Fig. 12). The flag bits are part of each stripe's control word and specify if that stripe reads or writes to each of the four buses. The data controllers receive these flag bits from the fabric and generate the necessary address and control lines for the memory bus controller (see Fig. 16). Therefore, when a stripe is configured to produce data on a bus, the controller

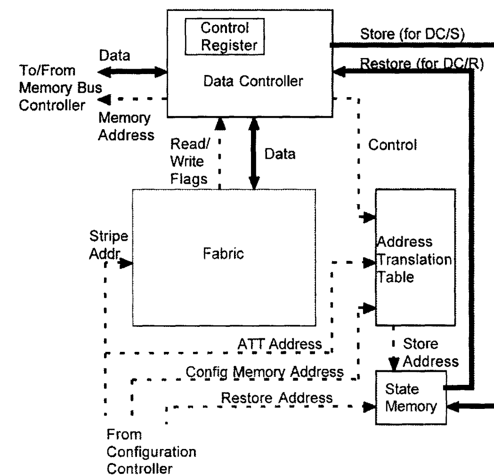


Figure 16. Data controller architecture. Solid lines are data and dashed lines are address or control.

generates the appropriate signals to write the data (likewise for a read).

The data controller is also responsible for generating the addresses for both the input and output data streams. We currently can generate addresses that are affine functions of the loop index. The starting address is supplied by the host when the application starts and the stride is specified as part of the application. When the fabric performs a read or write, the next address is in the sequence is generated by incrementing the current address by the stride. We are examining ways of generating addresses for a richer set of applications.

5.2. *Example: Convolution Data Flow*

To make the function of the data controllers more concrete, we now illustrate how the application presented

www.manaraa.com

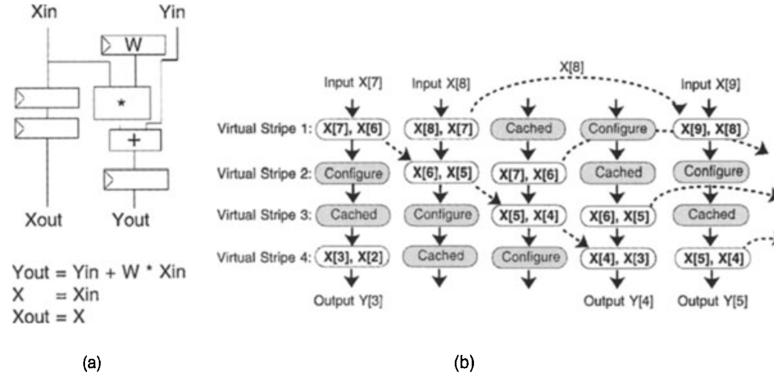


Figure 17. Systolic convolution. (a) Each stage's function contains a double pipelined X input, single pipelined Y output, and stationary weight W, (b) example of the data flow for this implementation. The dashed lines indicate how Y is accumulated as time progresses. The dashed arcs indicate state store and restore.

in Fig. 1 is virtualized on PipeRench. In the terminology presented in Kung [1], this is a strictly systolic implementation with the *X* input stream doubly pipelined. In [14], we also present a semi-systolic implementation of this same application, and illustrate the significant difficulties caused by an operand broad-cast over multiple stripes. For this reason, PipeRench current only supports strictly systolic implementations.

Figure 17 shows a fully systolic implementation of the application, which contains a single pipelined output Y, a double pipelined input X, and stationary weight W. In this example, we will assume that the functionality for one tap of this convolution can be supplied by one stripe. The X's enter the pipeline from the first stage. Every cycle a new X with a higher index is inserted. The data controller for this bus addresses the data memory from the beginning address supplied in its control registers. The data is driven on the bus and is read by the first stripe. When the first stripe asserts the corresponding read flag, the data controller increments the memory address by the contents in the stride register (in this case, 1) and readies the next piece of data on the bus. A controller for the pipelined Y output is similar, with the exception that it monitors the write flags and writes the data into memory instead.

In this example, some of the data in a stripe needs its state stored or restored. The double pipelined X contains state that needs to be stored and restored; the registered feedback is from the first register delay to the second register delay in the same stripe. The single pipelined Y value does not require storing or restoring since the stripe's functions do not contain registered feedback.

5.3. *Summary*

Data management should be transparent to applications no matter how many physical stripes are present and virtual stripes are needed. Our data controller architecture handles this transparency with communication between the stripes in the fabric and the data controllers. Through several flags in the control word of each stripe, the data controllers can tell what is needed by the fabric and the status of execution.

6. Performance

In this section we compare the expected performance of our architecture against commercial FPGAs with similar processing technology and area, and against commercial DSP processors on FIR filters of varying sizes.

Based on our design of the PipeRench prototype in 0.5 micron silicon, we believe that in 50 mm^2 of 0.35 micron silicon it is possible to have 28 stripes, each with a 128-bit wide datapath. Expected cycle time for this datapath is 100 MHz. An SRAM for configuration memory will consume another 50 mm^2 of area, and will store 256 configuration words of 768 bits each. One 128-bit wide stripe is capable of holding one tap of a 8-bit FIR filter with 12-bit coefficients. The total area for this chip would be 100 mm^2.

As shown in Fig. 18 the virtualization enables an FIR filter with less than 29 taps to run at the full clock rate of 100 MHz. Larger filters demonstrate a graceful degradation of performance out to around 256 taps, at which point the on-chip configuration storage is full.

للاستشارات المنارة
www.manaraa.com

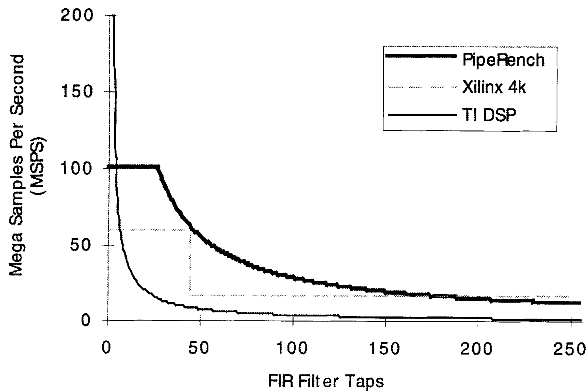


Figure 18. Performance on 8-bit FIR filters. PipeRench, Xilinx FPGA using parallel and serial arithmetic and Texas instruments DSP.

For larger filters, smart cache management techniques can be used to continue the degradation, albeit at a steeper rate due to the need to fetch some configuration data from off-chip.

Based on measurements of Xilinx FPGAs built in 0.35 micron technology [15], 100 mm^2 of area is equivalent to about 1750 CLBs. Given this amount of logic, and using parallel distributed arithmetic, it is possible to create filters that run at around 60 MHz and have up to 48 taps [16]. More than 48 taps will not fit. No widely known techniques can increase the throughput of this implementation. To implement larger filters, it is necessary to transform the algorithm to use a more efficient arithmetic. Using double-rate distributed arithmetic, it is possible to construct filters with up to 260 taps given the same amount of silicon [16]. Due to the serial nature of these implementations however, the maximum sampling rate of these filters is 14 MHz. There is a larger space of filters that are unfulfilled by this solution. The discontinuities in this graph make it difficult to compute the cost/performance of the device. In addition, the discontinuities represent a significant redesign effort. The two types of arithmetic used in this case require complete new run through the synthesis and physical design tools.

The Texas Instruments TMS320C6201 [17] is a commercial DSP which runs at 200 MHz and contains two 16- by 16-bit integer multipliers. For filters with less than four taps, the high clock speed of this device yields the highest possible performance. This performance decays rapidly with an increasing number of taps due to the presence of only two multipliers. PipeRench exhibits a very similar curve to this DSP, only the capacity of the device is significantly higher. PipeRench can hold about 29 taps until the hardware is time-multiplexed. Due to the word-oriented functional units in PipeRench, the maximum clock rate is significantly higher than the FPGA. And like the DSP, the degradation in performance has no large discontinuities, and requires no re-design to adapt to less hardware.

7. Related Work

PipeRench provides robust compilation by allowing an application to transparently exceed the logical capacity of the physical FPGA at runtime. The Virtual Wire "softwire" compiler [18] provided a degree of robustness by virtualizing the I/Os between FPGAs in a multi-FPGA logic emulation system at compile time. The challenge faced by most FPGA-based logic emulators is that the input netlist is usually too large to fit into one FPGA. The netlist must be partitioned across multiple devices and meet FPGA I/O constraints. When I/O constraints are violated, the "softwire" compiler time-multiplexes different logical I/Os on a single physical I/O. The I/O constraint violation is fixed by reducing performance. PipeRench is a single-chip FPGA computing devices, not a logic emulator. Our objective is to deal with large logical netlists, not by overflowing into other devices and dealing with I/O constraints, but by time-multiplexing the on-chip logic to emulate the desired design at a degraded level of performance.

Multiple context FPGAs [9–11, 19], have been proposed as a way to create logically larger devices through rapid reconfiguration. These architectures do allow idle logic to be stored outside of the active FPGA fabric, and allow the active fabric to change very rapidly. They can be used to virtualize hardware, but because they cannot be incrementally reconfigured they suffer from the pipeline fill and empty penalty. Furthermore, the task of compilation for these architectures is more complex than it is for a flat, single context FPGA, because the compiler needs to place and route multiple, interdependent contexts simultaneously. PipeRench simplifies the compilation process by allowing the compiler to create a pipeline of unbounded length. The only real design constraint is making the individual pipeline stages fit into PipeRench stripes.

A form of pipeline reconfiguration for commercial FPGAs, such as the XC6200, has been described [13]. The XC6200 cannot be reconfigured at the same rate as the data flow, and it is therefore necessary to segment the pipeline. PipeRench has the configuration

المنارة للاستشارات
www.manaraa.com

bandwidth necessary to support pipeline reconfiguration, and includes mechanism for control of the configuration stream and data stream with respect to the virtualization.

Other devices are capable of partial, run-time reconfiguration, such as GARP [20], NAPA [21], and Chimeara [22], and could potentially operate using pipeline reconfiguration. Exploration of the interface between FPGAs and CPUs has been investigated in [20–24].

PipeRench also addresses many of the problems faced by other computer architectures. The most-insightful comparisons are to MMX, VLIW, and vector machines.

The mismatch between application data size and native operating data size has been addressed by extending the ISAs of microprocessors to allow a wide data path to be split into multiple parallel data paths, as in Intel's MMX [25]. Obtaining SIMD parallelism to utilize the parallel data paths is non-trivial, and works only for very regular computations where the cost of data alignment does not overwhelm the gain in parallelism. PipeRench has a rich interconnect to provide for alignment and allows PEs to have different configurations so that parallelism need not be strictly SIMD.

VLIW architectures are designed to exploit dataflow parallelism that can be determined at compile time [26]. VLIWs have extremely high instruction bandwidth demands. A single PipeRench stripe is similar to a VLIW processor using many small, simple functional units. In PipeRench, however, a stripe remains configured for a number of cycles, and the same computation is performed on a larger data set, thereby amortizing the instructions over more data.

The instruction bandwidth issue has been addressed by vector machines such as the T0 [27] and IRAM [28]. In many ways, PipeRench is similar to vector machines with an unbounded vector size and with VLIW functional units. The problem with classical vector architectures is the vector register file is a physical or logical bottleneck that limits scalability. Allocating additional functional units in a vector processor requires additional ports on the vector register file. The physical bottleneck of the register file can be ameliorated by providing direct forwarding paths to allow chained operations to bypass the register file, as in the Cray-1 [29]. PipeRench eliminates these constraints by eliminating the vector register file. All connections in PipeRench are local, and the chaining is explicit. Therefore, the number of functional units can grow without increasing the complexity of the issue and control hardware.

8. Conclusions

Pipeline-reconfigurable FPGAs provide the high-performance associated with FPGAs for DSP applications. In addition, they provide the forward-compatibility and robust compilation that is associated with more traditional processors. We believe these benefits enable the development of FPGAs that have the performance advantages for DSP applications associated with current FPGAs, and the ease and economy of development associated with microprocessors.

Managing the configuration and data flows is a significant issue in the design of these devices. PipeRench's configuration controller performs run-time mapping and scheduling of configuration transfers, interfaces to the host processor, and manages the configuration storage. The data controllers provide mechanisms for storing and restoring of state, as well as access to operand data for a variety of systolic and semi-systolic pipeline implementations.

Acknowledgments

We would like to acknowledge the efforts of former members of the PipeRench research group: Jeffry Weener, Kevin Jaget, Matthew Myers and Ronald Laufer. We thank the anonymous reviewers for their helpful feedback.

References

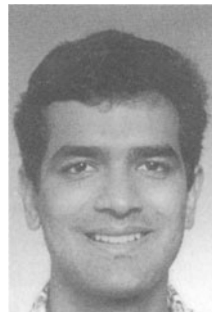
1. H.T. Kung, "Why Systolic Architectures?" *IEEE Computer*, Piscataway, NJ, 1982, pp. 37–45.
2. R. Miller and Q.F. Stout, *Parallel Algorithms for Regular Structures*, MIT Press, 1996.
3. D.I. Moldovan, "On the Analysis and Synthesis of VLSI Algorithms," *IEEE Transactions on Computers*, vol. C-31, no. 11, 1982, pp. 1121–1126.
4. D.I. Moldovan, "On the Design of Algorithms for VLSI Systolics Arrays," *Proceedings of the IEEE*, vol. 71, no. 1, 1983, pp. 113–120.
5. B. Von Herzen, "Signal Processing at 250 mhz Using High-Performance FPGAs," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 1997, pp. 62–68.
6. Xilinx, *XC4000 Series Field Programmable Gate Arrays*, revision 1.04, Sept. 1996.
7. Altera, Data book, 1998.
8. Xilinx, *XC6200 Field Programmable Gate Arrays*, revision 1.7, Oct. 1996.
9. A. DeHon, "DPGA-Coupled Microprocessors: Commodity ICs for the Early 21st Century," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (Eds.), Napa, CA, April 1994, pp. 31–39.

للاستشارات المنارة

www.manaraa.c

10. S. Scalera and J.R. Vazquez, "Design and Implementation of a Context Switching FPGA," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K.L. Pocek (Eds.), Napa, CA, April 1998, pp. 78–85.
11. S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A Time-Multiplexed FPGA," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, J. Arnold and K.L. Pocek (Eds.), Napa, CA, April 1997, pp. 22–28.
12. M.J. Wirthlin and B.L. Hutchings, "Sequencing Run-Time Reconfigured Hardware with Software," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, CA, February 1996, pp. 122–128.
13. W. Luk, N. Shirazi, S.R. Guo, and P.Y.K. Cheung, "Pipeline Morphing and Virtual Pipelines," *Field-Programmable Logic and Applications*, P.Y.K. Cheung, W. Luk, and M. Glesner (Eds.), London, England, Sept. 1997, pp. 111–120.
14. S. Cadambi, J. Weener, S.C. Goldstein, H. Schmit, and D.E. Thomas, "Managing Pipeline-Reconfigurable FPGAs," *Proceedings ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, February 1998, pp. 55–64.
15. J. Rose, Private communications, 1997.
16. S. Knapp, Using Programmable Logic to Accelerate DSP Functions, 1995. http://www.xilinx.com/appnotes/dspintro.pdf.
17. Texas Instruments, TMS320C6201 digital signal processor, revision 2, 1997.
18. J. Babb, R. Tessier, and A. Agarwal, "Virtual Wires: Overcoming Pin Limitations in FPGA-Based Logic Emulators," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, D.A. Buell and K.L. Pocek (Eds.), Napa, CA, April 1993, pp. 142–151.
19. N. Bhat, K. Chaudhary, and E.S. Kuh, Performance-oriented fully routable dynamic architecture for a field programmable logic device. M93/42, U.C. Berkeley, 1993.
20. J.R. Hauser and J. Wawrzynek, "Garp: A MIPS Processor With a Reconfigurable Coprocessor," *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997, pp. 24–33.
21. C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale, "The NAPA Adaptive Processing Architecture," *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1998, pp. 28–37.
22. S. Hauck, T.W. Fry, M.M. Hosler, and J.P. Kao, "The Chimaera Reconfigurable Functional Unit," *IEEE Symposium on FPGAs for Custom Computing Machines*, April 1997, pp. 87–96.
23. R. Razdan and M.D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," *MICRO-27*, November 1994, pp. 172–180.
24. R. Witting and P. Chow, "OneChip: An FPGA Processor with Reconfigurable Logic," *IEEE Symposium on FPGAs for Custom Computing Machines*, 1996.
25. A. Peleg, S. Wilkie, and U. Weiser, "Intel MMX for Multimedia PCs," *Communications of the ACM*, vol. 40, no. 1, 1997, pp. 24–38.
26. R.P. Colwell, R.P. Nix, J.J. O'Donenell, D.B. Papworth, and P.K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler," *Proceedings of ASPLOS-II*, March 1987, pp. 180–192.
27. J. Wawrzynek, K. Asanovic, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan, "Spert-II: A Vector Microprocessor System," *IEEE Computer*, vol. 29, no. 3, March 1996, pp. 79–86.
28. C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable Processors in the Billion-Transistor Era: IRAM," *IEEE Computer*, 1997, pp. 75–78.
29. R.M. Russell, "The CRAY-1 Processor System," *Communications of the ACM*, vol. 21, no. 1, 1978, pp. 63–72.

Herman H. Schmit is an assistant professor of Electrical and Computer Engineering at Carnegie Mellon University. He received his Masters and Ph.D. from Carnegie Mellon in 1992 and 1995, respectively. His research interests include reconfigurable computing, low power digital design and IP-based design.
herman@ece.cmu.edu



Srihari Cadambi is currently a Ph.D. candidate in the department of Electrical and Computer Engineering at Carnegie Mellon University. He received his Masters from the University of Massachusetts, Amherst, in 1996. His research focus is compiler optimizations for reconfigurable architectures.
cadambi@ece.cmu.edu

Matthew Moe is currently a Ph.D. candidate in the department of Electrical and Computer Engineering at Carnegie Mellon University. He received his B.S. and M.S. degrees from Carnegie Mellon in 1996

المنارة للاستشارات
www.manaraa.com

and 1998, respectively. His research focus is on design optimization and scalability of reconfigurable computing architectures.
moe@ece.cmu.edu

Seth Copen Goldstein is an assistant Professor in the School of Computer Science at Carnegie Mellon University. He received his Ph.D. from the University of California at Berkeley in 1997. Before attending UC-Berkeley he was the founder and CEO of Complete Computer Corporation which developed Complete C, a cross platform object-oriented toolkit. His current research interests focus on architectures and compilers for reconfigurable computing systems.
seth@cs.cmu.edu

المنارة للاستشارات

www.manaraa.c

Journal of VLSI Signal Processing Systems 24, 147–164, 2000.
© 2000 Kluwer Academic Publishers. Printed in The Netherlands.

Design and Implementation of the MorphoSys Reconfigurable Computing Processor

MING-HAU LEE, HARTEJ SINGH, GUANGMING LU, NADER BAGHERZADEH AND FADI J. KURDAHI
University of California, Irvine, Electrical and Computer Engineering Department, Irvine, CA 92697, USA

ELISEU M.C. FILHO AND VLADIMIR CASTRO ALVES
COPPE/Federal University of Rio de Janeiro, Department of Systems and Computer Engineering, P.O. Box 68511 21945-970, Rio de Janeiro, RJ Brazil

Abstract. In this paper, we describe the implementation of MorphoSys, a reconfigurable processing system targeted at data-parallel and computation-intensive applications. The MorphoSys architecture consists of a reconfigurable component (an array of reconfigurable cells) combined with a RISC control processor and a high bandwidth memory interface. We briefly discuss the system-level model, array architecture, and control processor. Next, we present the detailed design implementation and the various aspects of physical layout of different sub-blocks of MorphoSys. The physical layout was constrained for 100 MHz operation, with low power consumption, and was implemented using 0.35 μm, four metal layer CMOS (3.3 Volts) technology. We provide simulation results for the MorphoSys architecture (based on VHDL model) for some typical data-parallel applications (video compression and automatic target recognition). The results indicate that the MorphoSys system can achieve significantly better performance for most of these applications in comparison with other systems and processors.

1. Introduction

Reconfigurable computing systems are systems that consist of some reconfigurable hardware along with software programmable processors. The reconfigurable component provides the ability to configure or customize the system for one or more applications [1]. In the ideal case, a reconfigurable system delivers high performance typical of ASIC devices and also provides the flexibility of a general-purpose processor (i.e. it can execute a wide range of applications). Conventionally, field programmable gate arrays (*FPGA*s) [2] are the most common devices used for implementing reconfigurable components. This is because *FPGAs* allow designers to manipulate gate-level devices such as flip-flops, memory and other logic gates. However, *FPGAs* have certain disadvantages such as low logic density and inefficient performance for word-level datapath operations. Hence, many researchers have proposed prototypes of reconfigurable computing systems that employ non-FPGA reconfigurable components such as *DPGA* [3], *Garp* [4], *PADDI* [5], *MATRIX* [6], *RaPiD* [7], *REMARC* [8], and *RAW* [9].

In this paper, we describe the implementation of MorphoSys, which is based on a novel model of a reconfigurable computing system. This model is aimed at applications that feature high data-parallelism, regularity, and are computation-intensive. Some examples of these applications are video compression, graphics and image processing, and DSP transforms. The implementation of MorphoSys operates at 100 MHz, and the entire design has a silicon area of about 200 sq.mm.

1.1. Organization of Paper

Section 2 introduces the system architecture and components of MorphoSys. A cross-section of related work is briefly described and contrasted with MorphoSys architecture in Section 3. The physical implementation aspects of the MorphoSys system, with its focus on clock cycle of 10 ns (for operating freq. of 100 MHz)

المنارة للاستشارات
www.manaraa.com

and low power consumption, are presented in Section 4. Section 5 gives an overview of the current programming and simulation environment for MorphoSys. Next, in Section 6, we provide performance estimates for a set of applications (video compression and ATR) that have been mapped to MorphoSys. Finally, we list some conclusions in Section 7.

2. MorphoSys Architecture

The MorphoSys design model incorporates a reconfigurable component (to handle high-volume data-parallel operations), on the same die with a general-purpose RISC processor (to perform sequential processing and control functions), and a high bandwidth memory interface.

2.1. MorphoSys Components

The MorphoSys architecture comprises five major components: the Reconfigurable Cell Array (RC Array), control processor (TinyRISC), Context Memory, Frame Buffer and a DMA Controller. Figure 1 shows the organization of the integrated MorphoSys reconfigurable computing system.

RC Array. In the current implementation, the reconfigurable component is an array of reconfigurable cells (RCs) or processing elements. Considering that target applications (video compression, etc.) tend to be processed in clusters of 8 × 8 data elements, the Reconfigurable Cell array (RC Array) has 64 cells in a two-dimensional matrix. This configuration is chosen to maximally utilize the parallelism inherent in an application, which in turn enhances throughput.

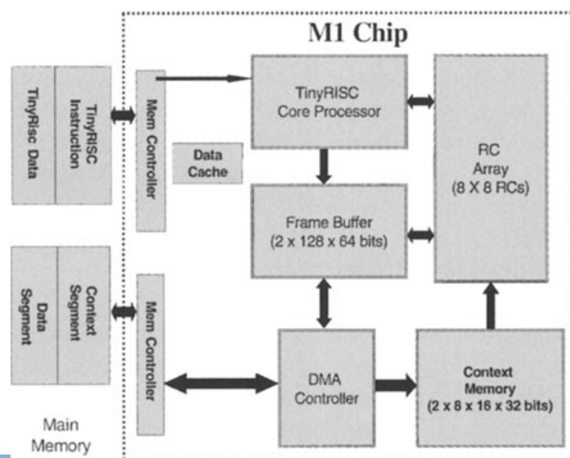


Figure 1. MorphoSys integrated architectural model.

The RC Array follows the *SIMD* model of computation. All RCs in the same row/column share same configuration data (context). However, each RC operates on different data. Sharing the context across a row/column is useful for data-parallel applications. The RC Array has an extensive three-layer inter-connection network, designed to enable fast data exchange between the RCs. This results in enhanced performance for application kernels that involve high data movement, for example, the discrete cosine transform (used in video compression).

Each RC incorporates an ALU-multiplier, a shift unit, input muxes and a register file. The multiplier is included since many target applications require integer multiplication. In addition, there is a context register that is used to store the current context and provide control/configuration signals to the RC components (namely the ALU-multiplier, shift unit and the input multiplexers).

TinyRISC Control Processor. Since most target applications involve some sequential processing, a RISC processor, TinyRISC [10], is included in the system. This is a MIPS-like processor with a 4-stage scalar pipeline. It has a 32-bit ALU, register file and an on-chip data cache memory. This processor also coordinates system operation and controls its interface with the external world. This is made possible by addition of specific instructions (besides the standard RISC instructions) to the TinyRISC ISA. These instructions initiate data transfers between main memory and MorphoSys components, and control execution of the RC Array.

Frame Buffer and DMA Controller. The high parallelism of the RC Array would be ineffective if the memory interface is unable to transfer data at an adequate rate. Therefore, a high-speed memory interface consisting of a streaming buffer (Frame Buffer) and a DMA controller is incorporated in the system. The Frame Buffer has two sets, which work in complementary fashion to enable overlap of data transfers with RC Array execution.

Context Memory. The Context Memory stores multiple (32) planes of configuration data (context) for RC

المنارة للاستشارات
www.manaraa.com

Array, thus providing depth of programmability. This implies that the system spends less time loading fresh configuration data. Fast dynamic reconfiguration is essential for achieving high performance with a reconfigurable system. MorphoSys supports single-cycle dynamic reconfiguration (without interruption of RC Array execution).

2.2. System Control Mechanism

MorphoSys implements a novel control mechanism for the reconfigurable component through the TinyRISC instructions. The TinyRISC ISA has been modified to include several new instructions (Table 1) that enable control of different components in the system. These instructions contain fields that directly provide the values for different control signals to the RC Array, DMA controller, Frame Buffer and the Context Memory. There are two major categories of these new instructions: DMA instructions and RC Array instructions. The DMA instructions contain fields that provide the DMA Controller with adequate information (starting address in main memory, starting address in Frame Buffer of Context Memory, number of bytes to load, load or store control). This enables transfer of data between main memory and the Frame Buffer or the Context Memory through the DMA Controller.

The RC Array instructions have fields that provide the control signals to the RC Array and the Context Memory. This is essential to enable the execution of computations in the RC Array. This information includes the contexts to be executed, the mode of context broadcast (row or column), location of data to be loaded in from Frame Buffer, etc.

Table 1. New TinyRISC instructions.

LDCTXT	Load Context from Main Memory to Context Memory
LDFB (STFB)	Load (store) data from (into) Main Memory to (from) Frame Buffer
DBCBC, DBCBR	Column (or row) context broadcast, get data from both banks of Frame Buffer
DBCB	Context broadcast, get data from both banks of Frame Buffer
SBCB	Context broadcast, transfer 128 bit data from Frame Buffer
CBCAST	Context broadcast, no data from Frame Buffer
WFB	Write the processed data back to Frame Buffer (in address from register file)
RCRISC	Write one 16-bit data from RC Array to TinyRISC

2.3. MorphoSys Execution Model

The execution model for MorphoSys is based on partitioning applications into sequential and data-parallel tasks. The former are handled by the TinyRISC, whereas the latter are mapped to the RC Array. TinyRISC initiates all data transfers involving application and configuration data (context). Tiny RISC provides various control/address signals for Context Memory, Frame Buffer and the DMA controller [11]. RC Array execution is enabled through special TinyRISC instructions for context broadcast.

The MorphoSys program flow may be summarized as: first, a special TinyRISC instruction, *LDCTXT* is issued. This initiates loading of context words (configuration data) into the Context Memory through DMA Controller (Fig. 1). Next, the *LDFB* instruction causes the TinyRISC to signal the DMA Controller to load application data, such as image frames, from main memory to the Frame Buffer. When both configuration and application data are ready, a TinyRISC instruction for context broadcast, such as *CBCAST*, *SBCB*, etc. is issued. This starts execution of the RC Array.

The context broadcast instructions specify the particular context (from among the multiple contexts in Context Memory) to be executed by the RCs. There are two modes of specifying the context: *column broadcast* and *row broadcast*. For column (row) broadcast, all RCs in the same column (row) are configured by the same context word. TinyRISC can also selectively enable a row/column, and can access data from selected RC outputs.

MorphoSys supports dynamic reconfiguration. Context data may be loaded into a non-active part of the Context Memory without interrupting RC Array operation. Since the Frame Buffer has two sets, it is possible to overlap computation in RC Array with data transfers between external memory and the Frame Buffer. While the RC Array performs computations on data in one Frame Buffer set, fresh data may be loaded in the other set or the Context Memory may receive new contexts.

3. Related Work

There are two major classes of reconfigurable systems: fine-grain (processing units have datapath widths of a

www.manaraa.com

few bits) and coarse-grain (basic processing elements have data-paths of eight or sixteen bits or more). Research prototypes with fine-grain granularity include *Splash* [12], *DPGA* [3] and *Garp* [4]. Reconfigurable processors with coarse-grain granularity are *PADDI* [5], *MATRIX* [6], *RaPiD* [7], and *REMARC* [8]. MorphoSys is a coarse-grain architecture, since the target applications mostly involve pixel-processing. In this section, we compare and contrast some of the previously developed coarse-grain systems with MorphoSys.

Among the systems of *MATRIX*, *PADDI*, *RaPiD*, *REMARC* and *RAW* [9], MorphoSys has some common features with each, as well as some differences. A major difference is that most of these designs have not been implemented at the physical hardware level, whereas MorphoSys has been developed from the VHDL level down to the physical layout level and will be actually fabricated.

PADDI [5] has a different mechanism for storing and broadcasting the context word, it has less depth of programmability, more complex interconnection network using crossbar switches, and a distinct VLIW flavor since the instruction word is 53 bits. The EXUs receive the same global instruction but the decoded instruction is different for each EXU, which is different from MorphoSys. In MorphoSys, each row (column) of RCs receives the same context word, and it has same function for each.

MATRIX [6] has a similar interconnection network as MorphoSys, but unlike MorphoSys, the control and array processors are configured out of the same hardware resources. This makes the dynamic system control becomes quite complex. *MATRIX* lacks a multiplier in the basic processing element, the BFU. The levels of interconnect have variable delay (in terms of pipeline stages); this is constant for MorphoSys. This work does not specify the data interface to the external world.

RaPiD [7] is designed as a linear array of functional units, configured as a linear computation pipeline. Therefore, it performs well for systolic applications, but has limited performance for block-oriented application tasks, which MorphoSys performs very efficiently (even transpose operations are not needed). However, there is no unified macro-controller and an integrated memory interface is missing.

REMARC [8] has 64 nano-processors but these nano-processors do not have a multiplier (even though it targets multimedia applications), but instead have a 16 entry data RAM. The interconnection network has two levels, and the global control unit has to perform the functions of data transfers to the main processor/memory, whereas in MorphoSys, these transfers are carried out by the DMA controller, and are concurrent with the program execution. It does not allow dynamic reconfiguration.

RAW [9] is a system with a set of interconnected tiles of RISC processors. The configuration time for the interconnect switches is quite high (several instructions per switch). The design has many VLIW features, and each tile includes a FPGA-like configurable logic. Having a large number of RISC processors seems inefficient, when we consider that MorphoSys is able to execute several data-parallel applications at a high performance level using just one RISC processor.

In summary, the most prominent features incorporated in the MorphoSys architecture are:

- *Integrated model:* This has a novel control mechanism for the reconfigurable component that uses a general-purpose processor. Except for main memory, MorphoSys is a complete system-on-a-chip.
- *Multiple contexts on-chip:* this feature enables fast single-cycle reconfiguration.
- *On-chip controller:* allows efficient execution of applications that have both serial and parallel tasks.
- *Innovative memory interface:* high data throughput by using a two-set data buffer that allows overlap of computation with data transfer.

4. Implementation and Verification

In this section, we describe the steps involved in the design and implementation of the major components of MorphoSys: the Reconfigurable Cell (RC), the TinyRISC, the Context Memory, the Frame Buffer and the DMA Controller. The chip is designed using 0.35 μm 3.3 V four metal layers CMOS technology.

Design Methodology: MorphoSys components are implemented using the twin approaches of custom design and standard cell design. The components that constitute the critical path (e.g. RC) or the components that have a regular structure (e.g. Context Memory, and Frame Buffer) are custom designed. This enables extensive optimization of these components for the delay and area. The components that are control intensive, consist of random logic or are not in the critical path, are designed using logic synthesis tools (Synopsys and Mentor Graphics software). Four metal layers are available for routing; out of these, two (Metal 3 and

المنارة للاستشارات
www.manaraa.com

Metal 4) are reserved for routing between the component blocks. Only two layers are used for routing within a component (such as the Reconfigurable Cell). We use both IRSIM (switch-level simulator) and Hspice (transistor-level simulator) to verify the design of custom components. For synthesized components, Lsim simulator (switch mode and adept mode) is used for functional verification and timing analysis.

4.1. *Reconfigurable Cell*

The Reconfigurable Cell (RC) is the basic element of the RC Array, which is the reconfigurable component of MorphoSys. Each RC (Fig. 2) has a 16 × 12 multiplier. Most multi-media applications required that the second data input to the multiplier be less than or at most equal to 12 bits. Since a 16 × 12 multiplier is significantly smaller (and faster) than a 16 × 16 multiplier, and these savings would accrue over 64 RCs, it was decided to use a 16 × 12 multiplier. Corresponding to this input data size, the output of the multiplier cannot be greater than 28 bits. Based on this, we designed a 28 bit ALU for the RC.

The data to the multiplier/ALU is provided through two 16-bit input muxes. These muxes allow selection of data operands from different options. The RC decoder generates control signals for the muxes and the ALU. The critical path of RC consists of the 16 bit input mux, the 16 × 12 bit multiplier, the 28 bit ALU, and a shift unit. Table 2 shows all the functions implemented in RC. The special functions such as absolute value, count one's, and round are implemented as separate units from the ALU to simplify the logic complexity of

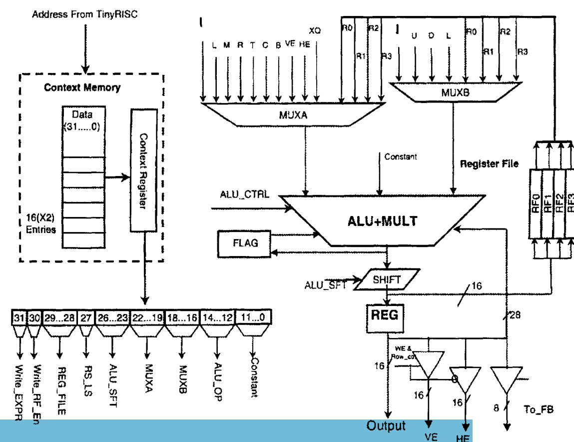


Figure 2. Reconfigurable cell architecture.

Table 2. RC functions.

Instruction	*Description*
A *OR* B, A *AND* B, A *XOR* B, A *OR* C, A *AND* C, A *XOR* C	Two-operand Logic functions
A + B, A − B, B − A, A + C, A − C	Two-operand arithmetic functions
A * C	Multiplication with constant
A*C + B, A*C + Out(t), A*C − Out(t)	Multiply-accumulate functions
| A − B | + Out(t)	Absolute difference accumulate
A *AND* B: Count One's	ANDing with count # of one's in result
A + B if A > 0, A − B if A < 0	Conditional add/subtract based on sign bit of A
Round {Out(t)} RESET, BYPASS A, LOAD Constant, No-op	Miscellaneous functions

A = Mux A operand, B = Mux B operand, C = constant, Out(t) = previous output, Out(t+1) = new output.

the ALU and improve the overall performance. In the following, the design of the three components which constitute the critical of the RC (multiplier, ALU, and shifter) will be discussed.

The constraint of completing the multiply-accumulate (MAC) and shift operations in one cycle (10 ns) is the most challenging part of the design of the Reconfigurable Cell. The tight delay constraint motivated the use of advanced circuit design techniques. Also, since there are 64 Reconfigurable Cells in the RC Array, a small increase in the area or power consumption of a RC would have resulted in a multiplicative effect. Hence, we manually designed the entire reconfigurable cell.

Multiplier. A 16 × 12 multiplier is implemented in RC. This is the component that requires the maximum area and has the longest delay in the RC. Therefore, we use complementary pass-transistor logic (CPL) circuit [13] for designing the multiplier. CPL allows the realization of complex logic functions with minimum number of transistors. It also features high speed operation and low power consumption.

Figure 3(a) (CPL1) shows the basic structure of the CPL circuit. The NMOS pass-transistor network is used to realize the logic function and the two output inverters are used as a level restoration block. This circuit suffers from static power consumption due to the low-swing feature of the NMOS pass-transistor networks. The high level of output inverter inputs are actually

المنارة للاستشارات
www.manaraa.com

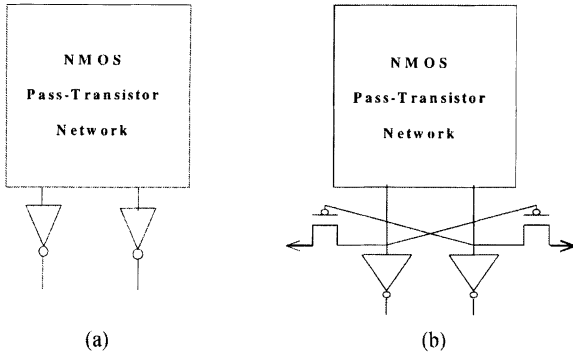


Figure 3. Complementary pass-transistor logic (CPL) structure.

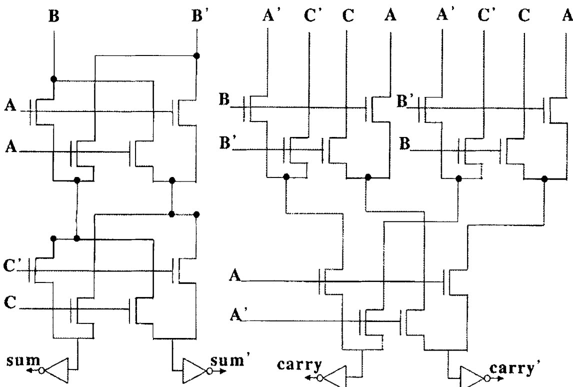


Figure 4. CPL implementation of carry-save adder.

lower than the supply voltage and PMOS transistors are not completely turned off in this situation and, therefore, leakage current will flow in the output inverters. Figure 3(b) (CPL2) shows the modification that solves the static power problem. The two small cross-coupled PMOS transistors are used to restore the outputs of the NMOS pass-transistor network to supply voltage level.

The multiplier is designed using carry-save adder (CSA) array structure with a 16 bits carry-skip adder. The CPL implementation of the CSA is shown in Fig. 4.

Table 3 shows the comparisons of three CSA designs: standard CMOS, CPL1, and CPL2. SPICE simulation (using HP level 39 0.35 μm device models) was carried out for each of the three CSA designs. From the data in Table 3 the CSA design using CPL2 has the lowest delay-power product, hence, it is used in current implementation.

Table 3. Comparison of standard CMOS and CPL carry-save adder design.

	Standard CMOS	CPL1	CPL2
Number of transistors	40	28	30
Delay (0.35 μm, 3.3 V)	0.54 ns	0.22 ns	0.20 ns
Power (100 MHz, 25°C)	0.21 mw	0.36 mw	0.18 mw

Several researchers [14] have shown that both Wallace [15] and Dadda [16] algorithms are efficient for array type multipliers and can be implemented using the minimum number of CSAs. However, we use a regular array structure instead, which requires more CSAs and has a longer critical path compared to Wallace or Dadda multiplier. The reason behind this decision is that our layout methodology allows only metal 1 and metal 2 for internal routing within components. If we design the multiplier using the Wallace or Dadda design, it would have a much larger area because of irregular structure of these designs. We estimate that both Wallace and Dadda multipliers are about 1.5 times larger than the regular array multiplier when only two layers are used for routing. Thus for 64 RCs, this increase in area is not tolerable. Hence, we use the regular array structure.

Another important design consideration is to find an efficient algorithm for 2's complement multiplication. The regular CSA array structure leads us to the decision of sign extension and array reduction algorithm [17] because the partial products can be generated without any recoding. Also, the summation of the partial products can be carried out by carry-save adders directly without any modification. Figure 5 shows the structure of the multiplier and the result of the 16 × 12 multiplication bit array after sign extension and array reduction.

It is important to note that the multiplier can be disabled when an application does not involve multiplication operations. This feature is realized by bypassing

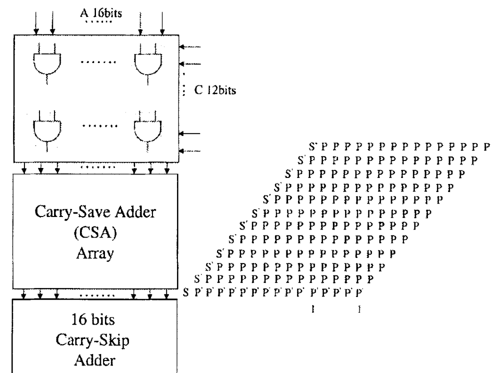


Figure 5. Structure and bit array of the 16 × 12 multiplier.

المنارة للاستشارات
www.manaraa.com

the inputs to the multiplier and having the RC decoder generate control signals to the bypass unit based on the context word. In MPEG, for example, only DCT/IDCT kernels require multiplication operations, which constitute less than 10% of the total operation count. By disabling the multiplier when not in use, a large amount of power can be saved for most of the applications.

SPICE simulations show that the multiplier delay is 4 ns (0.35 μm, 3.3V CMOS). The power dissipation is 150 mw at 25°C for 100 MHz operation with the input pattern of FFFF*FFF switching to 0000*000. The standby power consumption when inputs to multiplier are kept constant is only 0.6 mw.

ALU. The ALU of the RC is designed to implement basic and arithmetic functions. The logic core of the one bit ALU is shown in Fig. 6.

The important part of ALU implementation is to design the 28 bits adder/subtractor unit for minimum area and delay. The timing budget allows approximately 3 ns for ALU operations. The carry-ripple adder is too slow to accomplish 28 bits addition/subtraction operations in 3 ns. Both carry-lookahead adder and carry-select adder [18] are well-known schemes for high speed adder design, however, they require twice as much area as the carry-ripple adder. Consequently, we use carry-skip [18] scheme (that uses almost the same area as the carry-ripple adder but is much faster) for the ALU design. Figure 7 illustrates the structure of the 28 bits carry-skip adder.

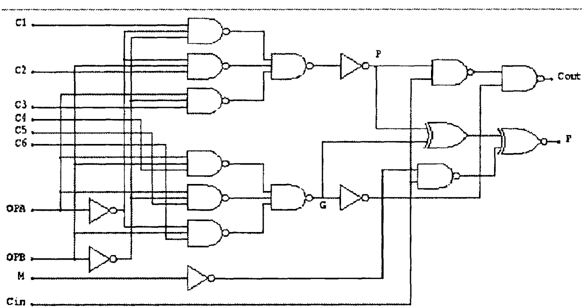


Figure 6. 1 Bit logic core of the ALU.

The 28 bits adder is divided into 7 groups with carry-ripple scheme used in each group. Every group also generates a carry-bypass signal that equals to 1 if all bits internal to the group satisfy $P_i = 1$ ($P_i = x_i \oplus y_i$, where x_i and y_i are the inputs to each bit). This signal can allow the incoming carry to bypass all bits within the group and propagate to the next group. Thus, it reduces the time needed to propagate the carry by skipping over groups of 4 consecutive adder bits.

The worst case operation time of the 28 bits ALU from SPICE simulation is within 3 ns. It consumes 15 mw of power at 25°C for 100 MHz operation.

Shifter. It is a logarithmic shifter with a maximum shift width of 16 bits. As depicted in [19], for large shift values, the logarithmic shifter is effective both in terms of area and speed, therefore, it is used for MorphoSys implementation.

Critical path. The critical path of the RC, which is also the critical path of the MorphoSys M1 chip, includes a 16-to-1 mux, a 16×12 bits multiplier, a 4-to-1 mux, a 28 bits ALU, a 8-to-1 mux, and a 16 bits shifter. We have performed SPICE simulations for the entire RC. In order to consider the effects of long wires and large fan-out, the maximum possible load of each RC is computed and replaced by the equivalent capacitance in the SPICE input file. The critical path delay of the RC is 9.5 ns. Each RC consumes 200 mw of power at 100 MHz (25°C) when the multiplier is activated.

4.2. *TinyRISC*

TinyRISC [10] is a simplified 32-bit MIPS RISC processor, which has four pipeline stages: fetch, decoder, execute, and write back stages, as shown in the Fig. 8. The MorphoSys decoder, a sub-component in the decode stage, decodes the TinyRISC instructions that are specifically for MorphoSys. The MorphoSys decoder activates the DMA Controller to transfer data, provides control signals to the RC Array to execute operations

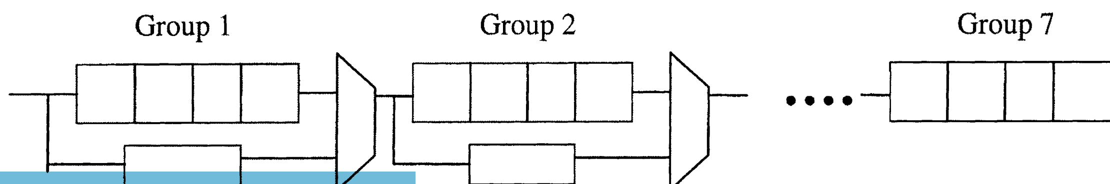


Figure 7. 28 Bits carry-skip adder.

www.manaraa.com

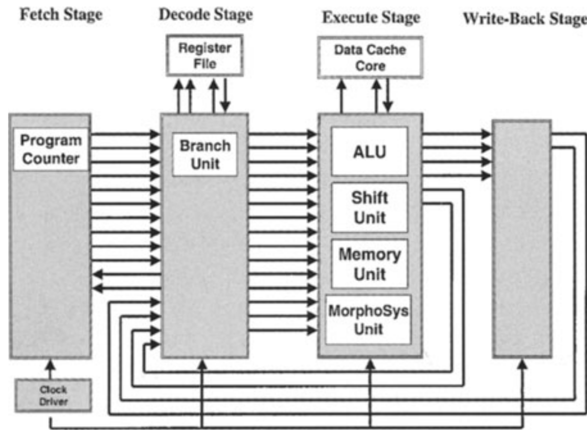


Figure 8. TinyRISC architecture.

defined by the configuration context. It also establishes communication among TinyRISC and DMA Controller, Frame Buffer, Context Memory and RC Array.

The register file consists of sixteen registers. It is a standard SRAM [19] design with precharge and sense amplifier. Each SRAM cell has one write port and two read ports. The register file is written in the first half of the clock cycle and read in the second half, which supports data forwarding.

All the TinyRISC components except for the register file and data cache, are synthesized using Synopsys and Mentor Graphics tools.

4.3. *Context Memory*

The Context Memory stores the configuration program (context) for the RC Array. The Context Memory is logically organized into two *context blocks*, each block containing eight *context sets*. Each context set has sixteen *context words*.

The major focus of the RC Array is on data-parallel applications, which exhibit a definite regularity. Following this principle of regularity and parallelism, the context is broadcast on a row/column basis. The context words from one context memory block are broadcast along the rows, while context words from the other block are broadcast along the columns. Each block has eight context sets and each context set is associated with a specific row (or column) of the RC Array. The context word from the context set is broadcast to all eight RCs in the corresponding row (or column). Thus, all RCs in a row (or column) share a context word and perform the same operations.

Thus, each row (column) of the RC Array receives a context word every clock cycle, from the Context Memory. This context word is stored in the Context Register of each RC (Section 2.1). This context word has different fields, as defined in Fig. 3. The field ALU_OP specifies ALU function. The control bits for Mux A and Mux B are specified in the fields MUX_A and MUX_B. Other fields determine the registers to which the result of an operation is written (REG #), and the direction (RS_LS) and amount of shift (ALU_SFT) applied to output.

The 12 LSBs of the context word represent the constant field. This field is used to provide an operand to a row/column of the RC directly through the context word. It is useful for operations that involve constants, such as multiplication by a constant. However, if such an operation is not needed, some of the extra bits in the constant field may be used to specify an ALU-Multiplier sub-operation. These sub-operations allow expansion of the functionality of the ALU unit.

The Context Memory is implemented using a standard CMOS SRAM cell with one read port and one write port [19]. The block diagram of the Context Memory is shown in Fig. 9. Corresponding to either row/column broadcast of the context word, a set of eight context words can specify the complete configuration (*context plane*) for the RC Array. As there are sixteen context words in a context set, up to sixteen context planes may be simultaneously resident in each of the two blocks of Context Memory.

Dynamic Reconfiguration. When the Context Memory needs to be changed in order to perform some

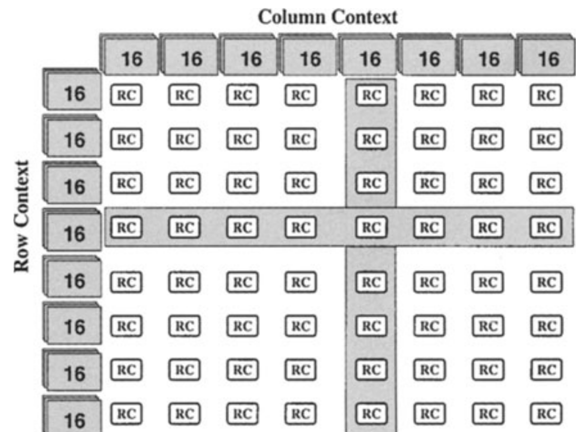


Figure 9. Context Memory.

المنارة للاستشارات
www.manaraa.com

different part of an application, the context update can be performed concurrently with RC Array execution. This dynamic reconfiguration enables the reduction of effective reconfiguration time to zero.

Selective Context Enabling. This implies that only one specific row or column may be enabled for operation in the RC Array. This feature is primarily useful in loading data into the RC Array. Since context can be used selectively, and because data bus design allows loading of one column at a time, one set of context words can be used repeatedly to load data into all eight columns of the RC Array. Without this feature, eight context planes (out of 32 available) would be required just to read/write data. This feature also allows irregular operations in RC Array, for e.g. zigzag re-arrangement of array elements.

4.4. Frame Buffer

An important component of the design is the Frame Buffer which serves as a data cache for the RC Array. The Frame Buffer consists of two sets of identical data memory (see Fig. 10). Each set consists of two banks of memory with each bank having 64×8 bytes of storage. These two sets of the Frame Buffer help make the memory accesses transparent to the RC Array, by overlapping of computation with the data load and store, alternately using the two sets. MorphoSys performance benefits greatly from the streaming process of this data buffer.

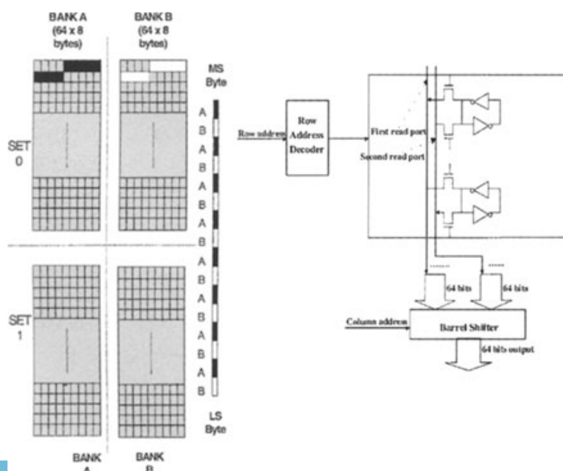


Figure 10. Frame Buffer structure.

Byte Addressing. An important feature of the Frame Buffer is the ability to provide any eight consecutive bytes of data to RC array in one clock cycle. As shown in Fig. 10, the Frame Buffer is implemented using an SRAM cell with two read ports and one write port.

To access eight consecutive bytes of data, the decoder enables the first read port of the associated decoded row and the second read port of the row next to the decoded row address. Then, these two rows of data are concatenated and a barrel shifter is used to select the desired eight bytes based on the column address. Figure 11 shows the block diagram of the Frame Buffer.

4.5. DMAC

The DMAC block handles all data/context transfers between Context Memory, Frame Buffer, and main memory. Three TinyRISC instructions for MorphoSys are used to direct the operations of DMAC.

The DMAC consists of three components: DMAC state machine, data register unit (DRU), and address generator unit (AGU) (Fig. 12). The DRU is used to pack or unpack data since the bus width between main memory and DMAC (32 bits) is different from the bus width between DMAC and Frame Buffer (64 bits). The AGU generates the addresses for the main memory and Frame Buffer when reading or writing Frame Buffer and Context Memory addresses during context loading. DMAC is synthesized using Synopsys and Mentor Graphics CAD tools.

4.6. Global Routing Network Layout

The global routing network consists of three parts: interconnection and data/context bus network, clock tree, and power/ground network. The RC interconnection network is comprised of three hierarchical levels.

Interconnection Network. The underlying network is the nearest neighbor layer that connects the RCs in a 2-D mesh (Fig. 13(a)). The second layer of connectivity is at the quadrant level (a quadrant is a 4×4 RC group), which provides complete row and column connectivity within a quadrant. Therefore, each RC can access data from any other RC in its row/column in the same quadrant. At the highest or global level, there are buses that support interquadrant connectivity (Fig. 13(b)). These buses are also called express lanes and they run across rows as well as columns. These

المنارة للاستشارات

www.manaraa.com

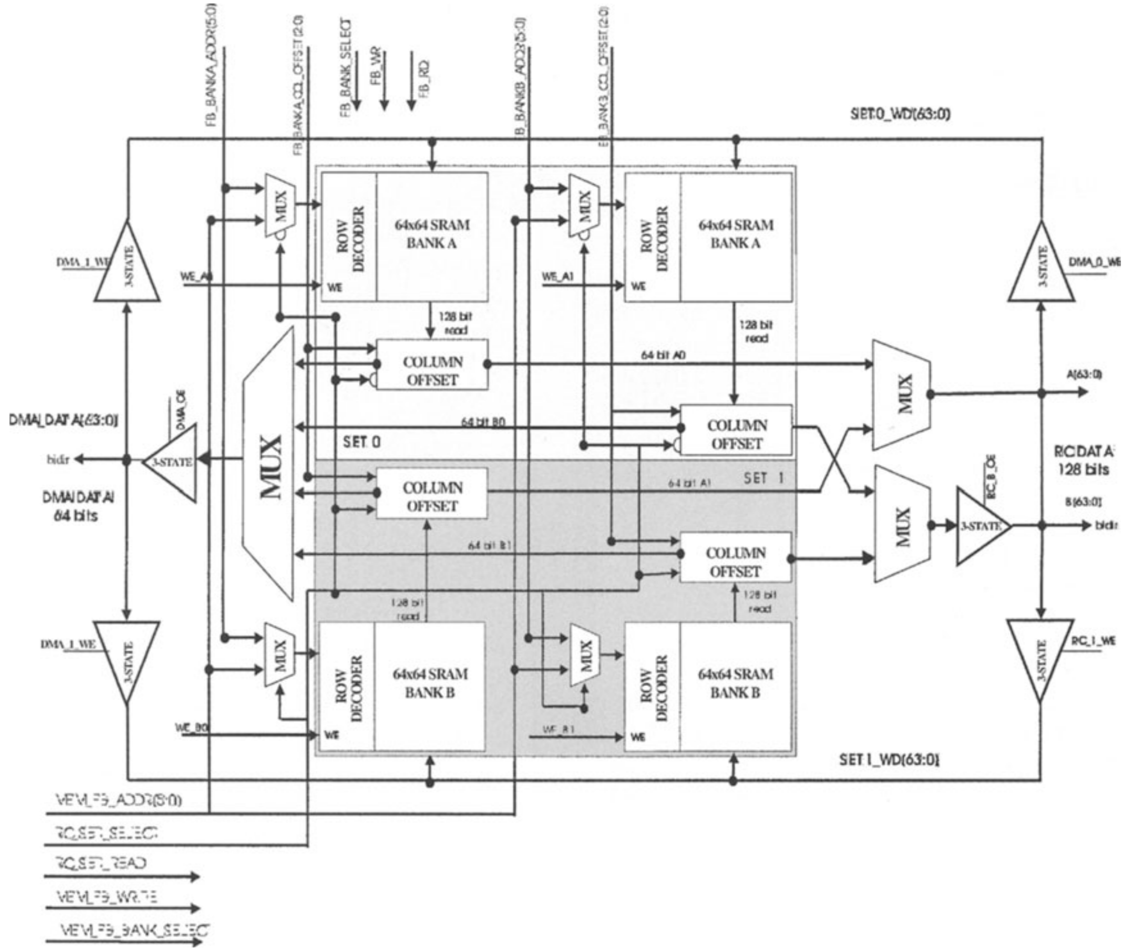


Figure 11. Block diagram of the Frame Buffer.

lanes can supply data from any one cell (out of four) in a row (or column) of a quadrant to other cells in adjacent quadrant but in same row (or column). Thus, up to four cells in a row (or column) may access the output value of any one of four cells in the same row (or column) of an adjacent quadrant. The express lanes greatly enhance global connectivity. Even irregular communication patterns, that otherwise require extensive interconnections, can be handled quite efficiently. For example, an eight-point butterfly is accomplished in only three clock cycles.

Data Bus. A 128-bit data bus from the Frame Buffer to RC array is linked to column elements of the array. It provides two eight bit operands to each of the eight column cells. It is possible to load two operand data (Port A and Port B) in an entire column in one cycle. Eight cycles are required to load the entire RC array. The outputs of RC elements of each column are written back to Frame Buffer through Port A data bus.

Context Bus. When a Tiny RISC instruction species that a particular group of context words be executed, these must be distributed to the Context Register in each RC from the Context Memory. The context bus communicates this context data to each RC in a row/column depending upon the broadcast mode. Each context word is 32 bits wide, and there are eight rows (columns), hence the context bus is 256 bits wide.

The dense connectivity of the interconnection network makes it difficult for automatic routing tools to maintain regularity of the global routing. A preliminary run of the Mentor Graphics automatic router gave a highly irregular layout. Hence, the global routing

المنارة للاستشارات
www.manaraa.com

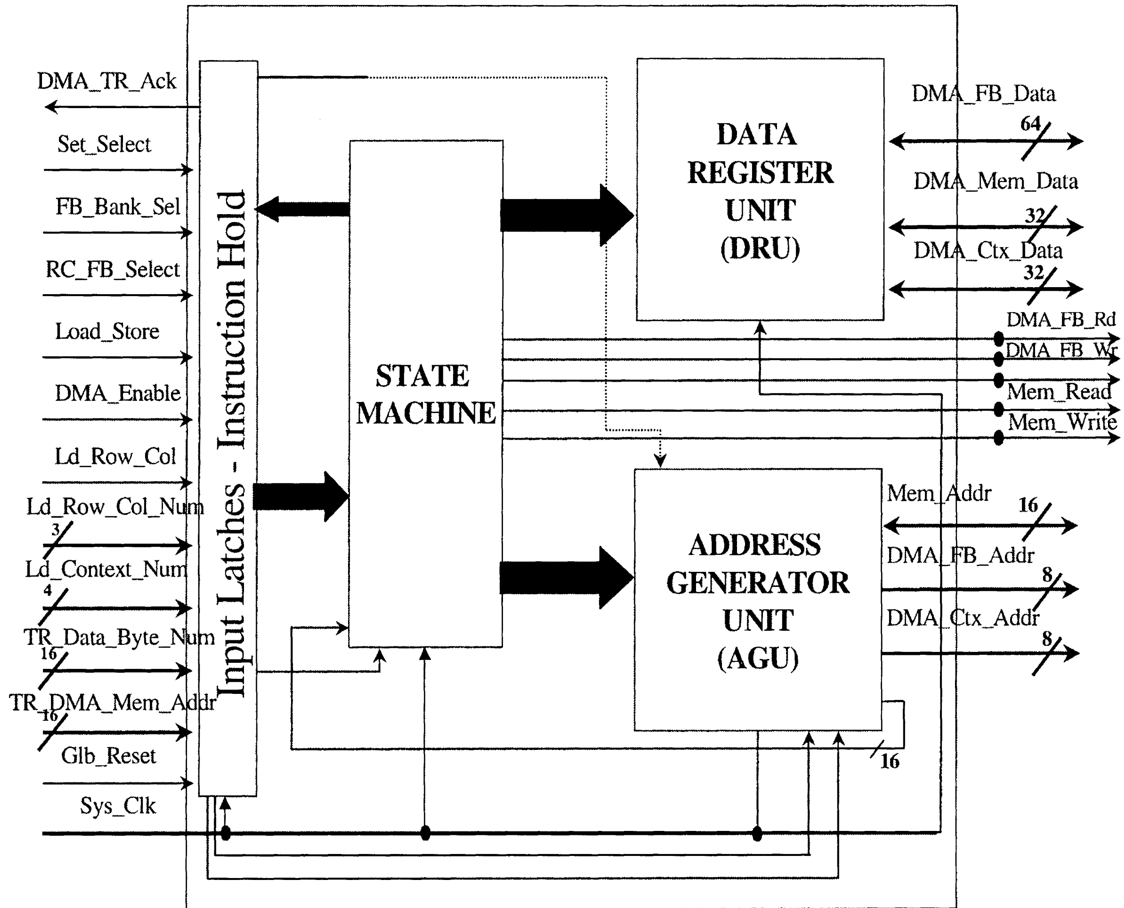


Figure 12. Internal block diagram of the DMAC.

layout was done using a combination of procedural and custom design approach as below:

(1) The clock tree was done as a custom layout using an H tree [19] pattern with tree-levels of buffers to balance the clock skew of RC array as shown in Fig. 14. The clock delay was measured using SPICE simulation. The buffers were subsequently inserted to other components of MorphoSys (e.g. TinyRISC, DMAC, and Frame Buffer) to balance the delay.
(2) The minimum width of metal layers were used for power and ground (based on the technology electron-migration rules) and accordingly the routing channel as shown in Fig. 15 was added manually by editing the layout file.
(3) The regular pattern of the interconnection network were captured using function calls that perform the procedural routing for creating the layout. We partitioned the interconnection network into four types of connectivity:

(a) intra-quadrant row/column full connectivity.
(b) inter-quadrant context connectivity.
(c) inter-quadrant express lane connectivity.
(d) cross quadrant boundary connectivity.

For (a) and (b), the routing channels are fixed for all RCs. For (c) and (d), the channels switch direction when crossing the quadrant boundary. Once the pattern of each RC is figured out, the routing of the interconnection network can be performed easily.

4.7. *MorphoSys Layout*

Figure 16 shows the layout picture of MorphoSys M1 chip. The integration of the five components (RC Array, TinyRISC, Frame Buffer, Context Memory, and

المنارة للاستشارات
www.manaraa.com

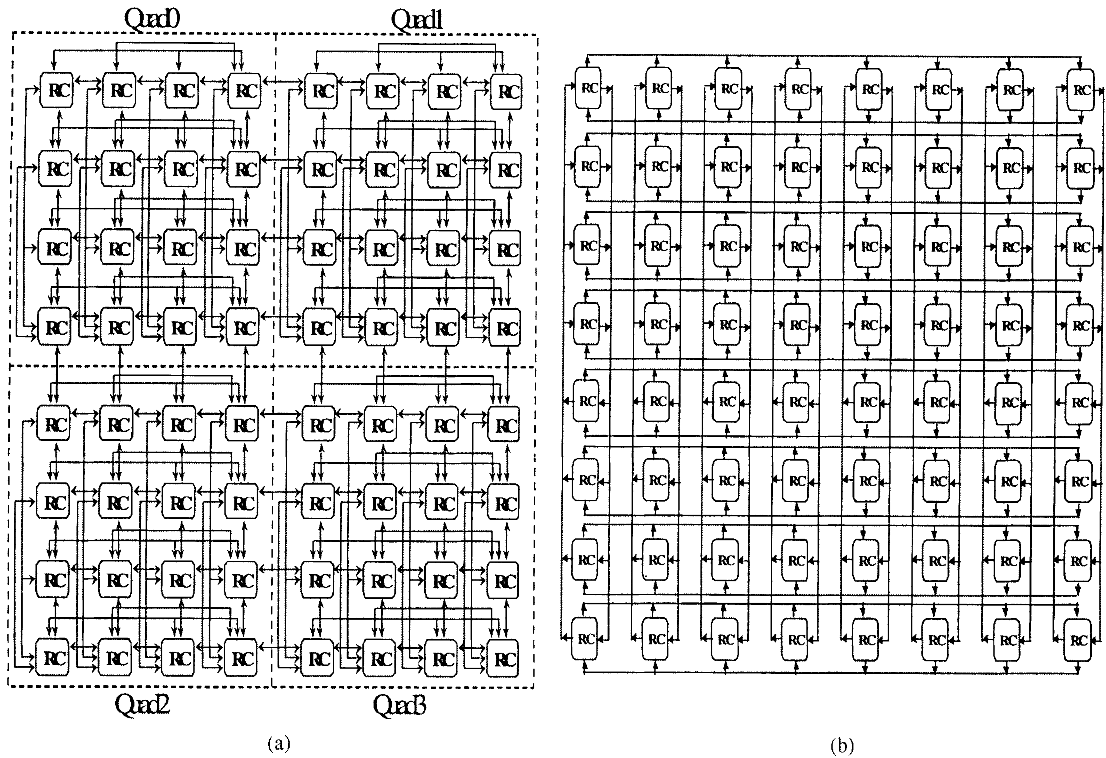


Figure 13. RC Array with interconnection network.

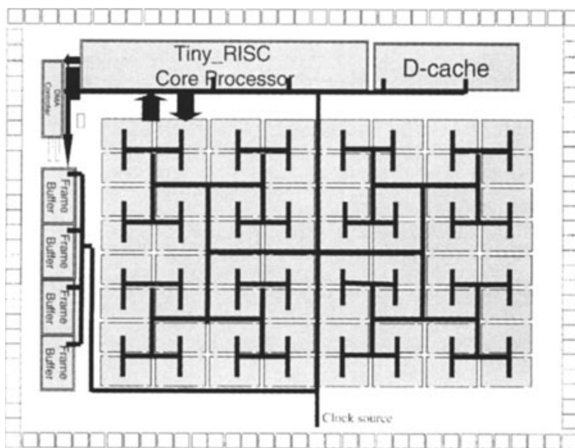


Figure 14. MorphoSys clock distribution.

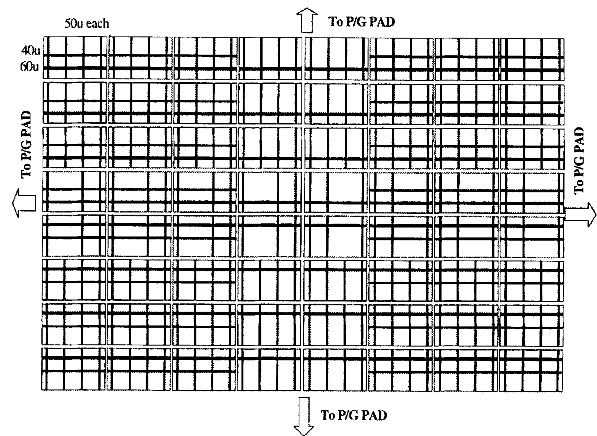


Figure 15. Power/ground routing channel.

DMAC) was carried out using Mentor Graphics CAD tools. The 8 × 8 RC Array, which is the largest component of the MorphoSys M1 chip, occupies more that 80% of the chip area. Each component of the MorphoSys has been fully verified and the VHDL model for the System has also been tested.

Table 4 lists the transistor count of each component and Table 5 summarizes the features of the MorphoSys M1 chip.

In Table 5, we provide three numbers for the power consumption. In the worst case where the multiplier of each RC is activated in each clock cycle, the power

المنارة للاستشارات

www.manaraa.com

Table 4. Transistor count of morphosys M1 chip.

Component	Transistor count
RC array	1,195,392
Frame buffer	139,106
TinyRISC	92,128
Context memory	105,096
DMAC	20,594
MorphoSys decoder	3,910
Main memory controller	1,180
Total	1,557,406

Table 5. MorphoSys M1 chip features.

Process technology	HP CMOS, 0.35 μm, 3.3 V, four-layer-metal
Area	14 mm × 12 mm
Transistor count	1,557,406
Peak performance	6.4 GOPS on 16 bits data
Clock frequency	100 MHz
Power consumption (100 MHz, 25°C)	15 W in the worst case $<$7 W for DCT $<$5 W for motion estimation
Pin count	240 (132 I/O, 54 power, 54 ground)

Table 6. Operation profile for DCT and motion estimation.

	DCT (% operation)	Motion estimation (% operation)
No-op	52.4%	29.2%
Reset	0%	1%
Addition	11.9%	36.5%
Subtraction	16.7%	0%
Absolute difference & accumulation	0%	33.3%
MAC (multiply/accumulate)	19%	0%

consumption is 15 W. However, among the applications we have investigated, there is no such case that the multiplier of each RC is enabled in each clock cycle. In order to get a more realistic measurement, we also estimated the power consumption for DCT and motion estimation. We use Hspice to simulate the power consumption of each function shown in Table 2 with the worst case scenario, which means each bit of the input switches every clock cycle. The power consumption of the MorphoSys is accordingly estimated based on percentage of each operation in our application mapping context shown in Table 6. The results show a difference from the worst case by more than a factor of 2.

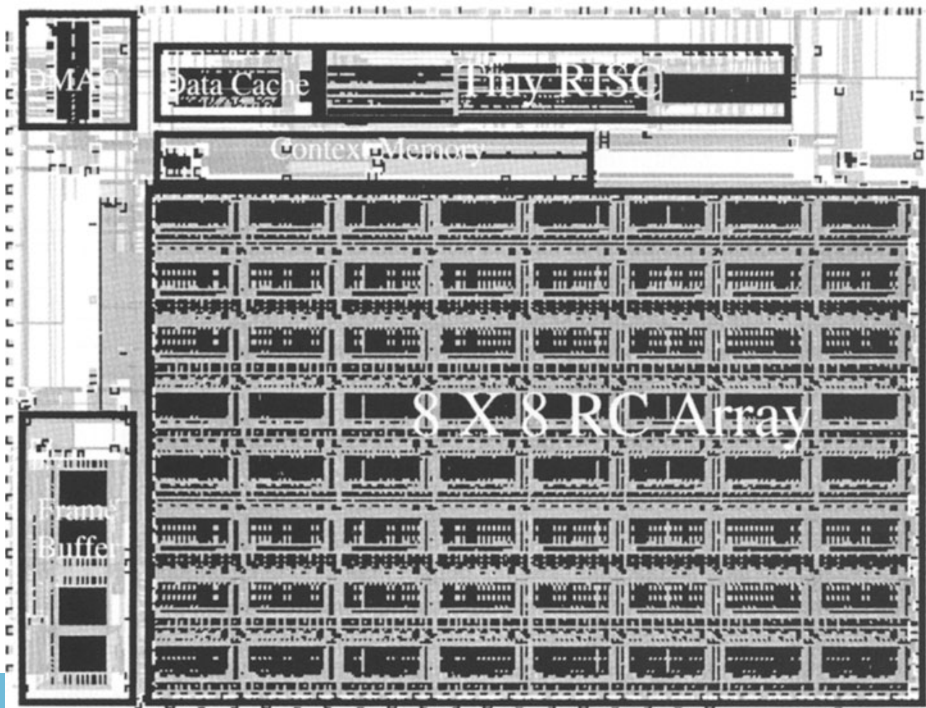


Figure 16. MorphoSys M1 layout picture.

للاستشارات
www.manaraa.com

5. Programming Environment

MorphoSim is the VHDL simulator for the MorphoSys reconfigurable computing processor. Through *MorphoSim*, one may efficiently verify RC Array performance using mapping algorithms for different applications, such as Motion Estimation, discrete consine transform (DCT), and automatic target recognition (ATR), and validate the physical design. MorphoSim needs 3 kinds of data: executable instruction, image data, and context data.

Furthermore, a graphical user interface, *mView*, has been developed in Tcl/Tk, that helps to visualize the data movement, and changes of the configuration in RC Array. The Perl based parser, mLoad, is used to generate the configuration contexts.

Another important aspect of our research is an effort to develop a programming environment for automatic mapping and code generation for MorphoSys. We have developed a compiler, *mCom*, to compile hybrid code for the TinyRISC and the RC Array using the SUIF compiler environment [20]. This compiler requires users to manually partition the application between the TinyRISC and the RC Array, for example by inserting pragma directives. C code is then mapped into MorphoSys configuration words using a C to VHDL translator.

6. MorphoSys Performance Analysis

In this section, we discuss the performance analysis through mapping of video compression and automatic target recognition (ATR) on MorphoSys. Video compression has a high degree of data-parallelism and tight real-time constraints. ATR is one of the most computation-intensive applications with bit-level operations. We also provide performance estimates based on VHDL simulations.

6.1. *Video Compression: Motion Estimation for MPEG*

Motion Estimation is the most computation-intensive algorithm in MPEG. Among the different algorithms, full search block matching (FSBM) [21] involves the maximum computations, however, gives an optimal solution with low control overhead. The detail description of the mapping can be found in [11]. For a reference block size of 16×16 and image size of 352×288 pixels at 30 frames per second (MPEG-2 main profile, low level), the processing of an entire image takes about 21.0 ms on a 100 MHz MorphoSys. This is much faster than the frame period of 33.33 ms.

Table 7. Performance comparison for motion estimation.

	MorphoSys	ASIC [21]	ASIC [22]	Pentium MMX
# of clock cycles	1020	581	1159	29000
Processing time	10.2 μs	2.9 μ	5.8 μs	145 μs

MorphoSys performance is compared with two ASIC architectures implemented in [21, 22] for matching one 8×8 reference block against its search area of 8 pixels displacement (see Table 7). The number of processing cycles for MorphoSys is comparable to the cycles required by the ASIC designs. Pentium MMX takes about 29000 cycles which is almost thirty times more than MorphoSys. It should be noted that the two ASIC systems implemented in [21] and [22] used an older technology. The processing unit that constitutes the critical path in these two implementations is the absolute difference and accumulation unit. Based on our simulation, the two ASIC systems can operate at about 200 MHz in 0.35 μm technology. We used a 233 MHz for Pentium MMX implementation [23], which is the highest clock rate for 0.35 μm Pentium processor. Taking into account the clock rate, we depict the performance comparison in Table 7. The result shows that MorphoSys can deliver an order of magnitude performance speedup over general purpose processors.

6.2. *Video Compression: Discrete Cosine Transform (DCT) for MPEG*

The forward and inverse DCT are used in MPEG encoders and decoders. In the following analysis, we consider an algorithm for fast 8-point 1-D DCT [24]. It involves 16 multiplications and 26 additions, leading to 256 multiplications and 416 additions for a 2-D implementation. The 1-D algorithm is first applied to the rows (columns) of an input 8×8 image block, and then to the columns (rows). The eight row (column) DCTs may be computed in parallel.

The cost for computing 2-D DCT on an 8×8 block of the image is as follows: 6 cycles for butterfly, 12 cycles for both 1-D DCT computations and 3 cycles are used for re-arrangement and scaling of data (giving a total of 21 cycles). This estimate is verified by VHDL

المنارة للاستشارات
www.manaraa.com

Table 8. Performance comparison for DCT/IDCT.

	MorphoSys	REMARC	V830R/AV	Pentium MMX
# of clock cycles	21	54	201	240
Processing time	210 ns	540 ns	1005 ns	1200 ns

simulation. Assuming the data blocks to be present in the RC Array (through overlapping of data load/store with computation cycles), it would take 0.49 ms for MorphoSys to compute the DCT for all 8×8 blocks (396×6) in one frame of a 352×288 image. The cost of computing the 2-D IDCT is the same, because the steps involved are similar. Context loading time is quite significant at 270 cycles. However, this effect is minimized through transforming a large number of blocks (typically 2376 blocks) before a different configuration is loaded.

MorphoSys requires 21 cycles to complete 2-D DCT (or IDCT) on 8×8 block of pixel data. This is in contrast to 240 cycles required by Pentium MMX™ [23]. Even a dedicated superscalar multi-media processor [25] requires 201 clocks for the IDCT. REMARC [9] takes 54 cycles to implement the IDCT, even though it uses 64 nano-processors. For the comparison of processing time, we use the clock rate of 200 MHz V830R/AV as presented in [25] although V830R/AV is implemented using 0.25 μm technology. REMARC has similar processing power (in terms of processing elements) to MorphoSys, so we assume 100 MHz for REMARC. The comparison is summarized in Table 8.

6.3. *Automatic Target Recognition (ATR)*

Automatic Target Recognition (ATR) is the machine function of detecting, classifying, recognizing, and identifying an object without human intervention. The ATR processing model [26] developed at Sandia National Laboratory has been mapped to MorphoSys [12].

For performance analysis, we chose the system parameters that were used in [26]. The ATR systems implemented in [26] and [27] were used for comparison. Two Xilinx 4013 FPGAs (one dynamic FPGA for most of the computations and one static FPGA for control) are used in Mojave [26], and Splash 2 system (consisting of 16 Xilinx 4010 chips) is discussed in [27]. For this study, the image size is 128×128 pixels, and the size of the target template is 8×8 bits. Table 9 summarizes the results of our comparison. For 16 pairs of target templates, the processing time is approximately 30 ms in the 100 MHz MorphoSys. This processing time is about an order of magnitude less than the 210 ms processing time of Mojave and 195 ms of Splash 2.

Table 9. ATR performance comparison

System	MorphoSys	Mojave	Splash 2
Processing time (before scaling)	30 ms	210 ms	195 ms
Processing time (after scaling)	30 ms	70 ms	65 ms

MorphoSys operates at 100 MHz, whereas, Mojave and Splash 2 run at 12.5 MHz and 19 MHz respectively. These two systems operate at a lower clock frequency than MorphoSys because of the older technology (0.6 μm is used in Xilinx 4010 series) and long wire propagation delays (characteristic of FPGAs). Without loss of generality, we can scale the clock frequency by a factor of 3 when counting for 0.35 μm technology (Xilinx data sheet [28]) used for the design of MorphoSys. After this scaling is taken into account, the computation time of Mojave and Splash 2 are 70 ms and 65 ms respectively. MorphoSys still outperforms Mojave and Splash 2 by a factor of 2. Although MorphoSys is a coarse-grained system, it achieves better performance compared to the FPGA-based systems (after accounting for speed scaling) for this fine-grain application. The FPGA-based systems are more appropriate for bit-level operations, and are inefficient for coarse-grain operations. These results demonstrate the flexibility of MorphoSys.

7. Conclusions and Future Directions

In this paper, we have presented the architecture and functionality of the MorphoSys, the design methodology and physical implementation of the MorphoSys M1 chip. We have also described the simulation environment—MorphoSim and MorphoSys compiler—mCom, and provided a comparative performance evaluation for applications such as Motion Estimation, DCT, and ATR mapped on MorphoSys. MorphoSys represents the implementation of a high performance reconfigurable system by integrating a general-purpose microprocessor with an array of coarse-grained reconfigurable cells.

Currently, the PCB design for the MorphoSys M1 system is under development. The PCB will include

المنارة للاستشارات
www.manaraa.com

the M1 chip, two banks of memory, and a standard PCI bus controller. Finally, this PCB will be plugged into PCI bus slot in the host PC to do the final test and the real performance evaluation. Meanwhile, we are continuing to develop the current compiler to provide the automatic partitioning of applications into sequential and data-parallel parts.

Acknowledgments

This research is funded by the Defense and Advanced Research Projects Agency (DARPA) of the Department of Defense under the contract number F-33615-97-C-1126.

References

1. W.H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V.K. Prasanna, and H.A.E. Spaaneburg, "Seeking Solutions in Configurable Computing," *IEEE Computer*, 1997, pp. 38–43.
2. S. Brown and J. Rose, "Architecture of FPGAs and CPLDs: A Tutorial," *IEEE Design and Test of Computers*, vol. 13, no. 2, 1996, pp. 42–57.
3. E. Tau, D. Chen, I. Eslick, J. Brown, and A. DeHon, "A First Generation DPGA Implementation," *FPD'95, Canadian Workshop of Field-Programmable Devices*, May 1995.
4. J.R. Hauser and J. Wawrzynek, "Grap: A MIPS Processor with a Reconfigurable Co-processor," *Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines*, 1997.
5. D.C. Chen and J.M. Rabaey, "A Reconfigurable Multi-processor IC for Rapid Prototyping of Algorithmic-Specific HighSpeed Datapaths," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 12, 1992.
6. E. Mirsky and A. DeHon, "MATRIX: A Reconfigurable Computing Architecture with Configurable Instruction Distribution and Deployable Resources," *IEEE Symposium on FCCM*, pp. 157–166, 1996.
7. C. Ebeling, D. Cronquist, and P. Franklin, "Configure Computing: The Catalyst for High-performance Architectures," *Proceedings of IEEE International Conference on Application-specific Systems, Architectures and Processors*, July 1997, pp. 364–372.
8. T. Miyamori and K. Olukotun, "A Quantitative Analysis of Reconfigurable Coprocessors for Multimedia Applications," *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, April 1998.
9. J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agrawal, "The RAW Benchmark Suite: Computation Structures for General-Purpose Computing," *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM 97, 1997, pp. 134–143.
10. A. Abnous, C. Christensen, J. Gray, J. Lenell, A. Naylor, and N. Bagherzaheh, "Design and Implementation of TinyRISC Microprocessor," *Microprocessors and Microsystems*, vol. 16, no. 4, 1992, pp. 187–194.
11. H. Singh, M. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, T. Lang, R. Heaton, and Filho, "Morphosys: An Integrated Re-configurable Architecture," *NATO Symposium on Concepts and Integration*, April 1998.
12. M. Gokhale, W. Holmes, A. Kopser, S. Lucas, R. Minnich, D. Sweely, and D. Lopresti, "Building and Using a Highly Parallel Programmable Logic Array," *IEEE Computer*, 1991, pp. 81–89.
13. K. Yano, T. Yamanaka, T. Nishida, M. Saito, K. Shimohigashi, and A. Shimizu, "A 3.8-ns CMOS 16 × 16-b Multiplier Using Complementary Pass-Transistor Logic," *IEEE Journal of Solid-State Circuits*, vol. 25, no. 2, 1990, pp. 388–395.
14. T.K. Callaway and E.E. Swartzlander, Jr., "The Power Consumption of CMOS Adders and Multipliers," *Low Power CMOS Design*, A. Chandrakasan and R. Brodersen (Eds.), IEEE Press, 1998.
15. C.S. Wallace, "A Suggestion for a Fast Multiplier," *IEEE Transactions on Electronic Computer*, vol. EC-13, 1964, pp. 14–17.
16. L. Dadda, "Some Schemes for Parallel Multipliers," *Alta Freq.*, vol. 34, 1965, pp. 349–356.
17. C.R. Baugh and B.A. Wooly, "A Two's Complement Parallel Array Multiplication Algorithm," *IEEE Transactions on Computer*, vol. C-22, no. 12, 1973, pp. 1045–1047.
18. I. Koren, *Computer Arithmetic Algorithms*, Prentice Hall Inc., 1993.
19. J.M. Rabaey, *Digital Integrated Circuits A Design Perspective*, Prentice Hall Inc., 1996.
20. SUIF Compiler system, The Stanford SUIF Compiler Group, http://suif.stanford.edu.
21. C. Hsieh and T. Lin, "VLSI Architecture For Block-Matching Motion Estimation Algorithm," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 2, 1992, pp. 169–175.
22. K.-M. Yang, M.-T. Sun, and L. Wu, "A Family of VLSI Designs for Motion Compensation Block Matching Algorithm," *IEEE Transactions on Circuits and Systems*, vol. 36, no. 10, 1989, pp. 1317–1325.
23. Intel Application Notes for Pentium MMX, http://developer.intel.com/drg/mmx/appnotes/
24. W.-H. Chen, C.H. Smith, and S.C. Fralick, "A Fast Computational Algorithm for the Discrete Cosine Transform," *IEEE Transactions on Communication*, vol. COM-25, no. 9, 1997.
25. T. Arai, I. Kuroda, K. Nadehara, and K. Suzuki, "V830R/AV: Embedded Multimedia Superscalar RISC Processor," *IEEE MICRO*, 1998, pp. 36–47.
26. J. Villasenor, B. Schoner, K. Chia, C. Zapata, H.J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith,"Configurable Computing Solutions for Automatic Target Recognition," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machine*, April 1996.
27. M. Rencher and B.L. Hutchings, "Automated Target Recognition on SPLASH 2," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machine*, April 1997.
28. XC 4000 Series High-Density Strategy, http://www.xilinx.xom.

المنارة للاستشارات
www.manaraa.com

Ming-Hau Lee received the Bachelor of Science degree from the Department of Electrical Engineering at the National Taiwan University, Taipei, Taiwan in 1993. In 1997, he received the M.S. degree in Electrical and Computer Engineering from the University of California, Irvine (UCI) where he is currently working toward the Ph.D. degree. He is currently a research assistant at the UCI Reconfigurable Computing Laboratory. His research interests are in computer architecture, reconfigurable computing, and VLSI design.
mlee@ece.uci.edu

Hartej Singh received the Bachelor of Technology degree in Electrical Engineering from the Thapar Institute of Engineering and Technology, Patiala, India in 1993. He received the M.S. degree from the Department of Electrical and Computer Engineering at the University of California, Irvine (UCI) in 1997. He is currently a doctoral candidate at the University of California, Irvine, and is associated with the MorphoSys project at the UCI Reconfigurable Computing Laboratory. His areas of interest are architectural mapping of multimedia applications, reconfigurable computing, and VLSI systems design.
hsingh@ece.uci.edu

Guangming Lu received the Bachelor of Engineering degree in Electronic Engineering from the Tsinghua University, Beijing, China in 1994. From 1994 to 1996, he was a graduate student of the Department of Computer Science at the Peking University, Beijing, China. He received the M.S. degree from the Department of Electrical and Computer Engineering at the University of California, Irvine (UCI) in 1997, where he is currently working toward the Ph.D. degree. Since 1997, he has been a research assistant at the UCI Reconfigurable Computing Laboratory. His areas of interest are VLSI systems design, CAD tools, Design Automation, Communication and Computer Architecture.
glu@ece.uci.edu

Nader Bagherzadeh received the Ph.D. degree in Computer Engineering from the University of Texas at Austin in 1987. From 1980 to 1984, he was with AT&T Bell Laboratories in Holmdel, New Jersey. Since 1988, he has been with the University of California, Irvine where he is currently a Professor and the Chair of the Electrical and Computer Engineering Department. His research interests are in parallel processing, reconfigurable computing, computer architecture, and VLSI design.
nader@ece.uci.edu

Fadi J. Kurdahi received the Bachelor of Engineering degree in Electrical Engineering from the American University of Beirut, Beirut, Lebanon in 1981. He received the M.S. degree in Electrical Engineering and the Ph.D. degree in Computer Engineering from the University of Southern California, Los Angeles, CA, in 1982 and 1987, respectively. Since 1987, he has been with University of California, Irvine where he is currently a Professor of Electrical and Computer Engineering Department and holds a courtesy appointment with the Information and Computer Science Department. He received an NSF Research Initiation Award in 1989, and two ACM/SIGDA fellowships in 1991 and 1992. His areas of interest are high-level synthesis of digital circuits, VLSI systems design and layout, and design automation.
kurdahi@ece.uci.edu

المنارة للاستشارات

www.manaraa.com

Eliseu M.C. Filho received an EE degree from University of Brasilia, Brazil, in 1983, an M.Sc. degree in Informatics from Catholic University of Rio de Janeiro, Brazil, 1987, and a D.Sc. degree in Systems and Computer Engineering from Federal University of Rio de Janeiro (UFRJ), Brazil, in 1994. Currently, he is an Associate Professor at the Department of Systems and Computer Engineering at COPPE/UFRJ. His interests include computer architecture, instruction-level parallelism, reconfigurable systems and VLSI systems.
eliseu@lam.ufrj.br

Vladimir Castro Alves obtained his electronic engineering degree from the Federal University of Rio de Janeiro, Brazil, in 1985. He obtained his DEA and Ph.D. degrees from the Institut National Polytechnique de Grenoble, France, in 1989 and 1992 respectively. He held an Eurochip Lecturer position in the University of Aveiro in 1993 and 1994 where he has been responsible for the VLSI design Laboratory. He joined in 1994 the Federal University of Rio de Janeiro, Brazil, where he is presently an Associate Professor at the Electrical Engineering Department. His current research interests are in VLSI design for high performance computing, high-level synthesis for testability and analog integrated circuits and memory testing.
castro@lpc.ufrj.br

المنارة للاستشارات
www.manaraa.com

Journal of VLSI Signal Processing Systems 24, 165–180, 2000.
© 2000 Kluwer Academic Publishers. Printed in The Netherlands.

Co-Synthesis to a Hybrid RISC/FPGA Architecture

MAYA B. GOKHALE* AND JANICE M. STONE
Sarnoff Corporation, Princeton, NJ

EDSON GOMERSALL
National Semiconductor Corporation, Santa Clara, CA

Abstract. Hybrid architectures combining conventional processors with configurable logic resources enable efficient coordination of control with datapath computation. With integration of the two components on a single device, housekeeping tasks and, optionally, loop control and data-dependent branching, can be handled by the conventional processor, while regular datapath computation occurs on the configurable hardware. This paper describes a novel approach to programming such hybrid devices that gives the programmer control over mapping of data and computation between conventional processor and configurable logic. With a simple set of pragma and intrinsic function directives, the NAPA C language provides for manual control over perhaps the most important aspect of programming such hybrid devices. Alternatively, as experience is gained about tradeoffs between the two computational resources, mapping directives may eventually be generated by an external tool. The paper further describes a research prototype compiler that targets the hybrid processor model, with a concrete implementation for the National Semiconductor NAPA1000 chip. The NAPA C compiler parses the mapping directives, performs semantic analysis, and co-synthesizes a conventional processor executable combined with a configuration bit stream for the configurable logic. Two major compiler phases, the synthesis of pipelined loops and the datapath synthesis, are described in detail.

1. Introduction

In recent years, a new architecture for system-on-a-chip has emerged, in which a simple controller, configurable logic, and fixed function units such as dedicated memories are combined on a single device [1–5]. Such devices appear attractive relative to conventional Field Programmable Gate Arrays (FPGA) because the tightly-coupled on-board controller can manage state transitions, control flow, and configuration management, aspects of a computation that are tedious, error-prone, and sometimes not possible to manage from within the FPGA. Since FPGA state is readily accessible to the main processor with single-cycle latency rather than at the other end of an I/O bus, such housekeeping functions can be managed efficiently by the processor. At the same time, processor state can be communicated to the FPGA circuits equally quickly, controlling inner-loop, compute-intensive datapath blocks. The compute-intensive tasks being realized in programmable hardware circuits can provide an order of magnitude or more performance boost over conventional software on suitable programs [6].

In short, this new capability enables application developers to shift focus rapidly between conventional processor and configurable logic without losing appreciable performance to transfer control and data between the two components.

Unfortunately, there is a dearth of tools that target such devices. Conventional compilers can generate object files for the RISC processor, but do not synthesize hardware for the parts of the computation mapped to the FPGA. Similarly, while many tools exist to synthesize logic from schematics, VHDL, or Verilog, these tools do not address the part of the computation mapped to the RISC processor. Further, current tools do not address the implicit communication that must occur between the two components as the focus of control shifts between them.

*Current affiliation: Los Alames National Laboratory, Los Alames, NM.

www.manaraa.com

In this paper we present NAPA C, a language and compiler that address the hardware-software co-synthesis problem in the context of hybrid RISC/FPGA processors. NAPA C constructs allow the programmer to explicitly map data and computations to either RISC processor or FPGA. The NAPA C compiler generates a conventional C program that contains portions of the computation assigned to the RISC processor as well as C code to control circuits generated for the FPGA. Through the Malleable Architecture Generator (MARGE) datapath synthesis tool, the compiler generates, for the computation mapped to the FPGA, a Verilog netlist that utilizes highly optimized pre-placed, pre-routed macro generators. The compiler targets the NAPA1000 hybrid processor, a chip which combines a small RISC processor (the Fixed Instruction Processor or FIP) with configurable logic (the Adaptive Logic Processor or ALP). The NAPA C compiler supports regular datapath computation on the ALP. The compiler also recognizes "ALP functions," which serve as new CISC instructions to augment the RISC instruction set. In addition, the compiler analyzes each C loop whose body is mapped to the ALP and where possible, generates hardware pipelines for pipelineable ALP loops. While we demonstrate a mapping of our compiler to a concrete realization, the NAPA1000, the compiler output is in a generic form mapping to a well defined hybrid RISC/FPGA API. The RISC output is in the form of an ANSI C program compilable by any ANSI C native compiler. The emitted C code calls a runtime software library to perform interface tasks such as load configurations, initiate ALP circuits, and read back state from the ALP. The ALP circuits are described in several different formats such as Register-Transfer Level (RTL) VHDL, structural Verilog, and structural VHDL. Thus the compiler is easily retargetable to similar hybrid RISC/FPGA devices.

The NAPA C compiler was the first to target a hybrid FIP/ALP architecture. A major contribution of our work is the modular compiler structure based on the SUIF [7] compiler infrastructure and the MARGE datapath synthesis tool. As the NAPA1000 architecture was evolving while the compiler was being developed, the modular compiler structure allowed the compiler development to track the hardware modifications. The modular structure also allows us to easily add new capabilities. For example, we recently added a new memory partitioning phase [8] that automatically allocates variables to memories to optimize pipelined loop throughput.

Another major contribution of our compiler is that it allows application developers to iteratively refine their applications, starting with a conventional C program running 100% on the FIP, and gradually transferring functions to the ALP to accelerate performance. To augment automatic partitioning, the programmer can experiment with different manual partitioning strategies simply by re-arranging directives at the C source code level, and have the compiler generate the hardware, software, and communications code for each different partitioning. This capability is invaluable in order to explore tradeoffs between hardware and software in novel devices such as the Napa1000 and other hybrid devices.

In terms of compiler capability, our compiler is one of the first to synthesize hardware pipelines from C loops. Our datapath synthesis technique was the first to rely on hand-optimized pre-placed, pre-routed module generators, demonstrating superior area and delay characteristics as well as fast compilation.

This paper extends and modifies earlier reports [9, 10] of this work, sections of which are reused here, in accordance with the IEEE copyright agreement and with the permission of the authors.

The remainder of the paper is organized as follows. The next section reviews related work. Section 3 contains a brief overview of the NAPA1000 architecture. Section 4 describes the NAPA C directives for mapping between FIP and ALP. The NAPA C compiler is described in Section 5. Two major compiler phases, the synthesis of pipelined loops and the datapath synthesis, are described in detail. Finally we end with conclusions and future directions.

2. Related Work

A number of research projects address compiling for hybrid conventional RISC/FPGA architectures. The RAW project of MIT [11] is studying a systolic-array-like tiled architecture whose components are hybrid RISC/FPGA processors. The RAW team are studying compiler optimizations to exploit instruction-level parallelism with multiple instruction streams, and to partition and map computation to the array of tiles.

The Berkeley BRASS project [3] is designing a hybrid MIPS architecture that includes a reconfigurable coprocessor. The Garp compiler [12] and its extension to the Nimble compiler [13] has similar goals to those of the NAPA C compiler, but has focused more

المنارة للاستشارات www.manaraa.com

closely on extracting instruction-level parallelism by creating hyperblocks. The NAPA C compiler enlarges the size of basic blocks by converting if statements to guarded assignment statements, but does not do trace scheduling.

Oxford University researchers have developed a co-synthesis system that accepts specifications in SML and generates occam (software) and Handel (hardware) [14] components. In contrast our work is based on a more pragmatic level in which algorithms expressed in C are partitioned between hardware and software, similar to the Handel technology. One key difference is that C-like statements expressed in Handel convey timing requirements: each statement is synthesized to execute in a single clock cycle. Thus the timing characteristics of the generated hardware circuit are controlled by the programmer's method of coding rather than by the data dependencies of the computation.

Weinhardt discusses generation of pipelines for FPGA processors [15] with implementation on the Virtual Computer Corporation EV-1 of a vector computational model for FPGA computing. This work is similar to ours in that we also pipeline loops. It differs in that Weinhardt is concerned with virtualizing the single memory module of the EV-1, whereas our target architecture has numerous simultaneously accessible memories, which we exploit. More recent research is concerned with vectorization of loops using traditional compiler analysis and transformations [16].

3. The NAPA1000 Architecture

The NAPA1000 (Fig. 1, from [17]) integrates on a single chip a small embedded 32-bit RISC processor (Fixed Instruction Processor, or FIP) along with configurable logic, on-chip memory, and an interconnection network interface. The Compact RISC CR32 processor is an embedded processor proprietary to National Semiconductor.

The configurable logic consists of a 64×96 Adaptive Logic array (ALP) whose core cells are very similar to National Semi's CLAy™ architecture. The ALP is partially and dynamically reconfigurable. It has hierarchical routing connections, with nearest neighbor, minor, and major block interconnection. In addition, a layer of global interconnection is available either for communication to the RISC processor, I/O to the external world, or for general routing. Like the Xilinx XC6200 (a fine-grained FPGA memory mapped to the host address space), the ALP array is directly accessible

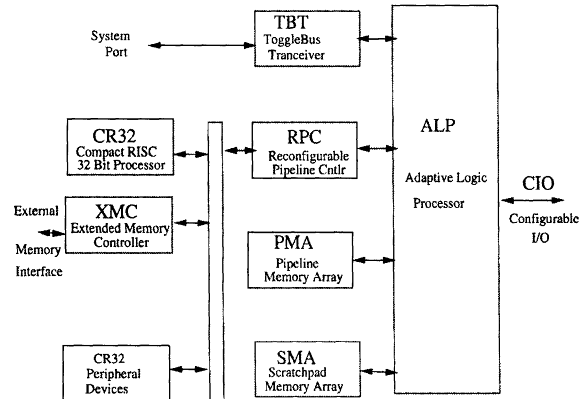


Figure 1. NAPA1000 block diagram.

from the FIP. It is a part of the FIP memory address space.

A small control module, the RPC (for Reconfigurable Pipeline Controller) performs configuration management and interface to the RISC processor. The Toggle Bus Transceiver connects a single NAPA1000 part to other NAPA1000s on a Multi Stage Interconnection Network. There are two 32 bits $\times$ 2 K on-chip memory banks (Pipeline Memory Array) and eight small 8 bits $\times$ 256 scratchpad memories (Scratchpad Memory Array). A Bus Interface Unit (labeled "CR32 Peripheral Devices" in Fig. 1) coordinates transfers between the NAPA1000 and external devices such as DRAM.

4. NAPA C Language

The NAPA C language presents a tightly integrated programming model encompassing data storage and computation on both FIP and ALP. These concepts are available to the programmer or automatic tool with explicit directives. The extensions provide the following capabilities:

1. indicate whether a variable will reside on the ALP as a register, in an ALP local memory, or in external memory accessible both to FIP and ALP.
2. indicate the bit length of an integer variable on the ALP.
3. indicate whether a subroutine is to be computed on the FIP or the ALP.
4. indicate whether one or more statements is to be computed on the FIP or the ALP.

www.manaraa.com

4.1. Language Extensions

There are several different styles that may be used to introduce these extensions into the language. The first is to add keywords and/or overload existing syntax in C. This approach limits portability. Another approach is to provide new classes in object-oriented languages such as C++ or Java. We have chosen not to use this approach because we are using the SUIF compiler infrastructure, which currently supports C, not C++.

The alternative we have chosen is to use compiler pragmas combined with calls to intrinsic functions. Pragma directives are commonly used in the high performance computing community to give the compiler out-of-band information on program mapping. For example, for vector supercomputers it is common to annotate the source program with vectorization directives. High Performance Fortran (HPF) [18] defines a set of directives to distribute arrays across processor nodes. In NAPA C, These "#" directives are processed by the compiler, and code is co-synthesized accordingly. Other C compilers will simply ignore the pragmas and compile the entire program to a FIP architecture. The advantage to this approach is that the program is in standard C. The C program, along with appropriate header files and libraries that define the behavior of intrinsic functions, can be compiled with pragmas and intrinsic calls in place and debugged on the workstation with a standard C compiler. It can also be compiled with the NAPA C compiler and simulated via the FIP/ALP simulator to tune performance by adjusting the program partitioning. The NAPA simulator is a cycle accurate combined FIP/ALP simulator. FIP operations are simulated in a simulation model of the CR32 processor. ALP operations are simulated by a logic simulator that models the behavior of core cells and routing resources. Finally, the program can be run on systems containing the NAPA1000 part or its follow-ons. Computation with non-standard bit lengths may perform differently on the workstation, the FIP, and the ALP. For those programs, the NAPA simulator is the best approach. For computation on 8-bit values such as pixels, accurate first-order simulation can be done on a workstation.

4.2. Data Definition Pragmas

Pragmas are used to define attributes of program variables. Pragmas can define either the location of the variables being declared (**loc** pragma) or the bit lengths of ALP register variables (**size** pragma). Each pragma begins with "#pragma ALP", and appears on a separate line. The word(s) following "#pragma ALP" select(s) among various options. The pragma is inserted after the normal C declaration of the variable(s) referenced. The options listed below tell the compiler where variables are to reside. There are five alternatives: ALP core cells; ALP scratchpad memory; ALP memory modules 1 or 2; external memory accessible to both FIP and ALP. External memory is the default alternative, which the compiler will assume in the absence of other pragmas. Arbitrary precision, up to the physical limit of the ALP area, is supported for ALP register variables, while variables in a memory must conform (or be coercible to) that memory's data word size.

```
int A[5] = {1, 2, 3,4, 5};
#pragma ALP loc m1 A
int b = 15, c = 10, d = 5;
#pragma ALP loc reg b c d
# pragma ALP size 4 b c d
```

Figure 2. ALP data declarations.

1. #pragma ALP loc reg ⟨variable-name-list⟩. This location directive means that the variables named will be allocated on ALP core cells as ALP "registers."
2. #pragma ALP loc m{0 | 1 | 2 | 3}. This location directive means that the variables will be allocated in a memory rather than on core cells. The data may get allocated in external memory accessible to both FIP and ALP (m0); ALP local memory (m1 or m2) or the scratchpad (m3).
3. #pragma ALP size ⟨bit-length⟩ ⟨variable-name-list⟩. This directive sets the bit length of the named variables to ⟨bit-length⟩.

Figure 2 show examples of pragmas defining location and size of ALP variables. The array `A` is assigned to memory bank 1. Variables `b`, `c`, and `d` reside on ALP core cells as ALP registers, and each is four bits in length.

4.3. ALP Function Pragma

To specify that an entire subroutine is to be executed on the ALP and that parameters to and from the subroutine are to be passed through the internal bus, the following pragma must be inserted following the function declaration:

www.manaraa.com

```
int Logic_Op( unsigned int x, unsigned int y,
              unsigned int z, unsigned int w)
{
#pragma ALP loc reg x y z w
#pragma ALP size 4 x y
#pragma ALP size 2 z w

  return ((x&y) ^ (x&~z) ^ (w|z));
}

#pragma ALP function Logic_Op size 4

void main()
{
  unsigned int a, b, c, d;

  /* FIP processing */
/* call ALP function */
  while (a = Logic_Op(a, b, c, d))
    a <<= 1;
}
```

Figure 3. An ALP function.

#pragma ALP function ⟨function-name⟩ size ⟨result-size⟩

The compiler synthesizes a circuit for the function body, with parameters copied from the internal bus to ALP registers. Upon function exit, the return value is written to the bus. In the FIP program, code is generated to

1. copy parameters to the internal bus,
2. activate the hardware function, and
3. copy the return value from the bus into FIP memory.

Figure 3 shows an example of using an ALP function from within a FIP loop. The function is called repeatedly until a termination condition is met. The example illustrates the fine-grained alternation of focus of control that is possible in this hybrid architecture. `Logic_Op` is effectively a new CISC instruction that augments the RISC instruction set. The low order four bits of a, b, c, and d are passed on the FIP-ALP communication bus, and the return value passed back on the same bus.

4.4. ALP Blocks

The following intrinsic functions bracket statements that are to be performed on the ALP. The directives indicate that the enclosed statements are to be synthesized as one or more ALP instruction, where an instruction may execute over multiple clock cycles.

alp_begin();
/* ALP-resident hardware structures will be */
/* synthesized from the C code here */
alp_end();

When the ALP function or ALP block is called, the FIP program initiates the ALP circuit, and then waits for the ALP computation to complete.

This collection of directives allows detailed specification of partitioning of data and computation between the FIP and ALP. The directives can be inserted manually or by an automatic system. Recent work has focused on automatically generating the data placement directives to optimize the throughput of compiler-generated pipelines [8] (see Section 5.2).

5. NAPA C Compiler

The NAPA C compiler, in conjunction with NAPA-1000-specific low-level tools, generates a combined FIP/ALP executable image. The compiler allocates the data to the desired memory space (ALP register, FIP memory, or one of the ALP memories). It generates ANSI C source for FIP subroutines and statements. The FIP native compiler is used to create the object module. The NAPA C compiler synthesizes hardware structures to represent the C code of the ALP subroutines and statements, and also generates FIP code to control execution of the ALP segments. When an ALP variable is referenced in the FIP code, the NAPA C compiler generates the code to fetch the variable into the FIP memory, which requires both FIP code as well as an ALP circuit.

5.1. Compiler Organization

As shown in Fig. 4, the NAPA C compiler consists of a number of phases. Input to the compilation system is ANSI C annotated with pragmas and intrinsic function calls as described above. The first phases are embedded in the SUIF compiler infrastructure. The pragmas are converted to SUIF annotations to the syntax tree. Semantic checks are performed to verify correct usage of the directives. Directives are propagated as annotations through the syntax tree. During this phase, data

المنارة للاستشارات www.manaraa.com

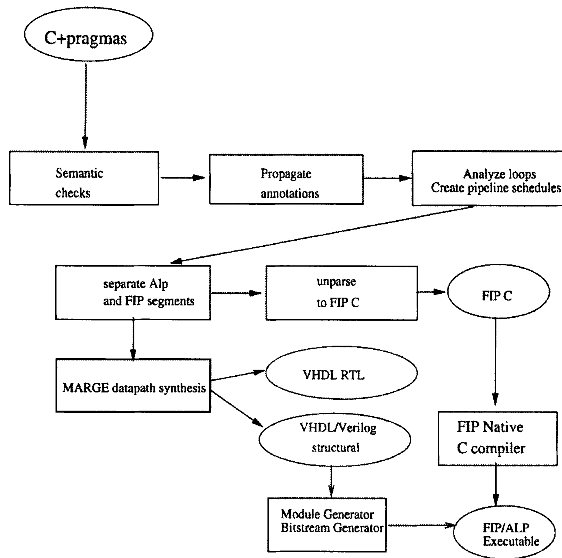


Figure 4. Organization of the NAPA C compiler.

transfers between FIP and ALP are inserted into the syntax tree. Such data transfers are needed, for example, when ALP data is accessed in a FIP computational block.

An analysis and scheduling phase processes program loops and generates pipeline schedules for pipelineable loops. Next, ALP segments are extracted from the syntax tree and passed to the MARGE datapath compiler. The remaining syntax tree contains purely FIP code, which is unparsed to C and processed by the RISC processor's C compiler. Our decision to unparse to C and use the native RISC compiler was based on uncertainty until fairly late in the design process of which conventional processor would be used. Since the generated FIP program is in standard C, our compiler is relatively platform independent, making the compiler portable across revisions of the NAPA1000 and even applicable to different hybrid FIP/ALP architectures.

The compiler backend MARGE synthesizes the hardware circuits for the ALP. As shown in Fig. 4, MARGE generates several equivalent representations of the hardware circuits. A Register-Transfer Level (RTL) representation in VHDL is generated, and can be processed by conventional synthesis tools. In addition, MARGE performs synthesis to generate a structural representation in Verilog and VHDL. The Verilog version calls macro module generators from a library provided by National Semiconductor [10]. The end result of the compilation is a combined FIP/ALP executable image. In the following sections, we describe two key compiler phases, the pipeline loop scheduling and datapath synthesis, in greater detail.

5.2. *Loop Analysis and Pipeline Scheduling*

Configurable logic is a natural candidate for pipeline synthesis. Hardware pipelines, with a long history of use, keep various stages of a unit busy by starting a new operation while the previous operation is still in progress. Software pipelining adapted this idea to a code sequence such as a loop: in some cases a new loop-iteration can be started while a previous iteration is in progress. The Napa C compiler's pipelining strategy for loops extends standard scheduling methods to pipeline loops. The algorithm implements the method described by Lam [19]. The key new ideas in this application of Lam's algorithm are the following:

1. Unlike Lam's VLIW target model, we are not constrained by a limited number of function units (eg. adders). We may synthesize as many function units as necessary to satisfy pipeline throughput goals. The limiting resource in our model is the number of memories and the location of data accessed in those memories during the loop.
2. The hardware we generate is capable of concurrent micro-operations, and we can exploit that concurrency to relax data dependency constraints of conventional software pipelining.

The pipeline scheduler uses a dependency graph in which the nodes correspond to ALP function units. The edges of the graph are precedence constraints. A pair (n_i, n_j) is a normal edge if, in any iteration of the loop, node n_i must be executed before node n_j. If (n_i, n_j) is a special edge, then, in any iteration node n_j must be not be executed before node n_i. A special edge permits concurrent execution of two nodes. This lets us model register use in which, during a single cycle, a value can be read from a register early in the cycle and written to the register late in the cycle.

The scheduler produces a schedule for a *regular* pipeline, in which every iteration is executed according to the same schedule, with successive iterations initiated at a fixed initiation interval s. A schedule gives a list of the nodes to be executed at each time step. The steady state of the pipeline is given by the last s stages of the schedule. Throughput is one iteration every s cycles.

المنارة للاستشارات
www.manaraa.com

The pipeline scheduler tries to find a schedule with successively increasing iteration intervals $s < m$, where m is the length of a nonpipelined schedule, which is the length of the longest path in the dependency graph. We use a technique suggested by Lam to find a lower bound for the initiation interval, based on multiple uses of resources, rather than beginning at $s = 1$.

The scheduling algorithm works as follows. For each node n in the dependency graph, the *precedence constrained range* for n is an interval $[t_1, t_2]$ in which n can be scheduled. For a graph containing N nodes, an upper bound on the length of the pipelined schedule with initiation interval s is $N + s$. The scheduler initializes the precedence constrained range for each node to $[0, N + s]$. The range is updated at each scheduling step, as described below, to preserve the partial order given by the dependency graph.

The modulo-s reservation table consists of s resource vectors *used* (i) that are initially zero. This models resource usage by time step modulo s, in an innovation that Lam introduced into software pipelining, extending the idea of a reservation table used by traditional hardware pipeline schedulers to track resources shared by pipeline stages.

The pipeline scheduler selects nodes in an order that ensures that all their predecessors in the partial order given by the dependency graph have already been scheduled. To schedule a node n, the scheduler first limits the range to the first s stages of n's precedence constrained range. This suffices because, with initiation interval s, at each stage t all the nodes scheduled at stages t *mod* s are executed. If a node cannot be scheduled within s stages, it cannot be scheduled. The scheduler places n at the first such stage t in which the resources it requires are available: *used* $(t \text{ mod } s) \cap$ *needs* $(n) = 0$ where $\cap$ is the logical AND operator.

Having chosen stage t for node n, the scheduler updates the vector *used* $(t \text{ mod } s)$ in the modulo-s reservation table. For each node n' that is a successor of n in the dependency graph, it updates the lower bound in the precedence-constrained range of n' from $[t_1, t_2]$ to $t_1^* = max(t_1, t + \delta)$ where δ is the length of the longest path from n to n'. Lam's algorithm and our implementation of it recognize a class of inter-iteration dependency termed *doacross*, in which a value computed in one iteration is used in a later iteration. If the newly scheduled node n is the target of a doacross edge in the dependency graph, then the scheduler also updates the precedence-constrained range of the node at the source of that edge.

When the hybrid system executes the program, it performs the pipelined loop as follows:

1. The FIP processor sends the ALP a signal to execute an initialization circuit to set up data and loop-control registers.
2. Then the FIP sends the ALP a signal to perform all iterations of the loop. On the ALP, the loop iterations are pipelined, with a new iteration started every s cycles, where s is the initiation interval determined for the schedule.
3. If there are final results to store, the FIP signals the ALP to execute a finalization circuit.

5.2.1. Pipelined Matrix Multiply. For an example of the Napa pipeline facility, consider the problem of matrix multiplication of two $N \times N$ matrices b[] and c[], to produce a result matrix a[]: for each pair i, j of indices, calculate

$$a[i, j] = \sum_{k=1}^{N} b[i, k] \times c[k, j]$$

Figure 5 shows the compiler intermediate 3-address code for the innermost loop of matrix multiplication. We assume the matrices b[] and c[] have been declared to reside in separate ALP memories. The initial assumptions listed in the figure are satisfied by an initialization circuit, not shown. The partial sum is accumulated in Reg3. After the inner loop is executed N times, a final circuit, not shown, is executed once to store the result to a[i,j].

```
                 Matrix Multiply Inner Loop
Initial assumptions:
    MAR0 gives the address of b[i,0]
    MAR1 gives the address of c[0,j]
    Reg3 = 0, used to accumulate the sum
    Reg4 = increment to the next element in row
    Reg5 = increment to the next element in column
    At completion, Reg3 gives the result.

1   LD    MDR0 [MAR0]
2   LD    Reg0  MDR0             b[i,k]
3   LD    MDR1 [MAR1]
4   LD    Reg1  MDR1             c[k,j]
5   MUL   Reg2  Reg0    Reg1     b[i,k] * c[k,j]
6   ADD   Reg3  Reg2    Reg3     partial sum
7   ADD   MAR0  Reg4    MAR0     incr. addr b[i,k]
8   ADD   MAR1  Reg5    MAR1     incr. addr c[k,j]
```

Figure 5. Matrix multiply three-address code.

المنارة للاستشارات
www.manaraa.com

Schedule for S=1				
time	*nodes*			
0	1	3	7	8
1	2	4		
2	5			
3	6			

Throughput analysis:

Nonpipelined schedule

L schedule length (cycles) 4

T throughput: 1 iteration every 4 cycles

Pipelined schedule

L schedule length (cycles) 4

S initiation interval 1

T throughput: 1 iteration per cycle

Improvement factor 4.0

Figure 6. Matrix multiply pipelined schedule.

For the inner loop, the Napa C compiler generates an ALP circuit with the nodes scheduled as shown in Fig. 6. Each node number corresponds to the corresponding line of intermediate code shown in Fig. 5. A memory access takes two stages, one to load the memory address register (MAR) with the memory address, and one to load or store the memory data register (MDR). In the first stage, the addresses for b[i,k] and c[k,j] are loaded into their respective MAR's. Nodes 7 and 8 are also executed, to increment the addresses of $b[i, k]$ and $c[k, j]$. Notice the concurrent reading and writing of registers. For example, node 1 reads MAR0 at the beginning of the stage, to get the memory address, and node 7 writes it at the end of the stage, after incrementing.

In the second stage the data are loaded from memory modules 0 and 1, respectively. The multiplication occurs in the third stage. The fourth stage has an accumulation operator that adds its input to the partial sum it maintains.

The throughput analysis summarizes the results of pipeline scheduling. Pipelining found a schedule with an initiation interval of one. The throughput of the pipelined version is one iteration every cycle, an improvement factor of four over the non-pipelined version.

5.2.2. Pipelined Walsh-Hadamard Transform. The Walsh-Hadamard transform, widely used in signal processing applications, presents an example in which the pipelined schedule is different from a nonpipelined schedule.

Inner loop of discrete Walsh-Hadamard transform

$$Z(k+1)[j] = \begin{cases} Z(k)[j] + Z(k)[j \oplus 2^k] & \text{if the } k\text{-th bit of } j \text{ is } 0 \\ Z(k)[j] - Z(k)[j \oplus 2^k] & \text{if the } k\text{-th bit of } j \text{ is } 1 \end{cases}$$

where $\oplus$ is logical operator "exclusive OR".

Figure 7. Hadamard transform, inner-loop equation.

Figure 7 shows the equation calculated in the inner loop. An outer loop indexed by k generates a new vector $Z(k+1)$ from the old vector $Z(k)$ by executing N iterations of the inner loop on j. To complete the transform, the k-loop is executed $log(N)$ times.

To eliminate control flow in the inner loop, we rewrite the code to calculate

$$Z'[j] = x \times A1 + y \times A2$$

where $A1$ and $A2$ are the two possible values shown in Fig. 7, and x and y are truth values that are 1 to select the term and 0 to ignore it.

Figure 8 shows the three-address code for this linear form of the inner loop of the code. (We note that this figure shows code that has been improved slightly by traditional compiler optimizations that will be incorporated into Napa C in the future.)

The code reads Z(k) from Memory 0 and writes Z(k + 1) in Memory 1. Successive iterations of the outer loop alternate in copying from one memory to the other. Notice that this code is an excellent candidate for dynamic partial reconfiguration, to reverse the addresses of input and output memory banks. In the absence of dynamic reconfiguration, additional control logic would need to get inserted to select between read or write access to the two sets of MARs and MDRs.

The compiler generates the schedule shown in Fig. 9. With an initiation interval of 2, the pipelined schedule delivers a throughput five times that of the nonpipelined schedule.

Our compiler presently generates pipelines for definite iteration (**for**) parallel loops. We are currently augmenting the dependency analysis phase to mark indefinite iteration (**while**) loops with do-across dependence as pipelineable.

5.3. Datapath Synthesis

The compiler front end phases are responsible for identifying and extracting ALP operations from the

www.manaraa.com

```
                 Inner loop of discrete Walsh-Hadamard transform
                    Calculate vector Z(k+1) for k+1-st stage

         Inner loop iteration count is N, where N is the length of the vector.
                              Gets repeated log N times

    Assume  Reg15 gives base address
            of Z(k) in Memory 0 and Z(k+1) in Memory 1
            Reg14 gives address of Z(k)[j]
            Reg13 gives offset of Z(k)[j ⊕ 2^k]
            Reg12 gives 2**k
            Reg11 gives j

1   LD    MAR0    Reg14
2   LD    MDR0    [MAR0]
3   LD    Reg6    MDR0            Z(k)[j]
4   XOR   Reg10   Reg11   Reg12   j ⊕ 2^k
5   MUL   Reg13   Reg10   sizej   offset of ( Z(k)[j ⊕ 2^k])
6   ADD   MAR0    Reg15   Reg13   addr( Z(k)[j ⊕ 2^k])
7   LD    MDR0    [MAR0]
8   LD    Reg5    MDR0            Z(k)[j ⊕ 2^k]
9   AND   Reg9    Reg11   Reg12   y = j ⊕ 2^k k -th bit of j
10  DIV   Reg8    Reg9    Reg12   y is 0 if k -th bit of j is 0, else 1
11  XOR   Reg7    Reg8    one     x is 1 if y is 0, else 1
12  ADD   Reg4    Reg5    Reg6    A1 = Z(k)[j ⊕ 2^k] + Z(k)[j]
13  SUB   Reg3    Reg5    Reg6    A2 = Z(k)[j ⊕ 2^k] − Z(k)[j]
14  MUL   Reg2    Reg7    Reg4    x * A1
15  MUL   Reg1    Reg8    Reg3    y * A2
16  ADD   Reg0    Reg1    Reg2    x * A1 + y * A2
17  LD    MAR1    Reg14           addr of Z(k+1)[j]
18  LD    MDR1    Reg0            Z(k+1)[j]
19  ST    [MAR1]  MDR1            store
20  ADD   Reg11   Reg11   incrJ   increment j
21  ADD   Reg14   Reg14   sizej   increment address in Z(k) to j+1-st
```

Figure 8. Hadamard transform, three-address code.

intermediate code. The individual operations are placed in one of two sorts of blocks, *pipeline* blocks, emitted for those loops that the compiler was able to pipeline, and *standard blocks*. The latter correspond to basic blocks from the source C program extended, if possible, by eliminating if-then-else control flow.

5.3.1. Overview. The purpose of datapath synthesis is

1. to map a block's arbitrary bit-length operations onto hardware function units,
2. schedule the clock cycle(s) during which the operation is active, and
3. generate control logic to arbitrate the use of the hardware function units.

The MARGE backend is responsible for synthesizing these hardware circuits. Each standard block is mapped into a single custom "instruction" which contains all the operations inherent in the block. Each pipeline block, created by the analysis and pipeline phase, is mapped to a hardware pipeline instruction customized to the loop computation in the block. The hardware function pipeline is controlled by a customized pipeline controller, also generated by MARGE. The structure of the MARGE-generated datapath is illustrated in Fig. 10. The figure shows an ALP program containing five instructions. Instructions 1–4 are standard blocks, containing the operations of four different extended basic blocks in the C program. Instruction 5 is a pipeline block, synthesized from a loop in the C

المنارة للاستشارات
www.manaraa.com

Schedule for S=2						
time	*nodes*					
0	1	4	9	17	20	21
1	2	5	10			
2	3	11				
3	6					
4	7					
5	8					
6	12	13				
7	14	15				
8	16					
9	18					
10	19					

Throughput analysis:

Nonpipelined schedule
L schedule length (cycles) 10
T throughput: 1 iteration every 10 cycles

Pipelined schedule
L schedule length (cycles) 11
S initiation interval 2
T throughput: 1 iteration every 2 cycles

Improvement factor 5.0

Figure 9. Walsh-Hadamard pipelined schedule.

program. During execution of the FIP program (also generated by the NAPA C compiler), a specific instruction number is sent to the ALP. The instruction is decoded, activating the appropriate block of logic. If the instruction is a pipeline instruction, the pipeline controller is activated, and controls the pipeline stages of the instruction.

An instruction contains multiple operations, some of which execute concurrently and others sequentially. Thus an instruction may span multiple clock cycles. In each cycle, a set of operations is performed. Each operation is assigned an instruction number and a "tick" or cycle number. Only when it is the right instruction number and the right cycle within the instruction will that operation be active. In our model, each operation completes in one clock cycle.

An instruction representing a standard block is scheduled into one or more stages determined by the data dependencies in the block. Exactly one stage of a standard block is active at one time. There may be multiple operations active in a stage at one time, assuming data dependencies permit. As shown in Section 5.2, the operations in a loop are scheduled into a number of pipeline stages. The number of stages concurrently active is determined by the initiation interval. The number of concurrent operations within a stage is determined by data dependencies and resource constraints.

During synthesis, an individual operation within a stage is mapped to a function unit appropriate to the operation, eg. adder or comparator. The function unit is selected from a library of module generators provided by National Semiconductor.

5.3.2. Module Generators. The module generation system Modgen was developed by Charle' Rupp

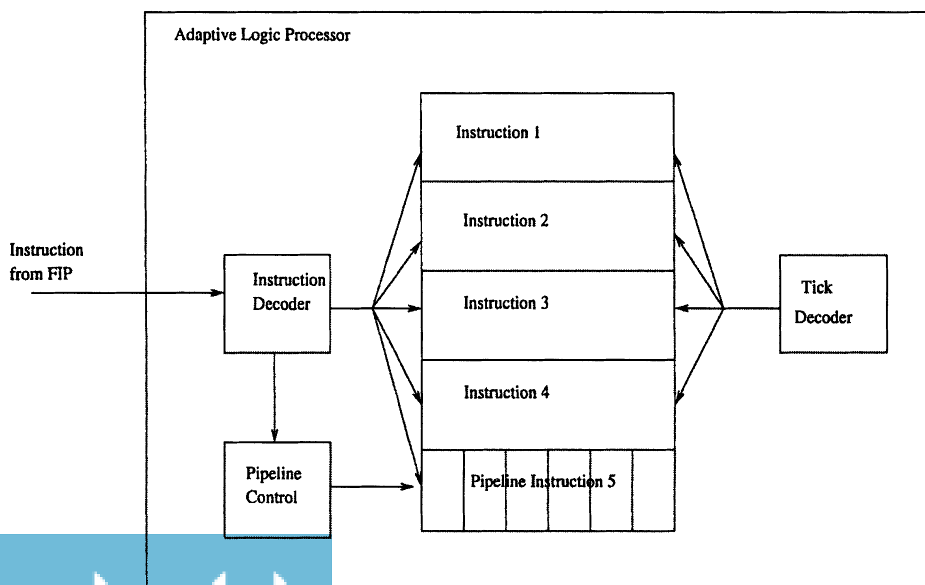


Figure 10. Block diagram of MARGE-generated ALP program.

www.manaraa.com

Arithmetic	Control	Signal Processing	Memory
ALUs	Comparators	FIR Filters	Register Banks
Absolute Value	Encoders	Lin. Seq. Gen.	ROM
Adders	Decoders	CRC detect/gen	EEROM
Decrementers	Hi/Lo/True/Comp blocks	Gray Code converters	SRAM
Incrementers	Mask generators		
Counters	Multiplexors		
Shift Operations	Tristate Buses		
Boolean Funcs.			
Multipliers			
Rotators			

Figure 11. Module generator library.

for National Semiconductor. The module generation system enables a designer to build a library of parameterized generators that allow the creation of a specific function with any number of bits (subject to the size limitations of the chip) with trade-offs between speed and area.

The library of generators encapsulate construction algorithms specific to the architecture which reflect design and layout complexity captured from an expert designer for the technology, thereby maximizing functional density and performance. The actual generators exist in a C-like language called D4 [20] and are easily written and tested once a specific function has been identified and prototyped. At the present time, the library of generators contain most of the commonly used compute functions used in datapath design.

5.4. Module Library

Figure 11 shows a partial list of generators available in the library [21].

Specifically, for the CLAy/ALP technology, Reed-Muller logic [22] has been highly leveraged for most of the compute-oriented functions as it offers an efficient alternative given the structure of the core cells in the array (XOR-based). Given the combination of this technique along with the expert place and route techniques imparted upon the generators, the macro functions result in extremely dense and performance-intensive manifestations for the technology.

As an example of quantitative comparison between functions generated using the module generators and functions generated using a combination of synthesis and automatic place and route tools, refer to Fig. 12. The figure compares area and delay for several representative functions. Area is expressed in cells. Delay has been normalized to the older CLaY cell architecture, with the Modgen delay for each function as the unit delay. The figure shows that the synthesis designs occupy between 9–15 times as many cells as the equivalent Modgen designs. The synthesis designs have 2.3–8.6 times the delay of the Modgen designs.

Function	Synth./APR	Modgen
8-bit Add	144 cells/2.75	16 cells/.5
14-bit ADD	420 cells/4.3	28 cells/.5
8x8 Parallel Mult.	2600 cells/1.6	255 cells/.7
9x11 Parallel Mult.	2800 cells/1.8	408 cells/.7

Figure 12. Area/delay comparison.

The parameterized generators not only eliminate the need to maintain large predefined macro libraries, but more significantly accommodate the generation of functions with arbitrary bit length. Rather than using functions divisible by 4-bits (as offered in most coarse-grain architecture systems) and wasting the unused bit logic, functions of specific bit-lengths can be invoked. This results from the bit slice techniques used in the generators. Given a large number of functions using non-standard bit sizes, significant area can be saved.

Additional flexibilities exist in the generators to allow the designer to make area and speed tradeoffs (algorithm selection) as well as physical options to ease integration with other functions. For example, different multiplier algorithms allow speed area tradeoffs (e.g. Ripple Carry v.s. Carry Save). Examples of physical options are output spacing and control line placement. Output spacing (referred to as the "pitch" parameter) is important when abutting to other modules during final integration. If a module is driving another module with input pins residing on a pitch of 2, selecting a pitch 2 output spacing allows perfect abuttment with no routing overhead (thereby minimizing area and delay). Strategic physical placement of control lines

المنارة للاستشارات
www.manaraa.com

```
! 2-1 mux bank
processor Q.width = _gen_MUXBNK(.width,.pitch,D0_.width,D1_.width,S)
{ int i;
  for (i = 0; i < width; i=i+1)
  { AT 0,(width-i-1)*pitch; Q.i = PMUX.1(D0_.i,D1_.i,S);
  }
}
}
! Tristate buffer bank
! _gen_TRIB
processor bus Q = _gen_TRIBNK(.width,.pitch,A.width,OE)
{ int i;
  for (i = 0; i < width; i=i+1)
  { AT 0,(width-i-1)*pitch; @,Q = PBUFZ.1(A.i,OE);
  }
}
```

Figure 13. Multiplexor and tristate generators.

also serves to minimize routing during the integration stage.

We have also gained efficiency by using generators for multiplexing structures (for control). Multiplexing three or less signals is generally achieved using 2 : 1 multiplex combinations. Multiplexing more than three signals becomes more efficient using a tristate structure, which packs quite nicely in the CLAy array compared to other technologies.

These functions are represented by very simple generators shown in Fig. 13 (2 : 1 mux bank and tristate buffer bank). We have also employed more complex tristate-multiplex structures, for example, 2-dimensional ($n \times m$) tristate banks.

The module generator library is used by MARGE. In addition, these module generators can be used manually to develop designs.

5.4.1. Automatic Module Expansion. In order to be used in a design, each module generator must be expanded first and then instantiated into the generated design. To make the module generator system work automatically with MARGE, we have developed a tool that invokes a generator whenever a generator call is detected in the netlist. The tool scans the design netlist to glean all module generator instances. A "variants" file is then constructed which contains the list of all macros to be generated along with parameter values extracted from the instantiation. The variants file is then passed to the module generation engine to automatically generate all necessary macros along with any design representations required. The generated macros are essentially treated as a design library used in subsequent integration, verification and bitstreaming steps.

An example of a generator instantiation using Verilog syntax appears below:

```
wire [7:0] sum,w1,w2;
.
// add two 8-bit words, put result in sum
add2_9 i15 (sum,cr,w1,w2);
```

The part `i15` is an `add2_9` component. By convention, this reference consists of two parts, first the name of the generator, in this case add2, followed by the bit length of the operands, in this case 9. The other parameters to the add2 macro generator include the name of the result, sum, the name of the carry, cr, and the names of the two inputs, w1 and w2.

The compiler computes bit lengths of operands and result, and constructs the instantiation name based on the combination of operation to be performed and desired bit length.

5.4.2. Details of Datapath Synthesis. The basic elements used by our model are registers, functional units, routers, and control logic.

- **Registers:** Arbitrary bit length registers mapped to configurable logic cells serve as the basic data storage element. Each variable annotated with a "reg" pragma is assigned a register. Additional registers may be allocated for intermediate storage of temporary values as needed. The Register Bank Generators are invoked for register elements.

المنارة للاستشارات
www.manaraa.com

- **Functional Units** (operators)**:** Functional units are blocks that perform basic operations on data. Some examples of arithmetic and logical operators are adders, subtractors, comparators, bitwise ANDs and bitwise ORs. The Arithmetic Generators are used for functional unit elements. We assume that each function unit takes a single clock cycle to execute.
- **Routing Control:** Data must be routed from registers to functional units to undergo some operation. In most cases these operations will result in some output value that must be routed back to a register for storage. In some simple cases the source of data may simply be hard-wired to the destination. The basic components used for more complicated routing are multiplexors and tri-state busses. Multiplexors are used when a destination has different data sources on different instructions: it will select between the possible sources during the appropriate instructions. When the number of input sources becomes sufficiently large it may be more efficient to replace the multiplexors with banks of tri-state gates. With the CLAy/ALP technology, tri-states become the more efficient option for three or more input alternatives. Control Generators are used for routing control.
- **Control Logic:** Basic logic gates (ANDs, ORs and NOTs) must be used to generate control signals to enable registers to latch data only on certain cycles of instructions. Similar control signals are needed to control routing circuitry to switch between different multiplexor inputs at different times.

The above components are controlled by signals from the instruction decoder and tick decoder.

By way of example consider the following four operation sequence:[1]

```
1) A=B+C; (operation 6, tick 2)
2) A=B+D; (operation 4, tick 0)
3) A=D-C; (operation 7, tick 3)
4) A=A+B; (operation 3, tick 1)
```

Since there are four variables in the program, four registers are needed (see Fig. 14.) There are two operators used in the program: a subtractor is needed for instruction 3, and an adder is needed for instructions 1, 2 and 4. The adder input is taken from four different sources on different instructions. The operand B is used in all addition operations, so the first input of the adder may be hardwired to the output of register B. However, the second input to the adder requires

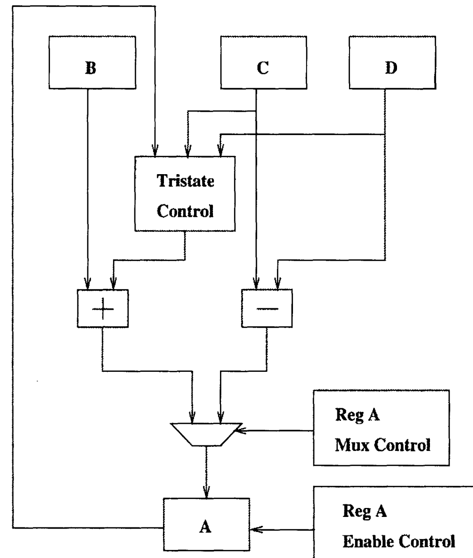


Figure 14. Structural hardware description from RTL methodology.

different sources on different instructions. The second input to the adder must be fed from the output of a tri-state router whose inputs are hardwired to be the outputs of registers A, C and D. The subtractor inputs are hard wired to be taken from the outputs of registers D and C. Since the source of the data being written to register A varies, a two input multiplexor is needed to route either the output of the adder or subtractor to its input.

The combination of signals from the instruction decoder and the multitick decoder creates a unique control signal that can uniquely identify every clock cycle of every instruction. These signals can be used to control multiplexor input selectors and to assert register enable inputs. Figure 15 illustrates the use of these signals in the context of the above example.

Since Register A is the destination of data on all four instructions, an enable signal must be generated to allow it to latch new data on all four instructions on the appropriate multitick cycles. The appropriate signals must also be generated to select the proper inputs of the multiplexor feeding register A. Likewise, the tristate enable signal must be created so that register C is routed to the adder on the first instruction, register D is routed to the adder on the second instruction, and register A is routed to the adder on the fourth instruction, all on the appropriate multitick cycles.

The example above maps to a relatively simple hardware description composed of basic functional building blocks. These building blocks are obtained from

www.manaraa.com

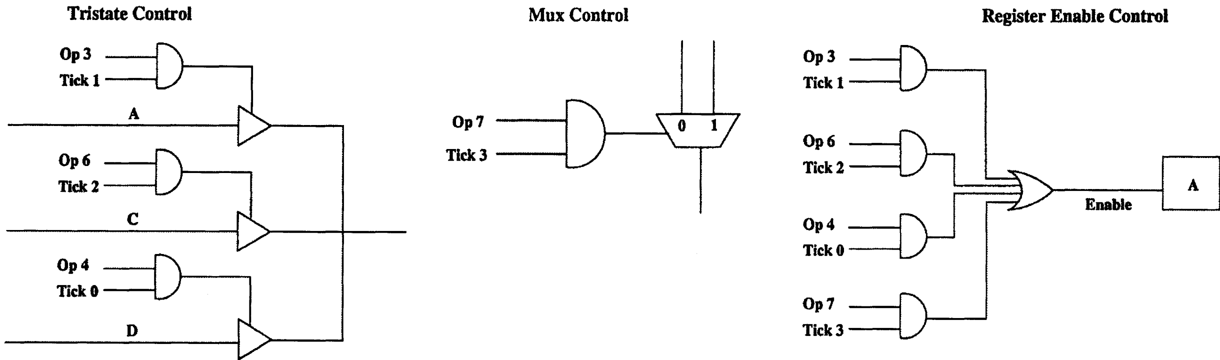


Figure 15. Data routing and control structures.

the Modgen module library, so that the operations are expressed in terms of optimized macros that have been generated on the fly.

The example above illustrates the basic methodology. MARGE also performs various optimizations:

- **Function unit reuse:** Function units are reused between operations whenever possible. It is possible to reuse a function unit if it is of the appropriate size and functionality and is not being used in a different operation on the clock cycle currently being scheduled. This optimization is controlled by a compiler switch since it is sometimes more efficient from the routing point of view to duplicate function units.
- **Commutativity optimization:** Operands to a binary function may be swapped to reduce mux/tristate control logic. In the fourth instruction of the above example, MARGE swaps the operands of the commutative add operation so that the left input to the adder can be hardwired to B. Without the transformation, a mux would have been generated for the left input.

The MARGE synthesis module can also handle bit insertion and extraction operations, in which a portion of a register is stored or a range of bits from a register is extracted. These operations can be expressed via instrinsic function calls and map directly into wires (plus mux and tristate control if required) in the hardware.

We have found that invoking the module generators through MARGE delivers significant performance improvement over conventional synthesis.

Figure 16 shows comparative results from two programs XCorr, a bit stream cross correlation program, and DNA, a DNA sequence match program. Both programs use systolic algorithms. XCorr performs bit-oriented operations and sums into a 16-bit counter. DNA uses operands of 2- and 4-bits, where the 2-bit operands are used in compares, and the 4-bit operands are used in adds. The gate-level structural code emitted by the compiler was processed by two different sets of tools. The column labelled Synthesis/APR shows the result of processing the structural code with the Synopsys Design Compiler and then using National Semiconductor tools to generate the bit streams. The column labelled Modgen show the result of mapping MARGE's structural output to pre-placed, pre-routed macros from the Module Library. For each application, the table shows both the number of cells used and the number of nets required by the alternative compilation methods. As the table shows, there is a dramatic reduction in the number of core cells used by the Modgen versions over the synthesis versions, with the synthesis versions taking up to 3.4 times as many cells. In addition, the number of nets, an indication of routing resources consumed, are also reduced in the Modgen versions. The synthesis versions require up to 1.9 times as many nets.

Program	Synthesis/APR	Modgen
XCorr	1444 cells	428 cells
DNA	2703 cells	1350 cells
XCorr	153 nets	81 nets
DNA	403 nets	221 nets

Figure 16. Synthesis vs. modgen on two applications.

We have obtained efficiency over traditional synthesis by targeting a library of pre-placed, pre-routed macro generators for a fine grained FPGA architecture. The macro generators have been designed by expert engineers to be arbitrarily expandable in bit width

المنارة للاستشارات
www.manaraa.com

and to fit well with each other. Our techniques show significant area and delay improvements for Modgen elements over equivalent functions compiled through traditional synthesis and APR.

6. Conclusions and Future Work

We have described a language and compiler that target a novel hybrid RISC/FPGA architecture. Contributions of our work include

- defining a methodology to explore performance trade-offs in mapping computation between FIP and ALP
- developing a small set of directives to allow programmer or automatic system to specify mapping between FIP and ALP
- modular construction of the various phases of the compiler, which allows us (and others) easily to insert new capabilities into the system
- synthesis of a new sort of executable image that combines conventional object code with configuration bit streams
- automatic analysis of loops for pipelining, and synthesis of customized hardware pipelines
- use of pre-placed, pre-routed macro generators to improve the quality of synthesized hardware and to reduce compile times.

We are working on several longer term projects. These include

- programming language and compiler support for parallel processing FPGA arrays with concurrent computation on multiple nodes [23],
- compiler interaction with an interactive performance analysis system [24] so that pragmas can be inserted automatically into the C source code, and feedback from lower-level CAD tools is available to compiler and performance analysis system.

Acknowledgments

This work was supported by National Semiconductor Corp., DARPA through Contract DAB63-94-C-0085 to NSC and the Sarnoff Corporation. Members of the NAPA1000 team include Jeff Arnold, Stephen Bade, Shao-Pin Chen, Tim Garverick, Maya Gokhale, Edson Gomersall, Harry Holt, Jeff Hutchins, Jim Kaba, Mark Landguth, Aaron Marks, Dr. Charle' Rupp, and Janice Stone.

Note

1. Note that this is not a semantically meaningful sequence of operations. A more realistic scenario is that the four instructions are separated by other instructions, possibly including control flow. The example is simplified for clarity.

References

1. R. Razdan and M.D. Smith, "A High-Performance Microarchitecture with Hardware-Programmable Functional Units," *Proceedings of the 27th Annual International Symposium on Microarchitecture*, IEEE/ACM, Nov. 1994, pp. 172–180.
2. R. Wittig and P. Chow, "Onechip: An FPGA Processor with Reconfigurable Logic," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, J.M. Arnold and K.L. Pocek (Eds.), Napa, CA, Apr. 1996.
3. J.R. Hauser and J. Wawrzynek, "GARP: A MIPS Processor with a Reconfigurable Coprocessor," *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, 1997, J. Arnold and K.L. Pocek (Eds.), Napa, CA, Apr., pp. 12–21.
4. S. Hauck, T. Fry, M. Hosler, and J. Kao, "The Chimera Reconfigurable Functional Unit," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, J.M. Arnold and K.L. Pocek (Eds.), Napa, CA, Apr. 1997, pp. 87–97.
5. C. Rupp, M. Landguth, T. Garverick, E. Gomersall, H. Holt, J. Arnold, and M. Gokhale, "The Napa Adaptive Processing Architecture," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, J.M. Arnold and K.L. Pocek (Eds.), Napa, CA, Apr. 1998.
6. M. Rencher and B. Hutchins, "Automatic Target Recognition on Splash 2," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, J.M. Arnold and K.L. Pocek (Eds.), Napa, CA, Apr. 1997.
7. SUIF Group. Suif Compiler System. *http://suif.stanford.edu*, 1997.
8. M. Gokhale and J.M. Stone, "Automatic Allocation of Arrays to Memories in FPGA Processors with Multiple Memory Banks," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, J.M. Arnold and K.L. Pocek (Eds.), Napa, CA, Apr. 1999.
9. M. Gokhale and J. Stone, "Napa C: Compiling for a Hybrid Risc/FPGA Architecture," *IEEE Symposium on FPGAs for Custom Computing Machines*, K. Pocek and J. Arnold (Eds.), Napa Valley, CA, Apr. 1998, pp. 126–135.
10. M. Gokhale and E. Gomersall, "High Level Compilation for Fine Grained FPGAS," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, J.M. Arnold and K.L. Pocek (Eds.), Napa, CA, Apr. 1997.
11. E. Waingold, M. Taylor, P. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it All to Software: Raw Machines," *IEEE Computer*, 1997, pp. 86–93.
12. T. Callahan and J. Wawrzynek, "Instruction Level Parallelism for Reconfigurable Computing," *Proceedings of FPL '98, Field-Programmable Logic and Applications, 8th International Workshop*, Estonia, Sept. 1998. Published in Springer-Verlag Lecture Notes in Computer, Hartenstein and Keevallik (Eds.).
13. R. Harr, Nimble Compiler. *http://www.arpa.mil/ito/psum1998/*, 1998.

المنارة للاستشارات
www.manaraa.com

14. I. Page, "Construction of Hardware-Software Systems from a Single Description," *Jounal of VLSI Signal Processing*, vol. 12, 1996, pp. 87–107.
15. M. Weinhardt, "Portable Pipeline Synthesis for FCCMS," Technical Report, Universitat Karlsruhe, 1996.
16. M. Weinhardt and W. Luk, "Pipeline Vectorization for Reconfigurable Systems," *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, J.M. Arnold and K.L. Pocek (Eds.), Napa, CA, Apr. 1999.
17. T. Garverick, C. Rupp, and J. Arnold, Napa 1000. *http://www.national.com/appinfo/milaero/napa1000/*, 1997.
18. High Performance Fortran. Hpf Forum. *http://www.crpc.rice.edu/HPFF/home.html*, 1999.
19. M. Lam, "Software Pipelining: An Effective Scheduling Technique for VLIW Machines," *SIG-PLAN '88*, June 1988.
20. C.R. Rupp, *D4 Language Reference Guide and Specification*, 1996.
21. C.R. Rupp, *D4 User's Guide*, 1996.
22. D. Green, *Modern Logic Design*, Addison-Wesley, 1986.
23. M. Gokhale, Compiling for FPGA-Based Parallel Processors. *http://www.sarnoff.com:8000/acshome/fpgahome.html*, 1997.
24. M. Martinosi, Performance and Synthesis Tools for Adaptive Computing. *//http://www.ee.princeton.edu/mrm/resconf.html*, 1997.

Maya B. Gokhale received the Ph.D. degree in Computer and Information Science from the University of Pennsylvania in 1983. Dr. Gokhale has been involved in tools and architectures for FPGA-based configurable computing for the last eleven years. She was part of the Splash team at the Supercomputing Research Center and developed a data parallel C compiler that targeted the Splash-2 FPGA array. At Sarnoff Corporation, Gokhale extended the FPGA compiling technology to target hybrid processor/FPGA chips. She is currently Principal Investigator of a DARPA-sponsored reconfigurable computing effort at the Los Alamos National Laboratory.
maya@sarnoff.com

Janice M. Stone received the A.B. degree in mathematics from Duke University in 1962, and pursued graduate studies in mathematics at Georgetown University, and in logic and philosophy of science at Stanford University. She joined IBM Research in 1984, where her research interests focused on parallel algorithms and tools for development and analysis of parallel programs. She is now an independent contractor in Princeton, NJ, where her recent work combines compiler construction and pipeline scheduling for configurable computing.

Edson Gomersall has 14 years of experience in the areas of device modeling, circuit design, design methodology, software engineering, and reconfigurable computing. He has been granted two circuit design patents and published several technical papers. Mr. Gomersall is currently involved in design methodology relating to the implementation of large systems on a chip at National Semiconductor Corp..
edson@nsc.com

المنارة للاستشارات
www.manaraa.c

Journal of VLSI Signal Processing Systems 24, 181–209, 2000.
© 2000 Kluwer Academic Publishers. Printed in The Netherlands.

Design-Space Exploration for Block-Processing Based Temporal Partitioning of Run-Time Reconfigurable Systems

MEENAKSHI KAUL AND RANGA VEMURI
Laboratory for Digital Design Environments, Department of ECECS, P.O. Box 210030, University of Cincinnati, Cincinnati, OH 45221-0030, USA

Abstract. The reconfiguration capability of modern FPGA devices can be utilized to execute an application by partitioning it into multiple segments such that each segment is executed one after the other on the device. This division of an application into multiple reconfigurable segments is called *temporal partitioning*. We present an automated temporal partitioning technique for acyclic behavior level task graphs. To be effective, any behavior-level partitioning method should ensure that each temporal partition meets the underlying resource constraints. For this, a knowledge of the implementation cost of each task on the hardware should be known. Since multiple implementations of a task that differ in area and delay are possible, we perform *design-space* exploration to choose the best implementation of a task from among the available implementations.

To overcome the high reconfiguration overhead of the current day FPGA devices, we propose integration of the temporal partitioning and design space exploration methodology with *block-processing*. Block-processing is used to process multiple blocks of data on each temporal partition so as to amortize the reconfiguration time. We focus on applications that can be represented as task graphs that have to be executed many times over a large set of input data. We have integrated block-processing in the temporal partitioning framework so that it also influences the design point selection for each task. However, this does not exclude usage of our system for designs for which block-processing is not possible. For both block-processing and non block-processing designs our algorithm selects the best possible design point to minimize the execution time of the design.

We present an ILP-based methodology for the integrated temporal partitioning, design space exploration and block-processing technique that is solved to optimality for small sized design problems and in an iterative constraint satisfaction approach for large sized design problems. We demonstrate with extensive experimental results for the Discrete Cosine Transform (DCT) and random graphs the validity of our approach.

1. Introduction

Reconfigurable Field Programmable Gate Arrays (FPGAs) [1–3] built of SRAM-based logic provide designers with *flexible* computing systems. In these devices, the state of the internal static memory cells determines the logic functions and interconnections resident within the FPGA device. This uncommitted array of programmable logic and interconnect on these devices allows reconfiguration between algorithm implementation on the devices. This advantage of FPGAs over Application Specific Integrated Circuits (ASICs) allows the user to use the same circuitry for completely different algorithms by configuring the device between applications. This design approach is generally referred to as *Compile-Time* or *Static Reconfiguration* [4]. Statically configured FPGAs have been used successfully in the rapid prototyping of designs [5, 6]. The long fabrication times associated with ASIC design is eliminated. But the device capacity of FPGAs is far less than that of ASIC chips. Therefore, when synthesizing large designs on FPGAs, usually multi-FPGA boards are used to increase device capacity. This necessitates spatial partitioning of the application. In this style of static FPGA design, the FPGA is configured once at the start of the application, and the same configuration continues till the execution ends.

المنارة للاستشارات
www.manaraa.com

However, by extending the idea of reconfiguration to intra-application reconfiguration an application which does not fit on the device is divided into multiple segments and multiple configurations of the same application are loaded at run-time. This technique is referred to as *Run-Time or Dynamic Reconfiguration* (*RTR*) [4]. Current design tools provide support for static reconfiguration, but little tool support exists for dynamic reconfiguration.

When a design is partitioned into mutually exclusive partitions that will execute serially on the reconfigurable processor, the design uses Global Run Time Reconfiguration. All modern FPGAs, whether fully (XC4000, XCV000) or partially (XC6200, XCV000) reprogrammable, can support this reconfiguration step. In partially programmable FPGAs the inactive parts of the FPGA can be reconfigured at run-time even while other parts of the FPGA are active. This flexibility of partial reconfiguration can be exploited in a design approach where subsets of the application are reconfigured as the application executes. This can reduce the time to reconfigure the FPGAs by making it possible to load only the necessary parts of the FPGA. However this increased flexibility also introduces a lot of complexity in the CAD process needed to design applications for such a design style.

The temporal partitioning approach that we have undertaken focuses on generating *global* run time reconfigured designs from behavior specifications of the design. We perform run time reconfiguration in which the entire device is reprogrammed at the boundaries of the temporal segments and data is passed from one temporal segment to the next through a RAM which is not part of the reconfigurable logic. Due to this, structural design is not necessary; behavioral synthesis can be effectively used.

A shortcoming of current automated temporal partitioning techniques is that they choose the underlying implementation of the components of their design before partitioning is performed. Since there are multiple implementations of the components of the designs that vary in the area/delay, it would be more effective to choose the design implementation while partitioning the design by exploring different design options. The search of the design space while partitioning would lead to better partitioned designs.

Due to very high reconfiguration overheads for commercially available reconfigurable hardware, existing automated temporal partitioning techniques [7–11] usually focus on reducing the latency of the temporally partitioned design by minimizing the number of temporal partitions in the design. But, many DSP applications process an infinite or semi-infinite stream of input data. We will demonstrate that the design with minimum latency *may not* be the best overall solution if we can process multiple inputs on each temporal partition. This technique, called *block-processing* can be used to reduce the the reconfiguration overhead.

If the reconfiguration overhead is ignored, the latency of a temporally partitioned design is usually less than the latency of a static design due to the larger area available. But since the reconfiguration overhead is an important factor in determining the run time of a design, an RTR system may perform poorly as compared to a static design if the reconfiguration overhead dominates the execution time of the design. To overcome the effects of high reconfiguration overhead, we demonstrate [12] how block processing can be introduced at a post-processing step after temporally partitioning a design to increase the throughput. In the current work, we develop a temporal partitioning technique to incorporate block-processing and design space exploration and demonstrate how this integrated processing can be used to search for optimal temporally partitioned designs. In this paper, an Integer Linear Programming (ILP) based integrated temporal partitioning and design space exploration technique forms a core solution method. For small sized design problems we solve the ILP model to obtain an optimal solution, and we demonstrate the effectiveness of our technique with experimental results. To handle large design problems with our technique we also present an iterative refinement procedure that iteratively explores different regions of the design space and leads to reduction in the execution time of the partitioned design. The ILP based integrated temporal partitioning and design space exploration technique forms a core solution method which is used in a constraint satisfying approach to explore different regions of the design space. Again, we demonstrate the effectiveness of this technique with experimental results.

We present the motivation of our work in Section 2, previous work in Section 3, the design flow of our tool in Section 4, the architecture model, design process model and memory model in Section 5, the ILP model, the optimal search algorithm and its results in Section 6, the iterative algorithm and its experimental results in Section 7, results on random graphs and comparison with other works in Section 8, some discussions on extensions and limitations in Section 9, and the conclusions in Section 10.

www.manaraa.com

2. Motivation

In the following discussion we present the problem of task level design space exploration in temporal partitioning and how its integration with block-processing techniques can improve the execution time of an RTR design.

Input Specification as Task Graphs: Growing design complexity has lead designers to generate designs at higher levels of abstraction, such as the behavior level. The designer can concentrate on the required behavior of the application, rather than its implementation. Also simulation at behavior level is much faster than Register Transfer level (RTL) or gate level simulation. In this paper, we concentrate on behavior level design descriptions to be temporally partitioned. We assume the input specification to be a task graph, where each task consists of a set of operations. Task boundaries can be given by the designer or, tasks can be automatically derived from the behavior specification by clustering or template extraction techniques [13]. Our approach can handle tasks of any level of granularity.

Design Alternatives for Tasks: Depending on the resource/area constraint for the design, different implementations of the same task which represent different area-time tradeoff points can be contemplated. These different implementations are *design points/Pareto points* [14] in the design space of a task. In Fig. 1, a task and two different implementations of the task are represented. Design Point 1 uses two adders and four multipliers, and is scheduled in two control steps. Design point 2, on the other hand uses less resources and more control steps. If a task is implemented with less resources then the operations in the task will be executed serially, thus increasing the latency of the task. On the other hand, an implementation with more resources reduces the latency but increases the area. Choosing the best design point for each task may not necessarily result in the best overall design for the specification. The most optimal point for a task in the context of optimizing the overall throughput of the design will depend on the architectural constraints of the reconfigurable hardware and the dependency constraints among the tasks. In the subsequent discussion we will express the latency of a design point in terms of total execution time and not in number of clock cycles.

If the number of design alternatives for a task are too many, then exploring the large design space can become too computationally expensive. In such cases, a few 'candidate' design points must be obtained by effective design space pruning techniques, such as discussed in [13]. Since there is a gap between the behavior description and the final synthesized design, it is important that we have accurate synthesis estimates for the tasks. As the size of a task in the task graph is quite small, we use sophisticated High-Level Synthesis estimators which incorporate layout estimation techniques. Such partitioned designs, can then be predictably taken down to the actual FPGA layout [15, 16].

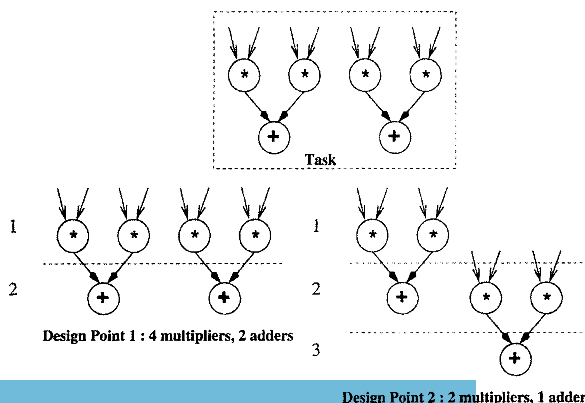


Figure 1. Multiple design points for a task.

Block-Processing in Temporally Partitioned Designs: In many application domains e.g., Digital Signal Processing, computations are defined on very long streams of input data. In such applications an approach known as *block-processing* is used to increase the throughput of a system through the use of parallelism and pipelining in the area of parallel compilers [17] and VLSI processors [18]. Block-processing is not only beneficial in parallelizing/pipelining of applications, but in all cases where the net cost of processing k samples of data individually is higher than the net cost of processing k samples simultaneously. We can also apply the concept of block-processing to a single processor reconfigurable system to speedup the processing time.

Fig. 2 illustrates the use of block-processing to speed up computation in a temporally partitioned design. The task graph consists of 4 tasks A, B, C, D. It is partitioned into two temporal partitions as shown in Fig. 2(b). The latency of temporal partition 1 is 50 ns and of partition 2 is 80 ns. The reconfiguration time is 500 ns. The latency of the design is

www.manaraa.com
الاستشارات المنارة

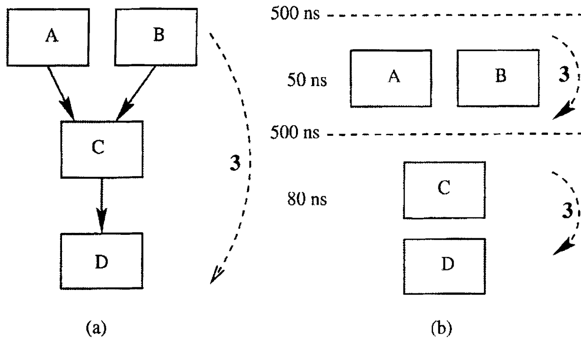


Figure 2. Temporally partitioned design example.

$50 + 500 + 80 + 500 = 1130$ ns. A single iteration of the task graph executes in 1130 ns. Now three iterations of this temporally partitioned design will take $3 \times 1130 = 3390$ ns. However if we perform block-processing by sequencing all 3 computations on each temporal partition, the time taken for the execution is $(50 + 50 + 50) + 500 + (80 + 80 + 80) + 500 = 1390$ ns. Thus block-processing amortizes the reconfiguration overhead over the 3 computations. Block-processing is possible only for applications that process a large stream of inputs. We represent such applications by a task graph having an implicit outer loop as shown in Fig. 2(a). Note that block-processing is possible if there are no dependencies among the computations for different inputs. In compiler terminology this means there should be no loop-carried dependencies due to the implicit outer loop, among different iterations of this loop. In this paper, we deal with applications for which no dependencies among computations is present. Most DSP applications such as Image processing, Template Matching, Encryption algorithms etc. fall in this category. The examples investigated in the RC community include DCT, FFT, DFT, FIR filter and various image averaging, smoothing and filtering algorithms. Also many matrix based computations eg. LU Decomposition for solving linear equations, polynomial interpolation, extrapolation etc. are acyclic in nature.

Integrating Design-Space Exploration and Block-Processing in Temporal Partitioning: For FPGA based architectural synthesis, the constraints of area of the FPGA in terms of CLBs (Configurable Logic Blocks)/FGs (Function Generators) and memory are to be met by the partitioned design. The design alternatives or solutions will vary in the number of temporal partitions and the latency of the partitioned design. For the *spatial* partitioning problem (partitioning of the design for a fixed number of co-existing FPGAs on a board), increasing the number of partitions has the effect of increasing the overall area for the design, and directly affects the latency of the design. Increasing the area, generally increases the number of operations that can execute in parallel (if no dependency constraints exist) and thus decreases the latency of the design. However, for a temporal partitioning system, increasing the number of partitions increases the area available for the design, but this increase is 'over time' and not 'over space'. This increase in number of partitions may or may not result in the reduction of the latency of the design.

When the reconfiguration overhead is very large compared to the execution time of the task it is clear that minimizing the number of temporal partitions will achieve the *smallest latency* in the overall design. In the resultant solution each task will usually be mapped to the smallest area design point among the set of design points for a task. However, it is *not necessary* that the minimum latency design is the best solution. We illustrate this idea with an example. In the Fig. 3(a) a task graph is shown. Each task has two different design points on which it can be mapped. Two different

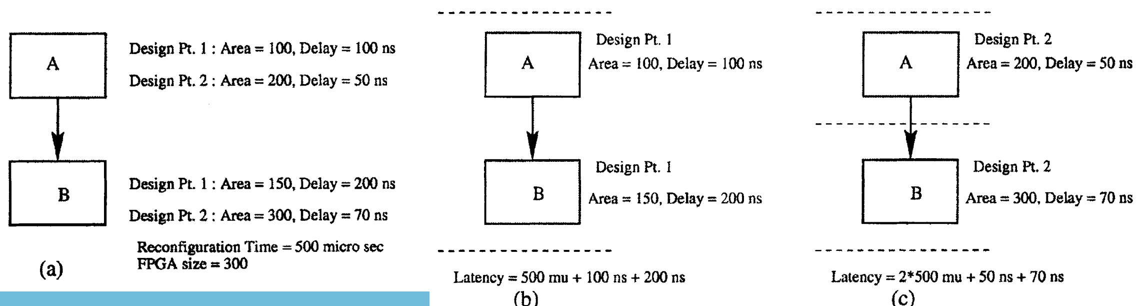


Figure 3. Design space exploration.

المنارة للاستشارات
www.manaraa.com

solutions (b) and (c) are shown. If minimum latency solution is required then solution (b) will be chosen over solution (c) because the latency of (b) is 500.3 μsec and latency of (c) is 1000.12 μsec. Now, if we use (b) and (c) in the block-processing framework to process 5000 computations on each temporal partition, then the execution time for solution (b) is 2000 μsec and for solution (c) is 1600 μsec. Therefore if we can integrate the knowledge about block-processing while design space exploration is being done, then it is possible to choose more appropriate solutions.

The price paid for block-processing is the higher memory requirements for the reconfigured design. We call the number of data samples or inputs to be processed in each temporal partition to be the the *block-processing factor*, k. This is given by the user and is the minimum number of input data computations that this design will execute for typical runs of the application. The amount of block-processing is limited by the amount of memory available to store the intermediate results.

3. Previous Work

Design for reconfigurable architectures involves temporal and spatial partitioning and synthesis [15]. There has been significant research on spatial partitioning [19–21] and synthesis [16, 22], though the research on temporal partitioning is in a nascent stage. Currently many designers perform temporal partitioning manually [23, 24] or the designer needs to specify the partitioning points of the application to the partitioning tool [25]. Luk, Shirazi and Cheung [26] take advantage of the partial reconfiguration capability of FPGAs and automate techniques of identifying and mapping reconfigurable regions from pre-temporally partitioned circuits. Chu et al. [27] present a partial evaluation technique in their circuit generator. In this technique the programmer can provide partial evaluation routines for his components. These partial evaluation routines can then be used to reduce the complexity of the component based on its inputs available at run-time. This technique also utilizes the partial programming capability of the FPGAs, however the programmer has to explicitly define the components that can be partially evaluated and also the method to do so.

Existing automated temporal partitioning techniques, extend scheduling and clustering techniques of high-level synthesis [7–9, 11] and focus on minimizing the number of partitions of the design. In [8, 9, 11] the temporal partitioning technique involves partitioning gate-level designs. Since the design to be partitioned is already synthesized, different synthesis options for achieving partitioned solutions with lower execution times cannot be explored. Since the reconfiguration overhead for currently available hardware is very large and dominates the latency of the design, we need to concentrate on techniques to minimize the effects of the reconfiguration overhead. We present an automated technique for DSP style applications, that automatically sequences multiple computations in each temporal partition to reduce the reconfiguration overhead. To our knowledge, no existing tools perform automated block-processing techniques to reduce the reconfiguration overhead in the context of reconfigurable processors. Our technique can also simultaneously handle multiple design constraints, e.g., FPGA resources, on-board memories, and perform design exploration that cannot be handled by current techniques in [7–9, 11]. Kaul and Vemuri [10] presented a mathematical model for combined temporal partitioning and synthesis. In this approach, synthesis cost exploration is performed at an operation level in the task graph, and the number of alternative solutions explored becomes very large. This approach can be used to synthesize small-scale behavior specifications. Kaul and Vemuri also demonstrated the technique of integrated temporal partitioning and design-space exploration for large design problems by using an iterative constraint satisfaction approach [28]. The design space exploration was performed without considering block-processing, so the goal of the system was to minimize the latency of the design.

Wirthlin and Hutchings [29] developed an automated technique that uses partial reconfiguration to load custom instructions at run-time. The instructions in an application are loaded in a demand-driven manner, and unused instructions are removed. This work is in contrast to our approach as it performs the loading and unloading of instructions at run time, whereas in our approach the partitioning into global configurations is performed before the design executes and not at run time.

Kalavade [30] presents an extended bi-partitioning problem for co-design, where partitioning and design point selection is performed sequentially, unlike our combined approach. ILP models of other partitioning and synthesis problems have been addressed by researchers. Simultaneous spatial partitioning and synthesis is formulated as an ILP by Gebotys in [31].

www.manaraa.com

Niemann and Marwedel [32] present an ILP-based methodology for hardware software partitioning of co-design systems. Resource constrained scheduling and binding at operation level for ASICs has been formulated as an ILP by Gebotys in [33].

Our temporal partitioning approach is for a globally configured system and we do not consider the partial reconfiguration approach for designing a RTR system. We attempt to perform temporal partitioning at a high-level together with design-space exploration. No other approach to temporal partitioning has attempted to do so. The disadvantage of our technique is that it cannot make use of the partial reconfiguration capability of the FPGAs. This would involve FPGA-specific tools as the different FPGAs have different kinds of partial reconfiguration capabilities. However, our current work is focussed on developing a general purpose tool that can be used to develop temporal partitioned systems for any class of FPGAs on which global reconfiguration can be performed. Some of the partial reconfiguration techniques [26] assume that temporal partitions already exists when they attempt to find matching circuits across temporal partitions. Our technique can be used to automatically generate the temporal partitions that can then be used by such techniques to generate partial reconfigurations.

Contribution of this work: The current work makes several important contributions to the area of reconfigurable design synthesis. It has the following primary features that distinguish it from other works:

- We have integrated the problem of design space exploration into partitioning by using the idea of considering multiple design points for each task in the task-graph. This reduces the complexity of the design space search for the high-level synthesis process by making it concentrate on small portions of the design.
- Our approach performs design space exploration at the behavior level of abstraction, so that multiple design options are explored while performing temporal partitioning and appropriate design points based on the constraints of the architecture are chosen.
- Unlike traditional approaches that concentrate on minimizing the number of temporal partitions of the design, our approach introduces a novel concept of block-processing multiple computations to reduce the reconfiguration overhead and demonstrates that a temporal partitioning approach which combines block-processing and design-space exploration can reduce the design execution time.
- By using ILP as the core engine in an iterative search process we have the flexibility to produce optimal/near-optimal partitioned designs. User controlled parameters influence the search process. If the search for an optimal solution is too time intensive, then suitable search parameters can be given to produce near-optimal results in less run-time.

4. System Design Flow

In Fig. 4, we present the design flow for building a Run-Time Reconfigured (RTR) design. The input specification is a behavior level design description of the application to be implemented on the reconfigurable hardware. The input specification is shown in Fig. 5. It consists of an acyclic data flow task graph, with an outer implicit loop. The implicit loop signifies the successive items of input data that will be executed on this task graph. There are no inter-loop dependencies in the task graph due to this implicit loop, i.e., the processing of each input data is independent of any other input.

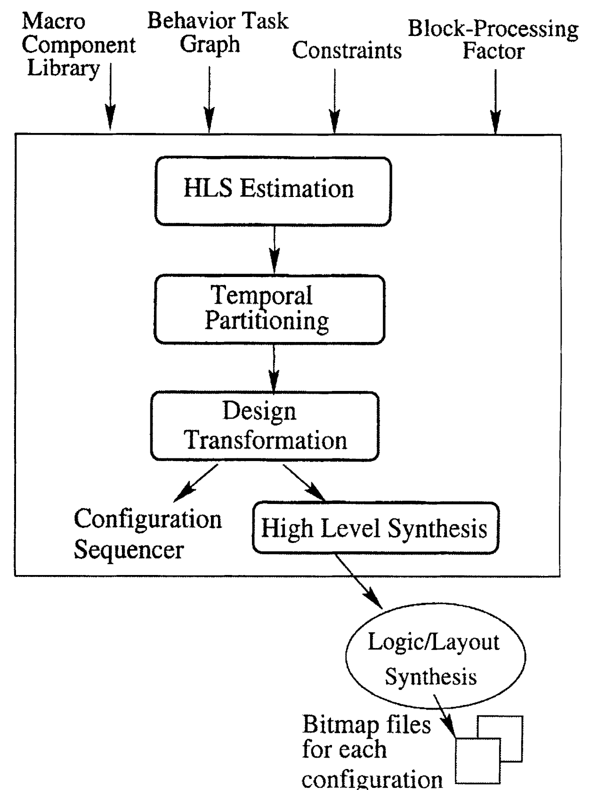


Figure 4. System design flow.

المنارة للاستشارات
www.manaraa.com

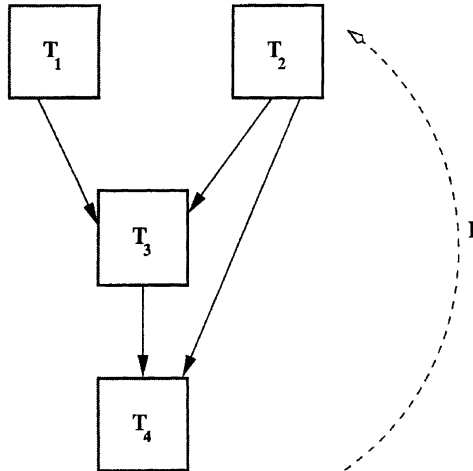


Figure 5. Behavior task graph with implicit outer loop.

Thus it is possible to perform block-processing for this task graph.

Task Estimation: First, the behavior level estimation engine, which is part of the SPARCS design environment [15], generates multiple design points for each task separately based on the architecture and user constraints. The architecture constraints are the resources available on the reconfigurable hardware, the user constraints are the maximum clock-width for the design. The HLS tool makes use of a component library, characterized for the particular reconfigurable processor, to estimate the resource and delay.

Temporal Partitioning: Next, the temporal partitioning tool divides the task graph into multiple temporal segments, while mapping each task to its appropriate design point. We discuss the ILP formulation used to solve the multi-constraint temporal partitioning problem later in detail.

Design Transformation: Some design transformations are needed so that block-processing can be performed on each temporal partition. This design transformation and the software code to sequence the configurations from the host is generated in this step.

High Level Synthesis: The high level synthesis system in SPARCS [15] is used to generate the RTL design for each temporal segment.

Logic/Layout Synthesis: We use commercial tools, for logic synthesis (Synplify tools from Synplicity) and layout synthesis (Xilinx M1 tools) to convert the RTL description of each configuration into bitmap files.

5. Architecture, Design Process and Memory Model

5.1. Reconfigurable Architecture Model

In Fig. 6, the reconfigurable architecture on which the Run-Time reconfigured design is to be mapped is shown. It consists of a reconfigurable hardware communicating with an external memory. Each temporal partition is mapped to the reconfigurable hardware, and the data flowing between two temporal partitions is mapped to the memory. The host stores all the temporal configurations. It interacts with the reconfigurable hardware to load new configurations and with the memory to load input data and retrieve output data at the end of the execution of the design. Except for the first and last temporal configuration it does not read or write to the memory at any other intermediate configuration.

5.2. Design Process Model

- Each behavior specification is in the form of a acyclic task graph. A task however has no restrictions and can contain any control structure within it. Each task is indivisible and parts of a task cannot be mapped across partitions.
- Each task has a set of distinct implementation options called design points. These are usually obtained by a high level estimation tool or can be specified by the user. No possible restrictions on the implementation of a task is required, only that the area and delay associated with each design point should be available.

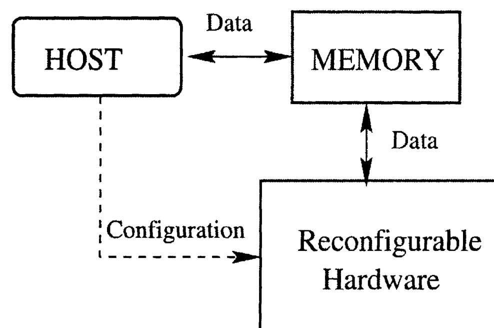


Figure 6. RTR architecture model.

المنارة للاستشارات
www.manaraa.com

- The high level synthesis process that will be used to synthesize the tasks in each temporal partition is expected to parallelize the intermediate memory transfer with the execution of operations in the task graph. We are assuming that there is enough slack available to do so. Therefore we do not add the intermediate data transfer time to the execution time of the design. If a simple synthesis system is used that does all the memory access in serial with the operations, then we need to add the memory access times in the execution time of the design. We have discussed this further in Section 9.
- The estimation process that develops the design point should be closely related to the actual synthesis process that will synthesize the temporal partition after the partitioning is performed. For our design process this implies that the area of the design point should reflect the data path, controller and routing resources required for the task. Xu and Kurdahi [16] and Ouaiss et al. [15] discuss some of the estimation techniques that incorporate low level details in the estimation process. However, if the estimation process is not a true reflection of the ultimate synthesis process then it is required that the user of our system should generate experimentally a factor that reflects the deviation of the actual values from the estimated ones. The area of the FPGA should be reduced by this factor.

5.3. Memory Model

- The host writes to the memory of the reconfigurable architecture before the start of the first configuration to place all the data that is to be read as input by the design, and reads from the last configuration's memory all the output data of the design.
- The intermediate data that is needed to be transferred from one configuration to another is written into the memory of the reconfigurable architecture.
- All data to be read in a configuration and written in a configuration is alive for the existence of the whole configuration. The input data from the host is present from the first configuration till the last configuration it is read from. The output data to the host is present from the configuration it is written till the last configuration. Data written in a configuration will remain alive in all subsequent configurations till it is consumed.
- The model for block processing is shown in Fig. 7. Each configuration processes the whole block of k computations completely and stores the intermediate data. This is repeated for all configurations. Currently, we does not support pipelining of the different computations in the same temporal partition. Therefore the delay for k computations is then equal to k*delay for processing one computation.

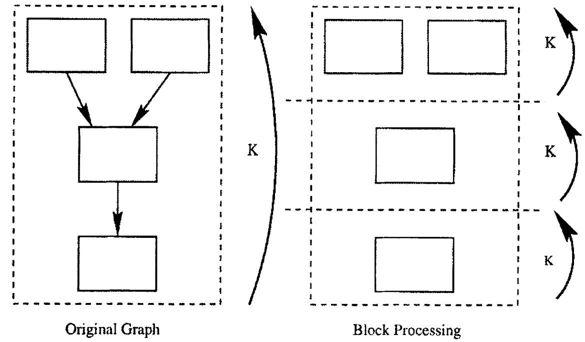


Figure 7. Block-processing model.

6. Temporal Partitioning and Design Space Exploration by an Optimal Search Algorithm

The inputs to our Temporal Partitioning system are—(1) Behavior Specifications (2) Target Architecture Parameters (3) Block-processing Factor.

In formal notation, the inputs are stated as

T	*set of tasks in the task graph.*
$t_i \rightarrow t_j$	*a directed edge between tasks,* $t_i, t_j \in T$*, exists in the task graph.*
$B(t_i, t_j)$	*number of data units to be communicated between tasks* t_i *and* t_j*.*
$B(env, t_j)$	*number of data units to be read by task* t_j *from the environment.*
$B(t_i, env)$	*number of data units to be written from task* t_i *to the environment.*
R_{max}	*resource capacity of the reconfigurable processor.*
M_{max}	*memory size of the RTR architecture.*
C_T	*reconfiguration time of the reconfigurable processor.*
k	*the block-processing factor for the design.*

The behavior specifications are in the form of a directed graph called the *Task Graph*. The vertices in the graph denote tasks, and the edges denote the dependency among tasks. Data communicated between two

المنارة للاستشارات
www.manaraa.com

tasks, $B(t_i, t_j)$, will have to be stored in the on-board memory of the processor, if the two tasks connected by an edge are placed in different temporal partitions. The target architecture parameters specify the underlying resources and the reconfiguration time, C_T, for the device. Typically, resource capacity, R_{max}, is the combinational logic blocks/function generators on the FPGAs of the reconfigurable device. M_{max}, is the memory for storage of intermediate data available on the reconfigurable processor. k, the block-processing factor is the lower bound on the number of computations that this design will usually perform. The total intermediate data for k computations of the task graph has to fit in the memory M_{max} of the RTR processor. The user can give k to be the minimum number of iterations of the implicit loop, I, shown in Fig. 5 for typical runs of the application.

6.1. *Preprocessing*

Design Point Generation: Each task in the task graph is processed by a design space exploration and estimation tool [15] which is part of a high level synthesis system. The high level estimation tool generates a set of *design points* for each task. Each design point is characterized by its area and latency. Each task t will have a set of estimated design points, M_t. We state this formally as

M_t	*set of design points*, m, *for a task* $t \in T$.
$R(m)$	*area of a design point* $m \in M_t$.
$D(m)$	*latency of a design point* $m \in M_t$.

Partition Bounds Estimation: To find the number of partitions over which the temporal partitioning solution should be explored we calculate two bounds:

1. *Lower Bound*: For calculating the lower bound on number of partitions N^l_{min}, we sum the *minimum* area design point, m, for each task. This value divided by the FPGA area will be the minimum number of partitions required to obtain a solution.

$$N^l_{min} = \sum_{t \in T} \frac{R(m)}{R_{max}}, \ \{m \mid \forall m \in M_t, min(R(m))\} \tag{1}$$

2. *Upper Bound*: Ideally, we would like to establish an upper bound on the number of partitions needed to be explored by the partitioner when the maximum area design point for each task is chosen. However, we cannot accurately establish, this upper bound on the maximum number of partitions. This is because if a task is too large to fit in some temporal partition, it must go to a later partition. Then all the descendents of this task also cannot occupy the earlier temporal partition even if they can fit in it because the dependency among the tasks will be violated. This will leave some area on temporal partitions unoccupied due to dependency constraints, and the task graph will not fit even though there is enough area left unoccupied on the partitions. We could have established an upper bound on the maximum number of partitions to be equal to the number of tasks in the task graph. However, this is a very pessimistic bound and usually so many partitions need not be explored. We first define, the minimum number of partitions, N^u_{min}, that need to be explored if the *maximum* area design point for each task is mapped by the partitioner, to be

$$N^u_{min} = \sum_{t \in T} \frac{R(m)}{R_{max}}, \ \{m \mid \forall m \in M_t, max(R(m))\} \tag{2}$$

To determine the upper bound on the number of temporal partitions that need be explored to get an optimal solution, we define a user controlled parameter γ, called the *Partition Relaxation*. γ defines the number of partitions beyond N^u_{min} that must be explored while searching for better solutions. We have introduced parameter, γ, so that a user can direct the partition space search if the user has more knowledge of the solution to the problem. Or, this evaluation of γ can be be done automatically by the tool using heuristic techniques. Using a heuristic, if we map the maximum area design points for each task we arrive at a solution with partition size N''. This can be an upper bound on the partition size. If $N'' > N^u_{min}$, then $\gamma = N'' - N^u_{min}$. We give an example of how this can be done in Fig. 8. A task graph annotated with the maximum area of each task is shown. If R_{max} is 100, then we calculate N^u_{min} to be 4 partitions. A heuristic algorithm maps the tasks as shown into 6 partitions. Therefore $N'' = 6$, and $\gamma = 6 - 4 = 2$. The optimal solution for this graph is obtained in 5 partitions. We claim that any solution obtained by a heuristic using the maximum area (minimum delay) design points will never have its number of partitions less than that of an optimal solution for the same graph. We are currently studying how to achieve tighter upper and lower bounds for partition size and incorporating them automatically in our

المنارة للاستشارات
www.manaraa.com

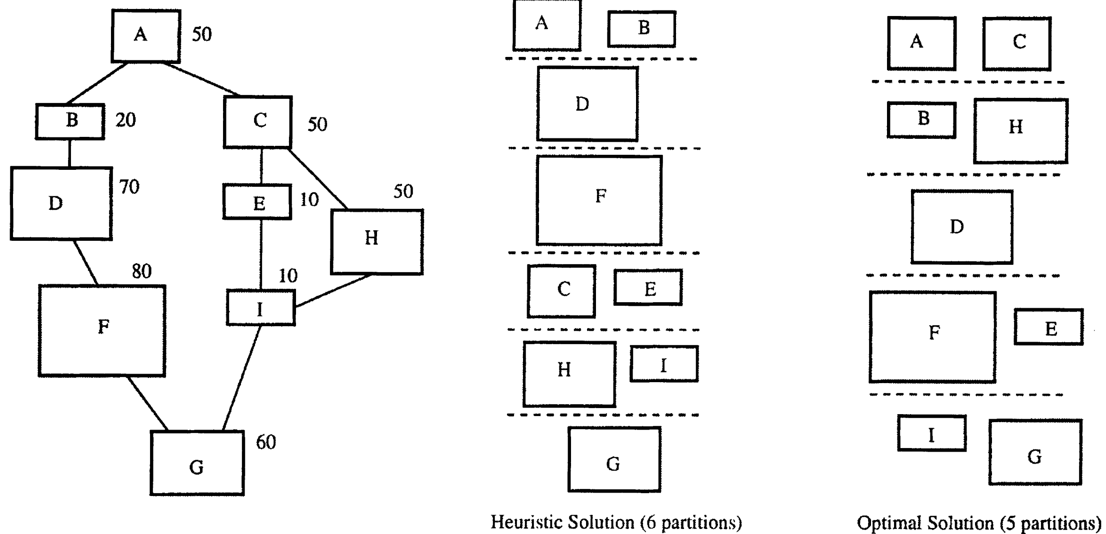


Figure 8. Generation of partition size upper bound.

algorithm. However, the facility of giving γ will still be provided to the user.

In the worst case, the total number of partitions to be explored range from the partition lower bound, N^l_{min}, to the number of tasks in the task graph, $|T|$. Therefore the value of γ can range from 0 to $|T| - N^u_{min}$. We may not get the optimal solution possible for the task graph if the value of γ is not set correctly.

6.2. *Partition Space Exploration Algorithm*

To explore better solutions for the temporal partitioning problem, we need to explore more than one partition bound. The partition bound is the number of partitions for which the current model has been formed and a solution is being explored. Finding the ideal partitions for the overall optimal solution is an iterative procedure, shown in Fig. 9. Informally, the algorithm consists of the following steps

1. The starting partition bound is $N = N^l_{min}$.
2. Obtain an optimal solution for the given partition bound, N. The design execution time achieved after solving for this partition bound is D_a. If $N = N^u_{min} + \gamma$, then stop.
3. Increase the partition bound, $N = N + 1$, and reformulate the problem with the new partition size. Also introduce a design execution time constraint so that the result is bounded by the execution time delay already achieved, D_a. Go to step 2.

We calculate the bounds on the number of partitions, N^l_{min} and N^u_{min}, as described earlier. We start the search at N^l_{min} and obtain an optimal solution, by forming and solving an ILP model of the temporal partitioning problem. The details of the model are described in Section 6.2.1. For the first ILP model there is no upper bound on the constraint on the execution time of the design. The result of solving this model is a temporal partitioning solution for N partitions and the execution time D_a of the solution. We now relax N by 1, form and solve the ILP model again. This time since we are looking for a better solution than the one we have already achieved, D_a is the execution time constraint for the new ILP model. We continue to relax N and look for better solutions until the value of N reaches $N^u_{min} + \gamma$.

6.2.1. *ILP Formulation for Design Space Exploration.*

We build the temporal partitioning model for the given tasks and their design points and the values of N and k. In the following discussion we present the variables and equations of the ILP model.

Variable y_{tpm} models partitioning and design point selection for a task and is described formally as

$$y_{tpm} = \begin{cases} 1 & \text{if task } t \in T \text{ is placed in partition } p, \\ & \quad 1 \le p \le N, \text{ using design point } m \in M_t \\ 0 & \text{otherwise} \end{cases}$$

المنارة للاستشارات
www.manaraa.com

Algorithm $Refine_Partition_Bound()$
begin
  $N^u_{min} \leftarrow$ MaxAreaPartitions()
  $N^l_{min} \leftarrow$ MinAreaPartitions()
  $N \leftarrow N^l_{min}$ /* starting partition bound */
  FormILPModel() /* Model with no execution time constraint */
  $D_a \leftarrow$ SolveILPModel_Optimal()
  while $D_a = 0$ **and** $N < N^u_{min} + \gamma$ /* Partition bound was infeasible */
    $N \leftarrow N + 1$ /* next partition bound */
    FormILPModel() /* Model with no execution time constraint */
    $D_a \leftarrow$ SolveILPModel_Optimal()
  end while
  while $N < N^u_{min} + \gamma$
    $N \leftarrow N + 1$ /* Relax N */
    FormILPModel() /* Model with execution time constraint $\leq D_a$ */
    $D'_a \leftarrow$ SolveILPModel_Optimal()
    if $D'_a \neq 0$ /* solution is feasible */
      $D_a \leftarrow D'_a$
    end if
  end while
  return(D_a) /* return with the last known best solution */
end Algorithm $Refine_Partition_Bound$

Figure 9. Partition refinement procedure.

where, N *is bound on the number of partitions.*

Variable y_{tpm} is a 0–1 variable.

Uniqueness Constraint: Each task should be placed in exactly one partition among the N temporal partitions, while selecting one among the various design points for the task.

$$\forall t \in T : \sum_{m \in M_t} \sum_{p=1}^{N} y_{tpm} = 1 \qquad (3)$$

Temporal Order Constraint: Because we are partitioning over time, a task t_1 on which another task t_2 is dependent cannot be placed in a later partition than the partition in which task t_2 is placed. It has to be placed either in the same partition as t_2 or in an earlier one. This constraint makes sure that the dependency constraints among the tasks are maintained. No task should execute earlier than a task on which it is dependent.

$$\forall t_2, \forall t_1 \rightarrow t_2, \forall p_2,\ 1 \leq p_2 \leq N-1$$
$$: \sum_{m_1 \in M_{t_1}} \sum_{p_2 < p_1 \leq N} y_{t_1 p_1 m_1} + \sum_{m_2 \in M_{t_2}} y_{t_2 p_2 m_2} \leq 1 \qquad (4)$$

Resource Constraint: The sum of area costs of all the tasks mapped to a temporal partition must be less than the overall resource constraint of the reconfigurable processor. Typical FPGA resources include function generators, configurable logic blocks etc. Similar equations can be added if multiple resource types exist in the FPGA.

$$\forall p, 1 \leq p \leq N : \sum_{m \in M_t} \sum_{t \in T} (y_{tpm} * R(m)) \leq R_{max} \qquad (5)$$

Memory Constraint: Intermediate data due to data transfer among dependent tasks will be stored in a

المنارة للاستشارات
www.manaraa.com

partition under two conditions. If the memory has been written in an earlier partition, and is to be read in this partition or any partition later than this partition. Or, if memory is being written in the current partition and is destined to be read in a later partition. Data transfer through memory will not take place if two dependent tasks are placed in the same temporal partition. Variable $w_{pt_1t_2}$ models data transfer requirement across partition boundaries for dependent tasks. It is stated formally as

$$w_{pt_1t_2} = \begin{cases} 1 & \text{if task } t_1 \text{ is placed in any partition } 1 \cdots p-1 \text{ and } t_2 \text{ is placed in any of } p \cdots N \text{ and } t_1 \rightarrow t_2 \\ 1 & \text{if task } t_1 \text{ is placed in partition } p \text{ and } t_2 \text{ is placed in any of } p+1 \cdots N \text{ and } t_1 \rightarrow t_2 \\ 0 & \text{otherwise} \end{cases}$$

$w_{pt_1t_2}$ is a 0–1 variable. It is a secondary variable which is described in terms of the y_{tpm} variables.

The intermediate data needs to be stored and should be less than the memory, M_{max}, of the reconfigurable processor. The variable $w_{pt_1t_2}$, if 1, signifies that t_1 and t_2 have a data dependency and are being placed across temporal partition p. Therefore the data being communicated between them, $B(t_1, t_2)$, will have to be stored in the memory of partition p. The following equation represents the memory constraint. It contains terms to represent the intermediate data transfer due to dependent tasks as discussed earlier. Since our memory model is such that all external inputs and outputs with the host also takes place through the memory, the equation also contains terms to represent the amount of data that has to read as input from the environment(host) and written out to the environment(host) by the tasks.

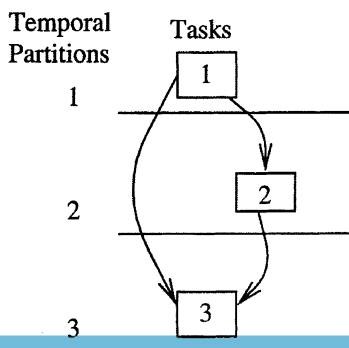
$$\forall p, 1 \leq p \leq N: \sum_{t \in T} \sum_{p \leq p_2 \leq N} \sum_{m \in M_t} y_{tp_2m} * B(env, t) * k + \sum_{t \in T} \sum_{1 \leq p_3 \leq p} \sum_{m \in M_t} y_{tp_3m} * B(t, env) * k + \sum_{t_2 \in T} \sum_{t_1 \rightarrow t_2} (w_{pt_1t_2} * B(t_1, t_2) * k) \leq M_{max} \quad (6)$$

As discussed earlier the variable $w_{pt_1t_2}$ has to model communication among tasks which can be mapped to adjacent and non-adjacent temporal partitions. In Fig. 10, we show how this variable models data transfer for a small taskgraph fragment. In the example shown there is no data transfer from the host only tasks communicating to each other. We show in the figure the original equations used to model the constraints for temporal partitions 2 and 3. The result equations show the $w_{pt_1t_2}$ variables which will be 1 in the mapping of tasks to partitions shown in the example and the constraints which has to be satisfied. $w_{pt_1t_2}$ are non-linear terms and can be generated by the following set of equations:

$$\forall p, 1 \leq p \leq N, \forall t_2 \in T, \forall t_1 \rightarrow t_2 : w_{pt_1t_2} \geq \sum_{1 \leq p_1 < p} \sum_{m_1 \in M_{t_1}} y_{t_1p_1m_1} * \sum_{p \leq p_2 \leq N} \sum_{m_2 \in M_{t_2}} y_{t_2p_2m_2} \quad (7)$$

$$\forall p, 1 \leq p \leq N, \forall t_2 \in T, \forall t_1 \rightarrow t_2 : w_{pt_1t_2} \geq \sum_{m_1 \in M_{t_1}} y_{t_1pm_1} * \sum_{p+1 \leq p_2 \leq N} \sum_{m_2 \in M_t} y_{t_2p_2m_2} \quad (8)$$

Equations (7) and (8) are non-linear. We can use linearization techniques [34, 35] to transform the non-linear equations into linear ones, so that the model can



MODELLING EQUATIONS:

$W_{212} * B(1,2)*k + W_{213} * B(1,3)*k + W_{223} * B(2,3)*k <= M_{max}$

$W_{312} * B(1,2)*k + W_{313} * B(1,3)*k + W_{323} * B(2,3)*k <= M_{max}$

RESULT EQUATIONS:

$W_{212} * B(1,2)*k + W_{213} * B(1,3)*k + W_{223} * B(2,3)*k <= M_{max}$

$W_{313} * B(1,3)*k + W_{323} * B(2,3)*k <= M_{max}$

Figure 10. Memory constraint.

المنارة للاستشارات
www.manaraa.com

be solved by a Linear Program solver. Linearization techniques have been used successfully before in [10] to solve the combined temporal partitioning and synthesis problem.

Execution Time Constraint: When the problem is formulated we have as input the partition bound N over which the current solution is to be explored. Variable η is the actual number of partitions finally used in the solution and will be less than or equal to N. Variable d_p models the execution time of a temporal partition.

$$\eta = \text{Number of partitions actually used in solution.}$$

$$d_p = \text{execution time of partition } p.$$

η is an integer variable and d_p can be an integer or real variable depending on whether the latency values are integer or real. The following definitions will be used to generate the execution time constraint.

D_a *constraint on the execution time of the design.*

T_l *set of tasks* $t_i \in T$, *where* $\forall t_j \in T$, $\neg(t_i \to t_j)$, *(leaf tasks of* T*).*

T_r *set of tasks* $t_j \in T$, *where* $\forall t_i \in T$, $\neg(t_i \to t_j)$, *(root tasks of* T*).*

$t_i \xrightarrow{p} t_j$ *a directed path from* $t_i \in T$ *to* $t_j \in T$.

$P_{l \xrightarrow{p} r}$ $\{t_i \xrightarrow{p} t_j \mid (t_i \in T_r) \wedge (t_j \in T_l)\}$, *(set of paths from root tasks to leaf tasks).*

The execution time of a partition will be the maximum execution time among all the paths of the task graph mapped to that partition. In Fig. 11, we show how the execution time for a partition is determined. The final mapping of tasks to partitions, with the latency value for each task, is shown. In partition 1, three paths are mapped. The latency of this partition is the greatest latency along a path mapped to the partition, i.e., maximum among 350 ns, 400 ns, 150 ns. The maximum latency in partition 2 is 300 ns. If the block-processing factor is k, then the execution time of the partition is the latency multiplied to the block-processing factor. Formally the execution time of a temporal partition is given as

$$\forall p, 1 \le p \le N, \forall (t_i \xrightarrow{p} t_j) \in P_{l \xrightarrow{p} r}$$

$$: \sum_{m \in M_t} \sum_{t \in t_i \xrightarrow{p} t_j} (y_{tpm} * D(m) * k) \le d_p \qquad (9)$$

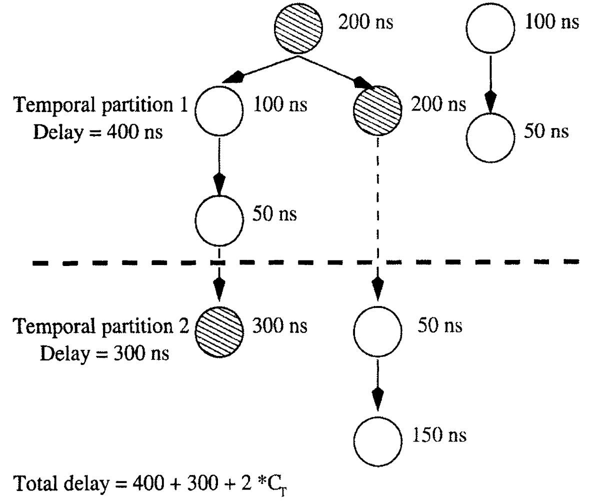


Figure 11. Execution time estimation.

All temporal partitions $1 \cdots N$ used in the formulation, may not be used in the final solution, if the tasks can fit in lesser number of partitions. To calculate the actual number of partitions used in the solution, we determine the highest numbered partition used by any leaf level task in the task graph by the following equation:

$$\forall t \in T_l : \sum_{m \in M_t} \sum_{p=1}^{N} (p * y_{tpm}) \le \eta \qquad (10)$$

Now the execution time constraint on the overall design can be stated in terms of Eqs. (9) and (10) as

$$\eta * C_T + \sum_{p=1}^{N} d_p \le D_a \qquad (11)$$

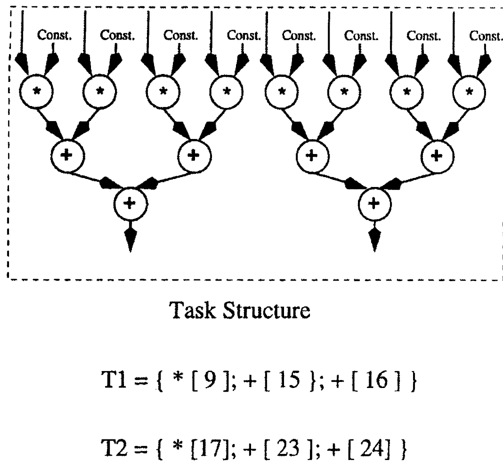
As discussed earlier, this constraint is used to search for a better solution as different partition bounds are being explored in Algorithm *Refine_Partition_Bound* in Fig. 9.

Optimality Goal: The most optimal solution will be the design with the least execution time.

$$\textit{Minimize} \quad : \eta * C_T + \sum_{p=1}^{N} d_p \qquad (12)$$

The solution of this ILP model gives us the optimum temporal partitioning for the give partition bound N, the block-processing factor k, and the set of design points for the tasks. If the amount of intermediate memory required to process k computations exceeds the memory constraint M_{max} of the architecture then

المنارة للاستشارات
www.manaraa.com



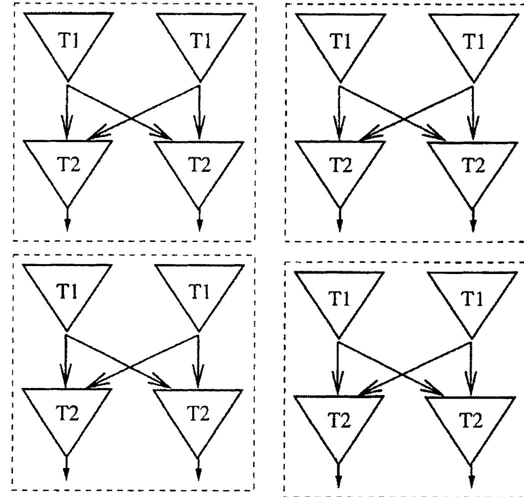


Figure 12. Task graph for DCT.

the user needs to reduce k and temporally partition the design again.

6.3. Experimental Results for Optimal Search Algorithm

We performed temporal partitioning on the 4×4 Discrete Cosine Transform (DCT) which is the most computationally intensive part of the JPEG [36] algorithm. In this study, the DCT is a collection of 16 tasks as shown in the Fig. 12. On the left of the figure we show the internal structure of a task in the DCT. There are two kinds of tasks in the task graph, $T1$ and $T2$, whose structure is similar but whose operations have different bit widths. Task $T1$ represents two vector multiplications in the first dimension of the DCT. Task $T2$ represents two vector multiplications in the second dimension of the DCT. We obtained all the design points for each kind of tasks by using estimation tools integrated in the SPARCS design environment [15], on the individual tasks. The functional units, area and latency costs for each is shown in Table 1.

In Tables 2–4 we present the results of our temporal partitioning tool. In all the tables, R_{max} is the resource constraint of the FPGA, C_T the reconfiguration time, k the block-processing factor, and N the number of partitions onto which the design is partitioned. The latency of the final design (with the reconfiguration overhead), is shown in the column Latency. The execution time of the design for the k blocks of data is given in the column Design Execution Time. Design Execution Time/k shows the average execution time per computation, Mem. Overhead shows the amount

Table 1. Design points for DCT tasks.

Task	Desgn. Pt.	Characteristics					
		Area (CLBs)	Latency (ns)	$*_9$	$+_{16}$	$*_{16}$	$+_{24}$
T1	1	336	375	8	4	–	–
	2	286	500	6	2	–	–
	3	220	625	4	2	–	–
	4	194	750	2	2	–	–
	5	174	875	1	2	–	–
T2	1	396	420	–	–	8	4
	2	356	560	–	–	6	2
	3	292	700	–	–	4	2
	4	276	840	–	–	2	2

of maximum memory stored in any of the temporal partitions (excluding the memory used to store the input and outputs) of the solution in terms of the number of words of the hardware. T(s) is the time taken by our temporal partitioning tool to execute in seconds. All experiments were run using an ILP solver called CPLEX on an UltraSparc Machine running at 175 MHz with 120MB memory.

In Table 2, we present the result of temporal partitioning and design space exploration of the DCT with and without block-processing factors. In all experiments the reconfiguration time considered is similar to the Xilinx 6200 series FPGAs. In Exp. 1, for a block-processing factor of 3,000, our temporal partitioning tool explores 3 temporal partitions for the design and results in a latency of 60,795 ns. In Exp. 2, with a block-processing factor of 1 (i.e., no computations are being

المنارة للاستشارات
www.manaraa.com

Table 2. Results for combined design-space exploration and block-processing.

Exp.	R_{max} (CLBs)	C_T (μs)	k	N	Latency (ns)	Design execution time	Design execution time/k (ns)	Mem. overhead	T (s)
1	4,000	30	3,000	1	31,590	4,800 μs	1,600	0	1
				2	60,795	2,445 μs	815	48,000	1
				3	–	–	–	–	Infeasible
2	4,000	30	1	1	31,590	31,590 ns	31,590	0	1
3	2,304	30	3,000	2	61,590	4,830 μs	1,610	48,000	11
				3	91,215	3,735 μs	1,245	48,000	22
				4	–	–	–	–	Infeasible
4	2,304	30	1	2	61,590	61,590 ns	61,590	16	57

Table 3. Results for different reconfiguration overheads.

Exp.	R_{max} (CLBs)	C_T	k	N	Latency (ns)	Design execution time (μs)	Mem. overhead	T (s)
5	2,304	30 ns	300	2	1,650	477.06	4,800	17
				3	1,305	364.59	4,800	19
6	2,304	30 ns	50	2	1,650	79.56	700	25
				3	1,305	60.84	700	7
7	2,304	3 ms	3,000	2	6,001,590	10,770	48,000	80
8	2,304	3 ms	30,000	2	6,001,590	53,700	480,000	29
				3	9,001,215	45,450	480,000	36

Table 4. Results for design-space exploration.

Exp.	R_{max} (CLBs)	C_T (μs)	k	N	Latency (ns)	Design execution time (μs)	Mem. overhead	T (s)
9	2,304	30	3,000	2	61,715	5,145.6	48,000	1
10	2,304	30	3,000	2	61,590	4,830	48,000	22
				3	91,215	3,735	48,000	204

sequenced), the tool gives a minimum latency design of 31,590 ns and uses just one temporal partition. This results in a statically configured design. Even though, the latency of the statically configured design in Exp. 2 is less than that of Exp. 1, this is not the best possible solution. This is because, if multiple computations are computed on both the static and RTR design, the RTR design will outperform the static design. For executing 3,000 computations, the RTR design will take 2,445 μsec, while the static design will take 4,800 μsec. This is a 49% improvement of the RTR design over the static design. Exp. 3 and 4 were performed for different FPGA size of 2,304 CLBs, which is the size of a Xilinx XC4062. In Exp. 3, again with a block-processing factor of 3000, the optimal design takes 3 temporal partitions with the latency of the design being 91,215 ns. For Exp. 4, with no block-processing factor the optimal latency of the design is 61,590 ns. Again, the actual execution time of the design when the block-processing factor is considered while exploring the design space is superior. In all the experiments the value of M_{max} is 64 K.

The experiments in Table 2 illustrate that combining block-processing and design space exploration gives better temporal partitioning solutions. If the block-processing factor is not considered at the time of temporal partitioning (i.e., is equal to 1), then the temporal partitioning tool will tend to pick the design with minimum number of temporal partitions. If a relevant block-processing factor is given the tool will search for a faster design with more temporal partitions, because block-processing will amortize the effects of

www.manaraa.com

reconfiguration overhead. Since we understand that the block-processing is necessary for good performance of a temporally partitioned design, we must integrate this idea early in the design process, while partitioning and design point selection is being performed.

Similar results will hold if the reconfiguration overheads are varied. In Table 3, we show results for different reconfiguration overheads. In Exp. 5 and 6, the reconfiguration overhead is in nano-seconds (similar to the reconfiguration overheads of context-switching FPGAs like the Time Multiplexed FPGA [37, 38]). In Exp. 7 and 8, the reconfiguration overhead is in milli-seconds (similar to commercially available reconfigurable hardware, the Wildforce board with Xilinx FPGAs [39]). As the reconfiguration overhead decreases we observe that for small values of k, the exploration process chooses more temporal partitions. However, for the reconfiguration overheads in milli-seconds even for values of k as large as 3,000 the temporal partitioner chooses designs with minimum temporal partitions. So for an architecture which has a very high reconfiguration a large number of blocks must be processed to amortize the cost of the reconfiguration overhead. Such is the case in Exp. 8 where for an overhead of 3 ms, 30,000 computations need to be sequenced to overcome the effect of the reconfiguration overhead and for the tool to partition the design over 3 temporal partitions. From these experiments we see that given the block-processing factor and the architecture constraints the temporal partitioning tool will select the most appropriate design point and the placement of tasks on partitions.

In Table 4, we illustrate how design space exploration is beneficial. For same values of the block-processing factor k, we perform experiment with and without design space exploration. In Exp. 9, temporal partitioning is performed with only one design point for each task, the minimal area design point. In Exp. 10, all the design points are used. Again we observe that the tool chooses the most appropriate design points for the given constraints, when multiple design points are given to it, and results in a 27% improvement of the design in Exp. 10. Therefore design space exploration must be *integrated* with and performed during the temporal partitioning process, rather than choosing the design point *before* temporal partitioning is performed.

The optimal solution process described in this section produced results in less run-times of the temporal partitioning tool when the size of the problem to be solved is not very large. For eg., a task graph of size 15 tasks, 3 temporal partitions and 3 design points per task solved quickly. But a task graph of 30 tasks, 6 temporal partitions and 3 design points per task took many hours to solve.

7. Temporal Partitioning and Design Space Exploration by Iterative Search Algorithm

To handle larger problem sizes, we have therefore developed a novel method of solving the ILP problem iteratively. With this method we break the large solution space and window in to smaller regions of the solution space progressively, to obtain near-optimal solutions for the problems. Instead of solving each ILP problem to global optimality we break the search space of the algorithm into smaller sections. An ILP problem for a section of the search space is formed and a constraint satisfying solution is generated. Success or failure of a search guides the algorithm to move iteratively into the next region of search while improving the solution. There can be many ways of dividing the search space into smaller sections. We have approached the problem by dividing the search space by a binary subdivision method.

7.1. Preprocessing

In this section, we discuss the additional preprocessing steps which need to be undertaken for the new algorithm that iteratively explores different regions of the design space. The other preprocessing steps of *Design Point Generation* and *Partition bounds Estimation* are as discussed in Section 6.

Execution Time Bounds Calculation: The execution time of the temporally partitioned design will involve two components—(1) execution time due to the actual execution of the tasks in each temporal partition for the given block-processing factor k, (2) execution time due to the reconfiguration overhead. For a given number of temporal partitions, N, we can calculate the upper and lower bounds on the execution time of the design as follows:

1. *Maximum Execution Time*: The worst case execution time D_{max}, will occur when all tasks are serially executed. For upper bound calculation, we will use the design point with maximum execution time for each task. The execution time for each task multiplied to the block-processing factor will give us the execution time of the design without considering

المنارة للاستشارات www.manaraa.com

the overhead of reconfiguration. This time added to the reconfiguration overhead will be the upper bound design execution time for N partitions.

$$D_{max} = \sum_{t \in T} D(m) * k + N * C_T \qquad (13)$$

2. *Minimum Execution Time*: For obtaining the lower bound for N partitions, we consider for each task the fastest (minimum latency) design point. We obtain the latency for all the paths in the task graph, by summing up the minimum latency of the tasks along each path. The maximum latency value over all such path latencies in the task graph gives us the lower bound on the latency. This latency value is multiplied by the block-processing factor to derive the execution time lower bound without reconfiguration overhead. This execution time added to the reconfiguration overhead will be the lower bound on design execution time for N partitions. In the following equation, p is a path in the task graph.

$$D_{min} = \max_{p} \{\text{latency of } p \text{ with fastest design point for each task in } p\} * k + N * C_T \qquad (14)$$

7.2. *Algorithm for Design Execution Time Reduction*

Fig. 13, describes the design execution time reduction algorithm. It is an iterative procedure that obtains near-optimal execution time solutions for a given partition bound, N, and execution time bounds D_{max}

```
Algorithm Reduce_ExecutionTime(N, D_max, D_min)
begin
  D_a ← 0
  FormILPModel()
  if SolveILPModel_Feasible() = Infeasible subject to Timeout
    return(D_a)
  D_a ← CalculateSolnDelay() /* Achieved execution time of solution */
  while (D_max − D_min ≥ δ) and (D_a − D_min ≥ δ)
    D'_max = D_max
    /* Binary subdivision of achievable design execution time range */
    D_max = (D_max + D_min)/2
    while (D_max ≥ D_a)
    /* we have already achieved execution time D_a which is less than D_max */
      D_max = (D_max + D_min)/2
    end while
    FormILPModel()
    if SolveILPModel_Feasible() = Infeasible subject to Timeout
      /* increase lower bound to overcome infeasibility */
      D_min = D_max
      D_max = D'_max
    else
      D_a ← CalculateSolnDelay()
    end if
  end while
  return(D_a)
end Algorithm Reduce_ExecutionTime
```

Figure 13. Iterative procedure for reducing design execution time.

المنارة للاستشارات
www.manaraa.com

and D_{min}. The procedure for obtaining appropriate partition bounds was explained in Section 6.1. It finds a constraint satisfying solution between D_{max} and D_{min}. Once a solution is obtained, the upper bound is reduced to $(D_{max} + D_{min})/2$, and a new solution for these constraints is found. If a feasible solution is obtained, then the obtained execution time of the solution becomes the upper bound for a new search. If no feasible solution is obtained, then this execution time becomes the new lower bound. It continues this binary subdivision on the execution time bounds, till the difference between the upper and lower bounds becomes very small, or no more feasible solutions are found. The tolerable difference between the lower and upper execution time bounds for the design is a user defined parameter, δ, called the *Design Execution Time Tolerance*. Design Execution Time Tolerance defines how much of the design space can be left unexplored in one run of the algorithm. If the tolerance is small, more iterations will be spent in obtaining a solution, thus increasing the run time. If a large run time is not acceptable then this tolerance can be increased. The optimality of the solution will be affected by the value of δ. If δ is very large then the algorithm may miss some solution which is better than the one found. We have shown in the experiments the effect of changing the value of δ on the search process. In practice, we can set the execution time tolerance to a small percentage of the *MaxExecutionTime* of the task graph.

We again use the temporal partitioning and design space exploration problem as modeled as an ILP (presented in Section 6.2.1), with some modifications discussed later. We do not use the ILP for finding optimal solutions, but instead use it to obtain a feasible solution for a problem. That is, the optimization goal explained in Section 6.2.1 is removed and some new constraints are added. These constraints will be presented shortly. Our reduction procedure then makes the constraints tighter, reformulates the ILP and solves it for the new problem. For larger designs, therefore we have developed this directed search procedure, which reduces the search space for each run of the ILP solver, while still exploring the whole search space. This claim has been substantiated, by observing that for small designs the solution obtained by this procedure and an ILP solved to optimality is the same, as discussed in Section 6.3. In the algorithm, the procedure *FormILPModel*() forms the ILP model. The procedure *SolveILPModel_Feasible*() then solves the model by a linear program solver and returns with the first feasible constraint satisfying solution.

7.3. Partition Space Exploration Algorithm

The partition space exploration procedure for the iterative execution time search is shown in Fig. 14. It is similar to the partition exploration procedure discussed earlier in Section 6.2.1, the only difference being that the iterative search algorithm *Reduce_ExecutionTime* is called to explore different temporal partitioning solutions for each partition bound rather than solving the problem to optimality. Informally, the algorithm consists of the following steps:

1. The starting partition bound is $N = N^l_{min}$.
2. Obtain a constraint satisfying solution for partition bound, N, and execution time constraints D_{max} and D_{min} for this partition bound.
3. Find lower execution time solutions by progressively exploring different regions of the search space, by tightening the execution time constraints, for the current partition bound. If $N = N^u_{min} + \gamma$, then stop.
4. Increase the partition bound, $N = N + 1$, and go to step 2.

7.4. Modifications to the ILP model

The ILP model discussed in the Section 6.2.1 remains the same, with some small modifications. We have two execution time constraints instead of Eq. (11) in the model. These are described below.

$$\eta * C_T + \sum_{p=1}^{N} d_p \leq D_{max} \tag{15}$$

$$\eta * C_T + \sum_{p=1}^{N} d_p \geq D_{min} \tag{16}$$

7.5. Experimental Results for the Iterative Constraint Satisfaction Algorithm

Case Study of AR filter: We present a case study of the Auto Regressive (AR) lattice filter [40] that has applications in signal and speech processing applications. In this experiment we demonstrate the closeness of the solution obtained by the iterative constraint satisfaction algorithm presented in this section and the optimal algorithm described in Section 6. The task graph for the specification consists of 6 tasks is shown in Fig. 15. Tasks A and B show the internal structures of the filter tasks. Tasks T1, T3, & T4 have a structure like

المنارة للاستشارات
www.manaraa.com

Algorithm *Refine_Partition_Bound*()
begin
$N^u_{min} \leftarrow$ MaxAreaPartitions()
$N^l_{min} \leftarrow$ MinAreaPartitions()
$N \leftarrow N^l_{min}$ /* starting partition number */
$D_{max} \leftarrow$ MaxExecutionTime(N)
$D_{min} \leftarrow$ MinExecutionTime(N)
$D_a \leftarrow Reduce_ExecutionTime(N, D_{max}, D_{min})$
while $D_a = 0$ /* Partition bound was infeasible */
$N \leftarrow N + 1$ /* next partition number */
$D_{max} \leftarrow$ MaxExecutionTime(N)
$D_{min} \leftarrow$ MinExecutionTime(N)
$D_a \leftarrow Reduce_ExecutionTime(N, D_{max}, D_{min})$
end while
while $N < N^u_{min} + \gamma$
$N \leftarrow N + 1$ /* Relax N */
$D_{min} \leftarrow$ MinExecutionTime(N)
if $D_{min} \geq D_a$
return(D_a) /* This is the best solution */
else
/* find a better solution by taking D_a as upper bound */
$D'_a \leftarrow Reduce_ExecutionTime(N, D_a, D_{min})$
if $D'_a \neq 0$ /* Feasible */
$D_a \leftarrow D'_a$
end if
end if
while
return(D_a) /* return with the last known best solution */
end Algorithm *Refine_Partition_Bound*

Figure 14. Partition refinement procedure.

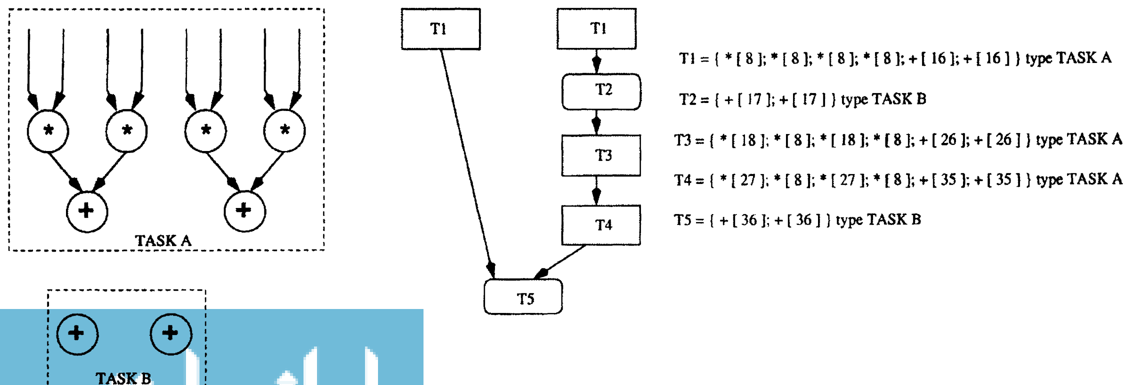


Figure 15. Task graph for the AR filter.

www.manaraa.com

Table 5. Design points for the AR filter tasks.

t	M_t	Characteristics									
		Area	Latency	$*_8$	$+_{16}$	$+_{17}$	$*_{18}$	$+_{26}$	$*_{27}$	$+_{35}$	$+_{36}$
T1	1	120	250	4	2						
	2	104	375	2	2						
	3	84	625	1	1						
T2	1	30	125			2					
T3	1	170	320	2			2	2			
	2	118	480	1			1	1			
T4	1	222	400	2					2	2	
	2	155	600	1					1	1	
T5	1	54	200								2

Table 6. Temporal partitioning of the AR filter, $R_{max} = 196$, $C_T = 30\ \mu s$, $\gamma = 0$, $\delta = 100\ \mu s$, $k = 3000$.

N	Result(Iterative)				Result(Optimal)	
	I	D_{max} (μs)	D_{min} (μs)	Design execution time (μs)	Design execution time (μs)	Mem. overhead
3	1	8,055	3,975	Inf.	Inf.	
4	1	8,085	4,005	6,210		
	2	6,045	4,005	5,355		
	3	5,025	4,005	Inf.		
	4	5,280	5,025	Inf.	5,355	15,000
5	1	5,355	4,035	5,010		
	2	4,650	4,035	Inf.		
	3	4,950	4,650	Inf.	5,010	18,000

Task A, but differ in the bit-widths of their operations. Tasks T2 and T5 are like Task B, but again differ in their bit-widths. The bit widths of each operation in each task is also shown in the figure. The design points are shown in Table 5. These design points were again estimated using an estimation tool integrated in [15]. Task T1 has three design points, tasks T3 & T4 have two design points each, and tasks T2 and T5 have one design point each. The result of the experimentation is shown in Table 6. N denotes the number of temporal partitions explored. The columns under Result(Iterative) state the result of running the iterative algorithm. I is the iteration of the algorithm, D_{max} and D_{min} are the design execution time bounds for that iteration calculated by the algorithm. D_a gives the design execution time of the solution. *Result(Optimal)* is the result achieved by solving the problem to optimality using the algorithm described in Section 6. Mem. Overhead shows the amount of maximum memory stored in any of the temporal partitions (excluding the memory used to store the input and outputs) of the solution in terms of the number of words of the hardware. We use *CPLEX* to solve the ILP problems both for constraint satisfaction and optimal solution. We see that the result of our algorithm matches the optimal solution for this task graph. We have performed a lot of experiments on small task graphs and the solution for our iterative procedure and an optimally solved ILP has been the same.

Case Study of DCT: For task graphs with larger number of tasks, the iterative constraint satisfaction approach is able to explore in reasonable time more solution space than solving the problem to optimality. To demonstrate this, we again undertook a case study of the 4×4 DCT, however this time the size of each task is smaller. In this study, DCT was modeled in the form of 32 vector products. The entire DCT is a collection of 32 tasks, where each task is a vector

74
www.manaraa.com

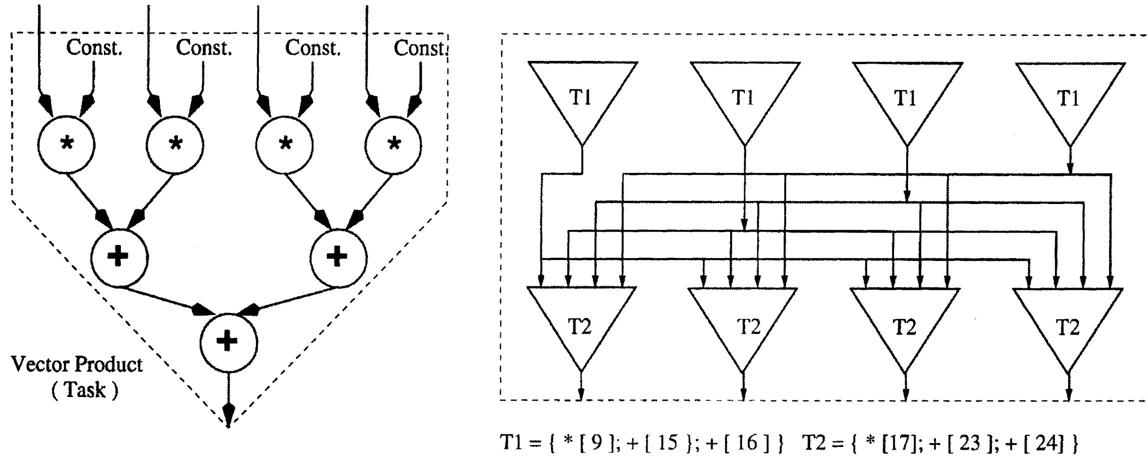


Figure 16. Task graph for DCT, 8 of the 32 tasks are shown.

product. A vector product is shown in Fig. 16. There are two kinds of tasks in the task graph, $T1$ and $T2$, whose structure is similar to the vector product, but whose bit-widths differ. A collection of eight tasks, forms a row of the 4×4 output matrix, as shown in the figure. The entire task graph consists of four such collections of tasks. Each task had three design points. Area and latency of the tasks for these design points were carefully estimated using an estimation tool [15]. The functional units, area and latency for each is shown in Table 7. The result of the iterative refinement procedure for minimizing the design execution time of the temporally partitioned DCT for various FPGA resource bound, R_{max}, and reconfiguration overhead, C_T, values is shown in Tables 8 through 11. For the current set of experiments, column N denotes the number of temporal partitions, D_{max} and D_{min} denote the maximum and the minimum design execution time bounds for the model being solved in that iteration. The design execution time of the solution produced is shown in the column Design Execution Time. Run times for the temporal partitioning tool, in seconds, are shown for each iteration of the algorithm separately in the column T(s). The total run time of the temporal partitioning tool in minutes for each experiment is shown in column T(m). All experiments have been run on an UltraSparc 1 machine running at 175 Mhz with 120 MB memory.

Table 7. Design points for DCT tasks.

		Characteristics					
Task	*D.*	Area	Latency	$*_{9}$	$+_{16}$	$*_{16}$	$+_{24}$
$T1$	1	180	375	4	2		
	2	138	500	2	2		
	3	121	750	1	2		
$T2$	1	216	420			4	2
	2	188	560			2	2
	3	162	840			1	2

In the first experiment, shown in Table 8, $R_{max} = 576$ CLBS (XC4013 fpga) and C_T is 30 μs and the block-processing factor $k = 3000$. The minimum number of partitions estimated by *MinAreaPartitions*() is 8 and by *MaxAreaPartitions*() is 11. We are able to reduce the execution time of the circuit in steps by doing a binary division. Once the difference between the maximum and minimum execution time is less than $\delta = 1000\ \mu$s, we stop. Then, the algorithm proceeds by searching the next partition bound by increasing N and repeats the iterative search procedure. We sometimes need to have a timeout, either if the problem is infeasible or a solution is too difficult to find. This timeout is shown in the results as Inf. For this set of experiments we kept the timeout to be 300 seconds to find each constraint satisfying solution. Notice that, while we are tightening the design execution time constraint in each iteration of the solution, we are in effect making the solver progressively look at different parts of the design space. Since Partition Relaxation, $\gamma = 1$, we stop our search at $N = 12$.

In the second experiment shown in Table 9, we present the temporal partitioning of the same design with no block-processing being performed i.e, $k = 1$. For this experiment, we have not shown the value of reconfiguration overhead $N * C_T$ in the table. We start

المنارة للاستشارات

www.manaraa.com

Table 8. DCT, $R_{max} = 576$, $\delta = 1000\,\mu$s, $\gamma = 1$, $k = 3000$.

			Bounds		Result		
C_T (μs)	N	I	D_{max} (μs)	D_{min} (μs)	Design execution time (μs)	T (s)	T (m)
30	8	1	76,580	2,625	Inf.	300	
	9	1	76,590	2,655	28,410	37.40	
		2	21,138	2,655	20,640	77.32	
		3	11,895	2,655	Inf.	300	
		4	16,515	11,895	Inf.	300	
		5	18,825	16,515	Inf.	300	
		6	19,980	18,825	Inf.	300	
	10	1	20,640	2,685	18,900	278.8	
		2	11,631	2,685	Inf.	300	
		3	16,104	11,631	Inf.	300	
		4	18,342	16,104	Inf.	300	
		5	18,621	18,342	Inf.	300	
	11	1	18,900	2,715	Inf.	300	
	12	1	18,900	2,745	Inf.	300	61.55

Table 9. DCT, $R_{max} = 576$, $\delta = 1000\,\mu$s, $\gamma = 1$, $k = 1$.

			Bounds (without $N * C_T$)		Result		
C_T (μs)	N	I	D_{max} (ns)	D_{min} (ns)	Design execution time (without $N * C_T$) (μs)	T (s)	T (m)
30	8	1	25,440	795	Inf.	300	
$\alpha = 0$	9	1	25,440	795	9,630	77.60	
		2	6,956	795	Inf.	300	
		3	9,266	6,956	9,100	78.95	
		4	8,111	6,956	81,00	185.73	
		5	7,533	6,956	7,380	281.93	
		6	7,244	6,956	Inf.	300	25.4

with 8 partitions, but no solution is possible. Then we relax the partition bound by 1, to 9 and continue the search for a solution. Notice that no relaxation of N was undertaken in this experiment, after a solution was achieved in 9 partitions. This is because, the algorithm *Refine_Partition_Bound* calculates the new lower bound, D_{min}, and finds that it is greater than the already achieved execution time, so it stops. Again, if we compare this result with the experiment where we had considered block-processing of designs, we see that the design in Table 8 will perform 3000 computations in 18,900 μ seconds while the current design will perform the computations in 22,410 μ seconds. So it is important to integrate both block-processing and design space exploration as part of the temporal partitioning process so that appropriate task mapping to partitions and design points is performed to produce designs that will give better performance.

In Table 10, we show the results on DCT with $R_{max} = 1024$ (XC4025 fpga). In this experiment the execution time tolerance δ is 1000 μs. To show how varying the parameter δ affects the performance of the algorithm, we reduce δ to 100 μs and repeat the same experiment whose results are shown in Table 11. The number of iterations spent looking for a solution increases, thus increasing the runtime. But a better solution is achieved. We therefore observe that reducing execution time tolerance increases the run time but

المنارة للاستشارات
www.manaraa.com

Table 10. DCT, $R_{max} = 1024, \delta = 1000\,\mu s, \gamma = 1, k = 3000$.

			Bounds		Result		
C_T (μs)	N	I	D_{max} (μs)	D_{min} (μs)	Design execution time (μs)	T (s)	T (m)
30	5	1	76,410	2,535	18,240	20.92	
		2	11,775	2,535	Inf.	300	
		3	15,816	11,775	Inf.	300	
		4	17,709	15,816	16,980	288.46	
	6	1	16,980	2,565	11,760	76.43	
		2	9,772	2,565	Inf.	300	
		3	11,574	9,772	Inf.	300	
	7	1	11,760	2,595	11,520	214.4	
		2	7,146	2,595	Inf.	300	
		3	9,483	7,146	Inf.	300	
	8	1	11,520	2,625	Inf.	300	45.00

Table 11. DCT, $R_{max} = 1024, \delta = 100\,\mu s, \gamma = 1, k = 3000$.

			Bounds		Result		
C_T (μs)	N	I	D_{max} (μs)	D_{min} (μs)	Design execution time (μs)	T(s)	T(m)
30	5	1	76,410	2,535	18,240	20.92	
		2	11,775	2,535	Inf.	300	
		3	15,816	11,775	Inf.	300	
		4	17,709	15,816	16,980	288.46	
		5	16,761	15,816	16,020	74.17	
		6	15,934	15,816	Inf.	300	
	6	1	16,020	2,565	11,760	76.43	
		2	9,772	2,565	Inf.	300	
		3	11,574	9,772	Inf.	300	
		4	10,359	11,574	10,560	104.04	
		5	10,510	11,574	Inf.	300	
	7	1	10,560	2,595	Inf.	300	
	8	1	10,560	2,625	Inf.	300	49.4

achieves better solutions. For all the experiments shown in this section, we also experimented with obtaining optimal solutions as we have shown for the AR filter. However, in *none* of these experiments could the optimal solution process get even a single feasible solution in the same run time as the iterative solution process. This is because in the iterative solution process we are dividing the solution space into smaller regions, thus reducing the size of the problem that the ILP solver has to solve in one run of execution. Also, we are directing the search process to look from higher design execution time solutions towards lower design execution time solutions and this directed search process seems to help the solver when solving problems with very large solution spaces.

We have applied this technique to various other examples like 2D-FFT and FIR filter, median filter. The results we noticed are similar and also since the their taskgraphs are very regular like DCT we have instead included results for random unstructured graphs in the next section. This shows the viability of the approach for both regular and non-regular graphs.

www.manaraa.com

8. Comparison with List Based Scheduling Algorithm

Following the two case studies we demonstrate the results of our techniques with another temporal partitioning algorithm based on the list scheduling technique. We will compare the results for the DCT example which is a very regular graph. To find how our algorithm works on unstructured graphs we also generated many random graphs. The characteristic feature in which they differ from DCT is that they are graphs with more number of tasks in their critical path i.e. they are long graphs. Also the various tasks are different in size and so vary in the number of design points.

We now discuss briefly the the scheduling algorithm which is similar to some other temporal partitioning works in literature [7]. In other partitioning works temporal partitioning is performed on an operation level data flow graph. Each operation in the data flow graph is placed in a priority list honoring the dependency among the operations. The priority list is formed by placing the nodes on the list one by one. A node is placed on the priority list if all its predecessors are already on the priority list. Then the algorithm assigns nodes starting from highest to the lowest priority in a partition until the area is filled. Once a partition is filled nodes are assigned to the next temporal partition. Each operation has one area and delay value associated with it. We will extend the same list based scheduling technique to work on task graphs instead of operation graphs. However there is no easy way to incorporate multiple design points in this technique. Therefore, going by the philosophy of this approach where the aim is to minimize the number of partitions in the design we choose the least area design point for each task prior to the start of the list based scheduling algorithm.

Table 12 presents the result of our comparison for the DCT and the random graphs. We have shown the design execution times for our iterative search algorithm and the list based scheduling algorithm. The results are presented for each partition bound for which a solution is generated by the algorithms. Since the reconfiguration overheads for both the algorithms is the same we show the design execution times without the reconfiguration overheads.

Graph Random 1 consists of 20 nodes, Random 2 has 30 nodes. Both the graphs have upto 4 design points per task. We have presented results for different area constraints and block-processing factors. The set of results on the DCT example and the random graphs demonstrate the improvement in performance of our algorithm over the list based scheduling method. The results are for varying block-processing factors. In each of the results the performance of our algorithm is superior by 7–40%. This demonstrates the following two significant points:

- Design space exploration without block-processing is meaningful because the exploration process will choose the most appropriate design points for the given constraints. The first line in Table 12 with no block processing demonstrates a performance

Table 12. Comparison with list based scheduling algorithm.

Exp.	R_{max} (CLBs)	C_T (μs)	k	N	Design execution time: List based	Design execution time: Iterative	% Improv
DCT	1024	30	1	5	7,200 ns	4,610 ns	35.97
			3,000	5	21,450 μs	16,680 μs	22.23
			3,000	6	–	11,400 μs	46.85
			3,000	7	–	11,110 μs	48.20
Random 1	1024	30	1	4	9,000 ns	5,100 ns	43.3
			3,000	4	27,000 μs	14,850 μs	45
Random 2	1024	30	1	8	13,500 ns	10,950 ns	18.88
			3,000	8	40,260 μs	27,450 μs	31.8
Random 2	2034	30	1	2	6,450 ns	4,290 ns	33.48
			3,000	2	19,350 μs	13,500 μs	30.23
				3	–	12,600 μs	34.88

المنارة للاستشارات

www.manaraa.com

improvement of 35% over the results from an algorithm that chooses the design points prior to the temporal partitioning step.
- Design space exploration with block processing demonstrates that the amortization of the reconfiguration overhead due to block processing will help in the usage of more temporal partitions. For DCT the result demonstrates an up to 47% improvement over the list based algorithm that does not consider block processing.

9. Extensions and Limitations of the Work

All our methodology is still applicable in case of inter-loop dependencies by simply setting the block processing factor to '1' (i.e., no block processing). However, in the presence of inter-loop dependencies (i.e., absence of block processing) temporal partitioning is generally not time-effective for the devices like XC4000 that have high reconfiguration time. However, for devices such as XC6200 and the context switching FPGAs, where reconfiguration time is relatively low, temporal partitioning remains viable and useful. In either case our formulation will produce an optimal or near-optimal temporal partitioning solution after performing design space exploration and choosing the most appropriate design point for each task. (This solution may have only one temporal segment for architectures with high reconfiguration overheads.) We now present some of the extensions of our work and the limitations of the current technique.

9.1. *Intermediate Data Transfer Time*

In the design process model in Section 5, we have assumed that a suitable high level synthesis system exists that can schedule memory accesses together with the operations in the task graph if there is enough slack available. However if such a synthesis system is not available and the memory accesses have to be performed prior to the execution of the task graph we need to account for the memory read write access times in our model. In the model presented earlier we have not integrated the calculation of read and write times for intermediate data in the calculation of the delay of each temporal partition. However, our model is very extensible in this respect. Since we are already calculating the amount of data transfer taking place across each temporal partition, the current model can be extended by modifying the minimization goal of the ILP model to include the intermediate memory read-write times. To do this we extend the minimization goal to also include

$$\text{Amount of data transfer} \\ * (\text{Memory read time} + \text{Memory write time}).$$

We can exclude the memory read by the input tasks and memory written to by the output tasks from the minimization goal as this factor is a constant of the graph and cannot be reduced.

In terms of the equations presented in Section 6.2.1, we already have a variable $w_{pt_1t_2}$ defined that is representative of whether data is being transferred across temporal partition p due to tasks t_1 and t_2. To calculate the read and write times for the intermediate data we only need to know, if a data transfer is taking place but are not concerned about the partition boundaries it is taking place. Therefore, we can generate a new variable $i_{t_1t_2}$ that represents the data transfer due to tasks t_1 and t_2 without considering the partition where this transfer takes place. This can be done in terms of the $w_{pt_1t_2}$ variables already generated. Formally, we generate the variable $i_{t_1t_2}$ below.

$$i_{t_1t_2} = \begin{cases} 1 & \text{if task } t_1 \text{ and } t_2 \text{ are not placed in the} \\ & \quad \text{same temporal partition} \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

$$\forall p, 1 \leq p \leq N, \forall t_2 \in T, \forall t_1 \rightarrow t_2 : i_{t_1t_2} \geq w_{pt_1t_2}$$

Then the time required for the data transfer of intermediate data is equal to

$$i_{t_1t_2} * B(t_1, t_2) * k * D_{mem}$$

where, D_{mem} *is the sum of the read and write time for one memory element of the reconfigurable processor.*

Now the minimization goal for the new problem will be

$$Minimize\ :\ \eta * C_T + \sum_{p=1}^{N} d_p \\ + i_{t_1t_2} * B(t_1, t_2) * k * D_{mem} \tag{18}$$

To extend the technique presented in Section 7 we need to include the intermediate data read and write time in the generation of the delay bounds D_{max} and

www.manaraa.com

D_{min} used in that algorithm. In the preprocessing step where we generate D_{max} and D_{min} we can also generate upper and lower bounds on the amount of data transfer that can take place for the given task graph. The upper bound on the intermediate data transfer is given by the sum of all data transfers that can ever take place in the task graph. This is available by summing all data transfers across all edges in the task graph. This value multiplied by D_{mem} is the upper bound on the time to transfer the intermediate data for the task graph. The upper bound on execution delay of design (as calculated in Section 7) + upper bound on the intermediate data transfer time will be the new D_{max}. The lower bound for the data transfer is 0, so D_{min} will remain the same as calculated in Section 7.

With the above extension, the intermediate data transfer time will be incorporated in the algorithm. The effect on the solution will be twofold. If the memory read/write time for tasks is very small compared to the execution times of the tasks then results similar to our experimental results will still be generated. However, if the memory read/write times are of comparable magnitude, then we will see solutions that have a tendency to avoid cutting across intertask edges. All the intermediate data-transfer time in Eq. (18) is added to the design execution time. However, in practice, part of this cost can be reduced in the following ways:

- It is not necessary to have a design where all the intermediate data is read and written completely excluded to the execution of the design. Much of this read/write can be performed in parallel to the execution of the rest of the design. We can also develop an estimation process that calculates the overhead of intermediate data transfer for each design point, if such a transfer were to take place because of a task being placed in the next temporal partition. This can be incorporated in our model and accurate execution time results will be generated. This estimation process and model is currently being investigated.
- Or, if the read and write have to be performed in serial to the design execution, we have developed a model that will reduce the time to access memory by generating two separate clocks—one for memory access and one for design execution. Figure 17 presents an overview of this approach. If a single clocking scheme is used for the FPGA the clock width is limited by the maximum combinational delay among all the functional units in the design. Usually the memory access can take place at a much faster rate than the clock frequency dictated by the design. Therefore we have split the memory access and the design execution so that memory access can occur at a faster rate. The time to program the clock from the host is usually negligible as it involves writing a single word to the reconfigurable processor.

We can extend Eq. (18) by multiplying the data-transfer time with a constant 'reduction factor' between 0 and 1

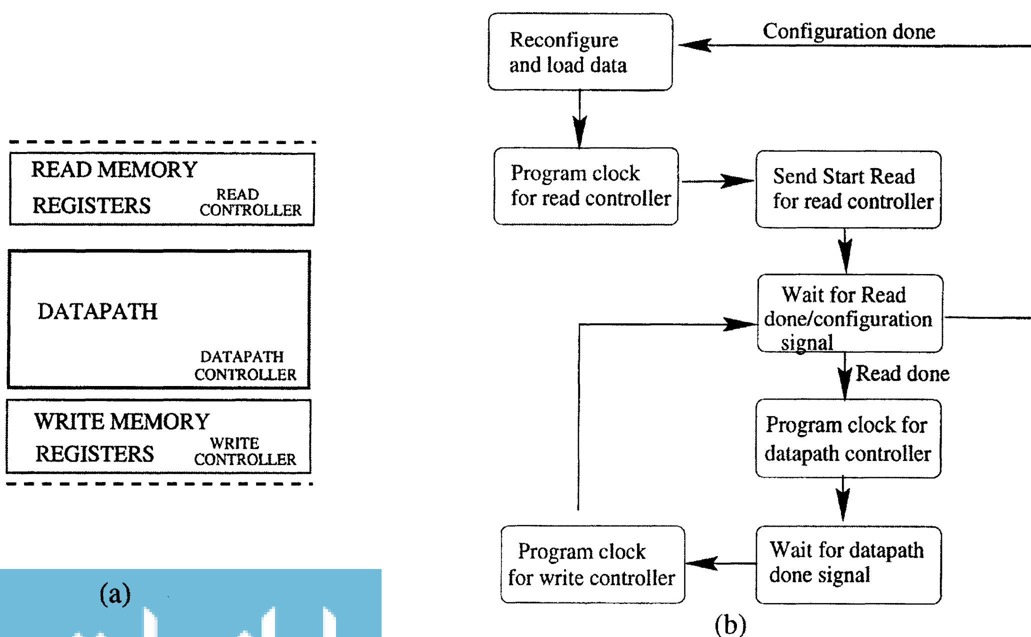


Figure 17. Reduction of time for memory access.

www.manaraa.com

Table 13. Results for variation of the factor reducing memory access time.

Exp.	R_{max} (CLBs)	Reduction factor	N	Design execution time (μs)
DCT	2304	0	2	4,830
$D_{mem} = 140$ ns			3	3,735
		.2	2	5,408
			3	5,063
		.4	2	6,885
			3	6,423
		.6	2	7,860
AR filter	196	0	4	5,355
$D_{mem} = 140$ ns			5	5,010
		.2	4	5,940
			5	5,845
		.4	4	6,531

that can be used to appropriately scale down the memory access time by the average amount that it is being reduced by using one of the techniques discussed above. If the factor is 0 then all the memory access costs have been absorbed. If the factor is 1 then none of the costs have been absorbed. We now present a few experimental results in Table 13 for different values of this reduction factor for the DCT and AR filter examples. For all the experiments $C_T = 30$ μs and $k = 3{,}0004$. We see from the tables that for the DCT temporal partitioning will be explored till the reduction factor is .4. For the reduction factor at .6 the the design with minimum number of partitions is the best solution. For AR filter the reduction factor of $\geq .4$ stops the exploration process.

9.2. *Intermediate Data Overhead*

Intuitively we can understand that due to block processing the amount of memory required for saving the intermediate data will be k times the amount of memory required for a temporally partitioned solution that does no block processing. This would happen if a solution generated for $k = 1$ (no block processing) is used to process blocks of data. However, it is not necessary that the solution generated by our algorithm for both block processing and non block processing in a design will gave same results. Formally, we can state the overhead of intermediate memory needed for block processing in each temporal partition in terms of the variables of the ILP model. The total amount of memory overhead in each partition is given by

$$\sum_{t \in T} \sum_{p \leq p_2 \leq N} \sum_{m \in M_t} y_{tp_2m} * B(env, t) * k$$
$$+ \sum_{t \in T} \sum_{1 \leq p_3 \leq p} \sum_{m \in M_t} y_{tp_3m} * B(t, env) * k$$
$$+ \sum_{t_2 \in T} \sum_{t_1 \rightarrow t_2} (w_{pt_1t_2} * B(t_1, t_2) * k)$$

The maximum of these values for all partitions would determine the size of the external RAM required for the system.

If we run two versions of a specification through our system, with and without block processing, we can determine the overhead due to block processing. As a byproduct of our model we can thus calculate precisely the memory overhead due to block processing in each design.

9.3. *Limitations*

As we have discussed earlier in Section 8, our techniques demonstrate that design space exploration with block processing is beneficial in amortizing the cost of the reconfiguration overhead. However, if data is to be processed in real time where blocks of data are not available a priori, our method can still be used to search for a temporally partitioned solution if one is possible within the inter block-arrival time constraint. It is possible for our system to take the inter block-arrival time constraint on the overall execution time rather than have one generated by the tool. If a static/temporally-partitioned solution is possible it will be generated by our tool. However, if the designer cannot specify an inter block-arrival time or if this time varies for various inputs and cannot be know a priori then our methods cannot be applied.

The current implementation does not support pipelining of the different computations in the same temporal partition. This would be particularly beneficial as it would reduce the execution time for the designs. Another limitation of the approach is that even though tasks can be of arbitrary granularity, splitting of tasks across temporal partitions is not allowed. Currently the memory read/written in a temporal partition remains alive for the life of a temporal partition. More detailed memory access models would require sophisticated foot-print analysis of the memory-bound data structures and is beyond the scope of the current work. The partial RTR capabilities of the reconfigurable device is also not exploited from within the algorithm.

www.manaraa.com

10. Conclusion

We presented an automated temporal partitioning methodology, which demonstrates how integrating design space exploration and block-processing procedures, can lead to performance enhancements in dynamically reconfigured designs even when the reconfiguration overhead is a dominating factor in the computation time. We have shown, that by using mathematical programming techniques we can model the task level temporal partitioning and design exploration problem incorporating multiple constraints of area, design execution time, and memory. We have also developed a framework in which these techniques can be used in a novel manner to solve constraint satisfaction problems for large specifications of real world examples such as the DCT. We are able to get near-optimal solutions in short run times with this iterative procedure. The effectiveness of the formulations and iterative procedure was demonstrated by the case study of the DCT.

This technique can handle tasks of arbitrary granularity, so the same technique can be used to handle task graphs with task sizes varying from small to very large. It is also possible to address sharing of resources in a temporal partition though the problem size and complexity will be increased as more variables will be added to the ILP model to model sharing of resources.

The algorithms presented here have been implemented within the temporal partitioning module of the SPARCS [15] integrated design environment.

Acknowledgment

This work is supported in part by the US Air Force, Wright Laboratory, WPAFB, under contract number F33615-97-C-1043. The authors would like to thank the reviewers for their suggested improvements to the manuscript. The authors would also like to thank Al Scarpelli, Kerry Hill, Darrell Barker and Dr. John Hines from WPAFB for their guidance to the work; Sriram Govindarajan, Iyad Ouaiss, Vinoo Srinivasan and other members of the SPARCS team for invaluable support and comments.

References

1. Xilinx Inc., http://www.xilinx.com. "*Data Book and Application Notes.*"
2. Altera Inc., http://www.altera.com. "*Data Book and Application Notes.*"
3. Atmel Inc., http://www.atmel.com. "*Data Book and Application Notes.*"
4. B.L. Hutchings and M.J. Wirthlin, "Implementation Approaches for Reconfigurable Logic Applications," *International Workshop on Field-Programmable Logic and Applications, FPL*, Springer, 1995, pp. 419–428.
5. M. Dorfel and R. Hofmann, "A Prototyping System for High Performance Communication Systems," *IEEE Workshop on Rapid System Prototyping, RSP*, IEEE Computer Society Press, 1998, pp. 84–88.
6. J. Babb, R. Tessier, M. Dahl, S. Hanono, D. Hoki, and A. Agarwal, "Logic Emulation with Virtual Wires," *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, vol. 16, no. 6, 1997, pp. 609–626.
7. M. Vasilko and D. Ait-Boudaoud, "Architectural Synthesis for Dynamically Reconfigurable Logic," *International Workshop on Field-Programmable Logic and Applications, FPL*, Springer, 1996, pp. 290–296.
8. K.M. GajjalaPurna and D. Bhatia, "Temporal Partitioning and Scheduling for Reconfigurable Computing," *FPGAs for Custom Computing Machines, FCCM*, IEEE Computer Society Press, 1998, pp. 329–330.
9. J. Spillane and H. Owen, "Temporal Partitioning for Partially-Reconfigurable-Field-Programmable Gate," *Reconfigurable Architectures Workshop, RAW in IPPS/SPDP*, Springer, 1998, pp. 37–42.
10. M. Kaul and R. Vemuri, "Optimal Temporal Partitioning and Synthesis for Reconfigurable Architectures," *Design, Automation and Test in Europe, DATE*, IEEE Computer Society Press, 1998, pp. 389–396.
11. S. Trimberger, "Scheduling Designs into a Time-Multiplexed FPGA," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA*, ACM Press, 1998, pp. 153–160.
12. M. Kaul, R. Vemuri, S. Govindarajan, and I. Ouaiss, "An Automated Temporal Partitioning and Loop Fission Approach for FPGA Based Reconfigurable Synthesis of DSP Applications," *Design Automation Conference, DAC*, IEEE Computer Society Press, 1999, pp. 616–622.
13. D.S. Rao and F. Kurdahi, "Hierarchical Design Space Exploration for a Class of Digital Systems," *IEEE transactions on Very Large Scale Integration (VLSI) Systems*, vol. 1, no. 3, 1993, pp. 282–294.
14. G.D. Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.
15. I. Ouaiss, S. Govindarajan, V. Srinivasan, M. Kaul, and R. Vemuri, "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures," *Reconfigurable Architectures Workshop, RAW in IPPS/SPDP*, Springer, 1998, pp. 31–36.
16. M. Xu and F. Kurdahi, "Layout Driven High Level Synthesis for FPGA Based Architectures," *Design Automation and Test in Europe, DATE*, IEEE Computer Society Press, 1998, pp. 446–450.
17. M. Wolf, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishers, 1996.
18. S.Y. Kung, *VLSI Array Processors*, Prentice Hall, 1988.
19. K. Roy-Neogi and C. Sechen, "Multiple FPGA partitioning with performance optimization," *ACM/SIGDA International*

المنارة للاستشارات
www.manaraa.com

Symposium on Field Programmable Gate Arrays, FPGA, ACM Press, 1995, pp. 146–152.
20. P. Chan, M. Schlag, and J. Zien, "Spectral-Based Multi-Way FPGA Partitioning," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA*, ACM Press, 1995, pp. 133–139.
21. W. Fang and A. Wu, "A Hierarchical Functional Structuring and Partitioning Approach for Multiple-FPGA Implementations," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 10, 1997, pp. 1188–1195.
22. H. Schmit, L. Arnstein, D. Thomas, and E. Lagnese, "Behavioral Synthesis for FPGA-Based Computing," *FPGAs for Custom Computing Machines, FCCM*, IEEE Computer Society Press, 1994, pp. 125–132.
23. R.D. Hudson, D.I. Lehn, and P.M. Athanas, "A Run-Time Reconfigurable Engine for Image Interpolation," *FPGAs for Custom Computing Machines, FCCM*, IEEE Computer Society Press, 1998, pp. 88–95.
24. M.J. Wirthlin and B.L. Hutchings, "Sequencing Run-Time Reconfigured Hardware with Software," *ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA*, ACM Press, 1996, pp. 122–128.
25. M. Gokhale and J.M. Stone, "NAPA C: Compiling for Hybrid RISC/FPGA Architectures," *FPGAs for Custom Computing Machines, FCCM*, IEEE Computer Society Press, 1998, pp. 126–135.
26. W. Luk, N. Shirazi, and P. Cheung, "Automating Production of Run-Time Reconfigurable Designs," *FPGAs for Custom Computing Machines, FCCM*, IEEE Computer Society Press, 1998, pp. 147–156.
27. M. Chu, N. Weaver, K. Sulimma, A. DeHon, and J. Wawrzynek, "Object Oriented Circuit-Generators in Java," *FPGAs for Custom Computing Machines, FCCM*, IEEE Computer Society Press, 1998, pp. 158–166.
28. M. Kaul and R. Vemuri, "Temporal Partitioning Combined with Design Space Exploration for Latency Minimization of Run-Time Reconfigured Designs," *Design Automation and Test in Europe, DATE*, IEEE Computer Society Press, 1999, pp. 202–209.
29. M.J. Wirthlin and B.L. Hutchings, "A Dynamic Instruction Set Computer," *FPGAs for Custom Computing Machines, FCCM*, IEEE Computer Society Press, 1995, pp. 99–106.
30. A. Kalavade, "System-Level Codesign of Mixed Hardware-Software Systems," Ph.D. Thesis, University of California, Berkeley, 1995.
31. C.H. Gebotys, "Optimal Synthesis of Multichip Architectures," *IEEE ICCAD*, IEEE Computer Society Press, 1992, pp. 238–241.
32. R. Niemann and P. Marwedel, "An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming," *European Design and Test Conference, ED&TC*, 1996, pp. 473–479.
33. C.H. Gebotys and M.I. Elmasry, *Optimal VLSI Architectural Synthesis*, Kluwer Academic Publishers, 1992.
34. P. Hansen, B. Jaumard, and V. Mathon, "Constrained Nonlinear 0-1 Programming," *ORSA Journal of Computing*, vol. 5, no. 2, 1993, pp. 97–119.
35. F. Glover and E. Woolsey, "Converting the 0-1 Polynomial Programming Problem to a 0-1 Linear Program," *Operations Research*, vol. 21, no. 1, 1974, pp. 156–161.
36. G.K. Wallace, "The JPEG Still Picture Compression Standard," *ACM Communications*, 1991.
37. S. Trimberger, D. Carberry, A. Johnson, and J. Wong, "A Time-Multiplexed FPGA," *FPGAs for Custom Computing Machines, FCCM*, IEEE Computer Society Press, 1997, pp. 22–28.
38. S.M. Scalera and J.R. Vazquez,"The Design and Implementation of a Context Switching FPGA," *FPGAs for Custom Computing Machines, FCCM*, IEEE Computer Society Press, 1998, pp. 78–85.
39. Annapolis Micro Systems, Inc., *WILDFORCE Reference Manual, Document #1189—Release Notes*.
40. Y. Hung and A.C. Parker, "High-Level Synthesis with Pin Constraints for Multiple-Chip Designs," *Design Automation Conference, DAC*, IEEE Computer Society Press, 1992, pp. 231–234.

Meenakshi Kaul received B.Tech. in Computer Science from Lucknow University, India in 1992 and M.Tech. in Computer Science from Indian Institute of Technology, Madras in 1994. She is presently working towards a doctoral degree in Computer Engineering at the University of Cincinnati. Her current research interests include design automation tools, synthesis for reconfigurable systems, high level synthesis and mathematical modeling techniques.
mkaul@ececs.uc.edu

Ranga Vemuri, an associate professor of electrical and computer engineering at the University of Cincinnati, also directs its Laboratory for Digital Design Environments. His interests include the computer-aided design of digital systems, formal verification, system synthesis, performance modeling, hardware description languages, and parallel algorithms. Vemuri received the M.Tech. degree from the Indian Institute of Technology, Kharagpur, and the Ph.D. from Case Western Reserve University, both in computer engineering. He received the Siddhartha gold medal, a distinguished research award, and an outstanding teacher award. He is a member of the IEEE Computer Society, IEEE Circuits and Systems Society, ACM SIGDA, American Society of Electronic Engineers, and Eta Kappa Nu.
ranga.vemuri@uc.edu

www.manaraa.com
المنارة للاستشارات

Journal of VLSI Signal Processing Systems 24, 211–221, 2000.
© 2000 Kluwer Academic Publishers. Printed in The Netherlands.

Application of Reconfigurable CORDIC Architectures

OSKAR MENCER, LUC SÉMÉRIA AND MARTIN MORF
Computer Systems Laboratory, Department of Electrical Engineering, Stanford, CA 94305, USA

JEAN-MARC DELOSME
Département Informatique, Université d'Evry, Cours Monseigneur Romero, 91025 Evry, France

Abstract. Reconfiguration enables the adaption of Coordinate Rotation DIgital Computer (CORDIC) units to the specific needs of sets of applications, hence creating application specific CORDIC-style implementations. Reconfiguration can be implemented at a high level, taking the entire CORDIC unit as a basic cell (CORDIC-cells) implemented in VLSI, or at a low level such as Field-Programmable Gate Arrays (FPGAs). We suggest a design methodology and analyze area/time results for coarse (VLSI) and fine-grain (FPGA) reconfigurable CORDIC units. For FPGAs we implement CORDIC units in Verilog HDL and our object-oriented design environment, PAM-Blox. For CORDIC-cells, multiple reconfigurable CORDIC modules are synthesized with state-of-the-art CAD tools. At the algorithm level we present a case study combining multiple CORDICs based on a geometrical interpretation of a normalized ladder algorithm for adaptive filtering to reduce latency and area of a fully pipelined CORDIC implementation. Ultimately, the goal is to create automatic tools to map applications directly to reconfigurable high-level arithmetic units such as CORDICs.

I. Introduction

Reconfigurable computing spans the space between programmable microprocessors and static Application Specific Integrated Circuits (ASICs). Reconfigurable architectures offer the flexibility of ASIC design and the programmability of microprocessors. The overhead of reconfigurability depends on the complexity of the reconfigurable cell. Bit-level cells in Field-Programmable Gate Arrays (FPGAs) offer high flexibility with a high overhead in latency and area. ASICs with programmable arithmetic units, "chunky architectures" [1], in our case Coordinate Rotation DIgital Computer (CORDIC)-cells, have a lower overhead with much less flexibility. In this research we are not going to end the debate about which level of reconfigurability is best. Instead, we show how CORDIC arithmetic units can be implemented on FPGAs and on ASICs. Given multiple CORDIC arithmetic units, we show a case study on how applications could be mapped onto a set of CORDIC arithmetic units by using a geometric interpretation of computation.

CORDIC arithmetic units use shift-and-add primitives to compute fixed-point elementary functions on relatively small silicon area. For a more detailed introduction to CORDIC algorithms see [2]. For advanced CORDIC techniques see for example [3, 4]. CORDIC units are known to be highly pipelineable, very small, with linear convergence towards the correct result. Linear convergence means that we can guarantee at least one bit of precision per shift-add iteration. The internal structure of CORDICs, consisting of adders and wired shifts in the case of parallel CORDICs, makes them well suited for FPGA implementation [5].

CORDIC functional units compute up to two elementary functions at the same time. Given three arguments x, y, z, basic CORDIC arithmetic units compute function pairs such as shown in Table 1.

The fundamental principles behind the CORDIC algorithms of Volder [6] and Walther [7] can be found in their scalar form in the work of Chen [8]. Ahmed showed in [4] how scale factor compensation can be avoided by choosing an appropriate shift-sequence to automatically compensate for the scale factor. A

المنارة للاستشارات
www.manaraa.c

Table 1. Functions computed by CORDIC.

$x \cdot \cos(z) - y \cdot \sin(z)$	$y \cdot \cos(z) + x \cdot \sin(z)$
x	$y + x \cdot z$
$x \cdot \cosh(z) + y \cdot \sinh(z)$	$y \cdot \cosh(z) + x \cdot \sinh(z)$
$\sqrt{x^2 + y^2}$	$z + \tan^{-1}(y/x)$
x	$z + (y/x)$
$\sqrt{x^2 - y^2}$	$z + \tanh^{-1}(y/x)$

refinement of this idea, in order to minimize overhead, was presented in [9]. All CORDIC implementations in this paper are therefore correctly scaled with minimal overhead.

Ahmed also showed in [4] that if Chen's convergence computation technique is applied to complex numbers instead of real numbers (as assumed by Chen) one obtains the class of CORDIC algorithms. The method of formally "replacing" real by complex numbers was extended in [3, 10] to obtain CORDIC algorithms for quaternions and pseudo-quaternions. When the CORDIC functions, especially the higher order functions, are matched to applications—a system design issue—the real power of CORDICs and related algorithms can be exploited.

One alternative to CORDICs are multiplication based algorithms. The major drawback of fast multipliers is their large size and irregularity of wiring for a logarithmic reduction of terms [11]. CORDICs compute two elementary functions on approximately the area and latency of 1–2 multipliers.

Reconfiguration of CORDICs enables effective hardware support of such complex functions, similar to micro-code or firmware (library functions). It becomes possible to hide the complexity involved from a typical application-level programmer. Custom design of CORDIC units for individual applications is a complex task, requiring both specialized low-level design tools and symbolic computing tools that support a domain expert. Sophisticated tools that can support a typical programmer will eventually become available. In the mean-time domain experts will have to use today's tools to create winning designs using these ideas in advanced applications.

As a first step towards an automatic CORDIC compiler for FPGAs, we introduce the hardware object as an intermediate level of abstraction [12]. We define a set of CORDIC module generators that could be targeted by a compiler similar to the instruction set of a microprocessor. Most of todays efforts at direct compilation from a high-level language to FPGAs target very simple arithmetic units such as adders, multipliers, shifters, etc. Generally, by targeting such simple modules, most of the power of reconfigurable computing is lost. Instead, more complex arithmetic units such as CORDICs coupled with various alternatives of number representations should be targeted by higher-level compilers to exploit the full potential of reconfigurable computing. We are at the beginning of the development process of special purpose compilers for complex arithmetic units such as CORDICs. One objective of this paper is to show a possible direction for high-level compilation to CORDICs.

Section 2 describes the methodology of this research. Section 3 presents the results at the module level for a "chunky" CORDIC architecture, and CORDICs on FPGAs. Section 4 presents a case study at the algorithm level: mapping an adaptive ladder filter to *fixed-point* CORDIC arithmetic units.

II. Methodology

Ultimately, the goal is to create automatic algorithms to map applications directly to reconfigurable high-level arithmetic units. As a starting point we split the problem into 2 parts:

- **module-generation level (Section III):** on this level we create building blocks based on CORDIC arithmetic units. Module generation implies that we can create application-specific CORDIC units given parameters such as data bit-width, precision, or number of stages. We consider two options at the module level:
  1. fine-grain reconfiguration: CORDICs on FPGAs
  2. coarse-grain reconfiguration: VLSI CORDIC-cells
- **algorithm level (Section IV):** we use the CORDIC modules generated at the lower level, and combine them to compute entire applications.

Module-generation is well understood, based on past research on CORDIC arithmetic units. We implement CORDICs for FPGAs with our module-generation environment, PAM-Blox, described in the next section, and Synopsys FPGA Express. Coarse-grain CORDIC cells are synthesized with Synopsys Behavioral Compiler.

المنارة للاستشارات
www.manaraa.com

The algorithm level poses similar challenges as compilation to a very complex instruction set. The process of combining complex building blocks to optimally compute a specific algorithm is still not very well understood. We propose a geometrical interpretation of computation to create a unified approach for combining multiple CORDICs given a specific algorithm.

III. Module Level

At the module level, our task is to map a CORDIC architecture to gates or look-up tables. In this section, we look at two approaches. The first approach is a fine-grain reconfiguration on FPGAs. The CORDIC modules are implemented using the PAM-Blox environment or commercial synthesis tools for FPGAs. The second approach is a coarse-grain reconfiguration in which the CORDIC module itself represents the basic reconfigurable cell implemented on an ASIC. Synthesis tools map CORDICs efficiently to hardware. By changing the constraints on the latency of the design, different implementations of the cell can be explored.

We implement fully parallel CORDIC modules. Figure 1 shows the parallel architecture of a generic CORDIC unit.

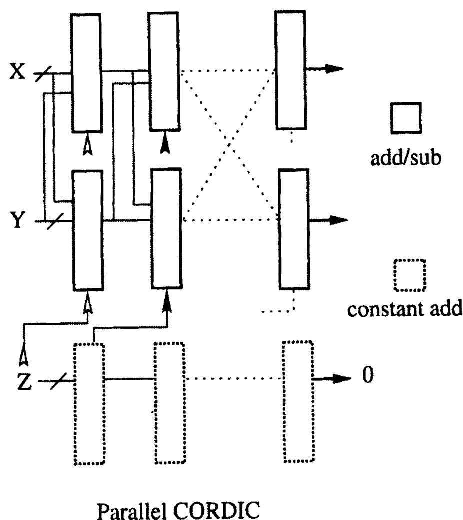


Figure 1. The figure shows a block diagram of a parallel CORDIC architecture. The table for the z-pipe is coded implicitly in the constant adders. All adders include a wired shift of the operands. The shift amount is chosen to eliminate the scaling factor. For a bit-serial CORDIC simply replace parallel adders with bit-serial adders, add delay elements and a table for z-values.

A. Fine-Grain Reconfiguration: FPGAs

We compare the implementation of CORDICs with PAM-Blox and a state-of-the-art synthesis tool for FPGAs.

A.1. PAM-Blox: Object-Oriented Module Generation. Traditional hardware synthesis is based on a top-down approach; starting from a high-level description, CAD tools synthesize and optimize the hardware level by level, until the final layout. Initial FPGA synthesis tools have taken the same approach, adding a last step of technology mapping at the end of the CAD hierarchy.

We propose a bottom-up approach to the design of synthesis tools/compilers for FPGAs. The main reason behind building FPGA circuits bottom up, is that the architecture and interconnect is limited to the resources on the FPGA, making the traditional top-down approach less optimal.

By creating a parameterizable repository of module generators, PAM-Blox [12], we add a level of abstraction that preserves optimal area and performance while simplifying the design process (compared to state-of-the-art high-performance FPGA tools). In terms of reconfigurable computing, these modules constitute the instruction set that could be targeted by the compiler.

Figure 2 shows an overview of the PAM-Blox system. We use PAM-Blox as the name for the entire design environment. PamBlox (see Fig. 2) stands for templates of hardware objects while the more complex

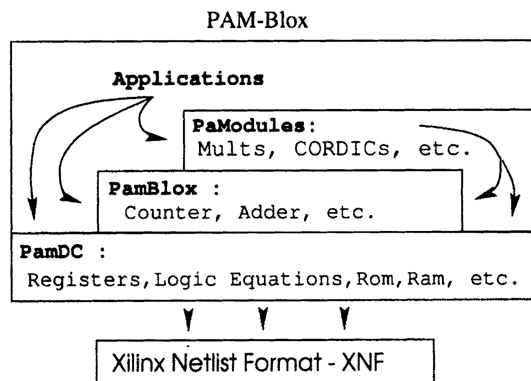


Figure 2. Layers of the PAM-Blox design environment: DIGITAL PamDC compiles the design to the Xilinx Netlist Format XNF; Pam-Blox are interacting with PamDC objects, PaModules interact with PamBlox and PamDC, and the application can access features from all three layers below.

www.manaraa.com

PaModules are objects with a fixed size. PAM-Blox simplifies the design of datapaths for FPGAs by implementing an object-oriented hierarchy described in C++. With PAM-Blox, hardware designers can benefit from some of the advantages of object-oriented system design that the software industry has learned to cherish during the last decade. Efficient use of function overloading, virtual functions, and templates makes PAM-Blox a competitive and yet simple to use design environment.

A major question is which modules are required. This question is actually similar to defining a hardware-software interface for hardware-software co-design. We believe that by providing higher-level modules such as specialized multipliers (see [13] for IDEA encryption), state-machines (see [14] for boolean satisfiability), arithmetic units for advanced number representations, etc., we can explore the benefits of reconfigurable arithmetic. In this study, we implement CORDICs and study how higher-level compilers might target such modules.

Currently, the PAM-Blox CORDICs are implemented as PaModules with a fixed bitwidth. A floorplan for a parallel CORDIC is shown in Fig. 3. The 8-bit parallel CORDIC requires 131 CLBs while a bit-serial CORDIC, with 23 bit-serial adders requires substantially more area, due to the inherent dependency structure of the CORDIC algorithm. In contrast, a CORDIC iterating with only 3 parallel ADD/SUB modules on the CORDIC equations would have very low throughput, and an area penalty for the z look-up table which is hardwired in the parallel case.

Although serial arithmetic usually takes less area, the bit-serial CORDIC occupies 30% more area than the parallel CORDIC. This counter-intuitive result is due to dependencies between the stages. A stage needs to know the sign of z of the previous stage in order to select the sign for its own computation. The resulting overhead of storing the intermediate values while waiting for the sign to compute and the increased overhead for control logic, making the bit-serial CORDIC a less desirable CORDIC solution.

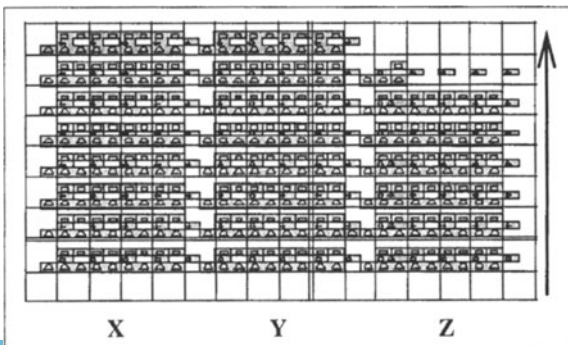


Figure 3. Layout of the PP-CORDIC, placed with PAM-Blox.

Table 2. The table shows area and cycle time for PAM-Blox and synopsys FPGA Express II (FE-II) for Xilinx XC4000 FPGAs at 0.5 μ technology (speedgrade-3), after Xilinx place-and-route. The area of the parallel CORDIC (PP-CORDIC) is given in configurable logic blocks, CLBs (cells).

CORDIC on FPGAs		
PP-CORDIC	Cycle time	Area
FE-II	25.4 ns	133 CLBs
PAM-Blox	23.7 ns	131 CLBs

The parallel CORDIC achieves a throughput of 33 million rotations per second at 33 MHz PCI clock speed. The results are summarized in Table 2. With current FPGA technology the throughput would scale up easily to 100 MHz, hence 100 million rotations per second.

A.2. Synthesis for FPGA. We compare PAM-Blox module-generation to Synopsys FPGA Express synthesis. We try to optimize a CORDIC architecture for Xilinx XC4000 FPGAs using Synopsys FPGA Express [15]. The results after optimization are comparable to the PAM-Blox results found in the previous section: after place&route the area of the circuit is 133 CLBs with a clock cycle latency of 25.4 ns.

As we will see in the next section, synthesis tools can be used to effectively optimize CORDIC modules for ASICs. However, for FPGAs, the possible optimizations are restricted by the internal architecture of the CLBs—especially the fast carry-chains. The advantage of FPGA synthesis over PAM-Blox, a structural bottom-up approach, for complex arithmetic units is therefore limited. As a consequence at the module level it is preferable to use module generation to create CORDIC units for FPGAs, and a compiler to optimize the application-level structure using reconfigurable CORDICs as elementary building blocks.

B. Coarse-grain Reconfiguration: ASICs

In this section we analyze how much a CORDIC unit can be optimized by state-of-the-art VLSI synthesis.

المنارة للاستشارات
www.manaraa.com

For the implementation of a CORDIC arithmetic unit in hardware, many of the operators (adders, subtracters) can be optimized. In particular, optimization can be performed when one of the operands is a constant (calculation of the z factors) or when some input bits have the same value (calculation of the x and y factors after shifting). We apply logic and architectural optimization for a non-pipelined version of the parallel CORDIC and synthesize the design for ASIC.

In general, the behavior of circuits can be represented by abstract models such as boolean functions and finite state machines which can be derived from higher-level models. In the case of combinational logic (i.e. circuits without feed-back), the abstract model is a set of boolean functions and relations on the circuit's inputs and outputs. These functions can be simplified for a given target architecture by employing logic synthesis and optimization [16]. Very powerful optimization can be performed under both area and/or time constraints.

For arithmetic operations, further optimization can also be performed at the architectural level by looking at different architectures of operators (e.g. ripple carry adder, carry save adder, etc.), trying to increase bit-level parallelism. In the past few years, such techniques have also been integrated within commercial tools [17] and allow quick estimation of the performance of many candidate architectures.

For ASIC synthesis we use the Synopsys Design Compiler to synthesize the circuit and the Synopsys Behavioral Compiler for the arithmetic optimization [17]. The target technology is the 'tsms 0.35 micron' logic process. We study the area/latency trade-off by changing the constraints on the optimizations. Figure 4 presents the area-time curves with and without architectural and logic optimizations. We observe that after optimization the circuit is at least 20% smaller for a given latency and at least 17% faster for a given area. The smallest design (with optimization) has a total area of 57 K library units and a latency of 41 ns, compared to an area of 75 K library units and a latency of 43 ns without optimization. Minimal latency with optimization is 17.94 ns for an area of 153 K library units. Without optimization the latency is 23.45 ns for the same area as the optimized design.

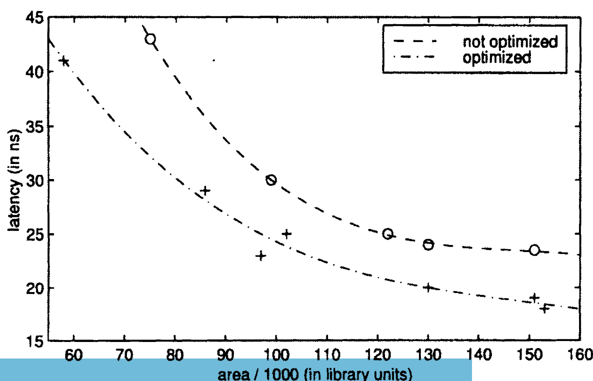


Figure 4. Area-time curves for synthesized CORDICs.

To increase the throughput for a given latency, we pipeline the implementation by inserting registers into the datapath. This can be done automatically by the synthesis tool. For any given clock frequency, an area/latency trade-off similar to Fig. 4 can also be identified with pipelined modules.

IV. Algorithm Level: Combining Multiple CORDICs

Combining multiple CORDICs to an entire application is currently more an art than a science. In order to illustrate some of the reasoning and manipulations involved when deriving CORDIC-style implementations for specific applications, we revisit an algorithm of Lee and Morf, summed up in [10] and detailed in Section 7 of the survey [18].

A. Adaptive Ladder Filter

There are many ways of developing adaptive ladder filters. A very typical case is represented by the recursive exact least-squares filters, such as discussed in [19, 20]. One of the most efficient implementations is based on adaptive ladder or lattice filters. The adaptive ladder filter is an FIR filter used for the prediction of stochastic processes, e.g. for channel equalization or speech encoding. The following development is the scalar version corresponding to a single channel filter. The corresponding multi-channel version could be derived from [21]. The filter is composed of n cascaded feed-forward stages, n being the order of the filter. Each stage has two outputs, the so-called forward and backward innovation, which are sent on to the next stage (the backward innovation being delayed by one sample period before being used). Each stage is parameterized by a "gain", the partial correlation between the forward and backward innovation. This gain varies with time and is updated whenever new values of the innovations are computed, i.e. each time there is a new sample; this is the "adaptive" part of the filter. Within each stage a time update consists of 3 equations

www.manaraa.com

(the stage and time indices are not shown here)

$$\begin{cases} \rho_+ = \rho\bar{\nu}\bar{\eta} + \nu\eta \\ \nu_+ = (\nu - \rho_+\eta)/(\bar{\rho}_+\bar{\eta}) \\ \eta_+ = (\eta - \rho_+\nu)/(\bar{\rho}_+\bar{\nu}) \end{cases} \tag{1}$$

where ρ denotes the normalized partial correlations, ν and η denote the normalized forward and backward innovations respectively, ρ_+, ν_+ and η_+ are the updated variables, and $\bar{x} = \sqrt{1 - x^2}$, the complement of x.

Usually adaptive filter implementations with Given's rotations require floating point arithmetic. The selected filter algorithm (equations above) has a built-in variance and magnitude-normalization property that allows us to use fixed-point arithmetic, which is more suitable for FPGAs. For more details on VLSI implementations and associated block diagrams of these equations see [19, 20].

B. Geometrical Interpretation of the Ladder Filter

The relations (1) are normalized versions of "Schur complement" identities relating the covariances of random variables. Since the Schur complement identities essentially capture the theorem of Pythagoras in Euclidean space, normalization, which amounts to projecting the objects from Euclidean space onto the unit sphere, yields identities of spherical geometry.

As a result the relations (1) have an elegant interpretation in terms of spherical trigonometry. Considering the triangle *FRB* in Fig. 5, and measuring both the angles R, F, B and the sides r, f, b in radians, we can write three identities from spherical trigonometry:

$$\begin{cases} \cos r = \cos R \cdot \sin f \cdot \sin b + \cos f \cdot \cos b \\ \cos F = (\cos f - \cos r \cdot \cos b)/(\sin r \cdot \sin b) \\ \cos B = (\cos b - \cos r \cdot \cos f)/(\sin r \cdot \sin f) \end{cases} \tag{2}$$

These identities enable the determination of information to the left of the dashed line (L) in Fig. 5 in terms of information to the right of (L).

With the correspondence

$$\begin{cases} \rho = \cos R, \quad \nu = \cos f, \quad \eta = \cos b, \\ \rho_+ = \cos r, \quad \nu_+ = \cos F, \quad \eta_+ = \cos B, \end{cases} \tag{3}$$

relations (1) are seen as relations providing the solution of a spherical triangle given two sides and the included angle. Such equations are found in navigation on the Earth's surface. Volder [6] developed the CORDIC procedure precisely to solve such problems digitally and

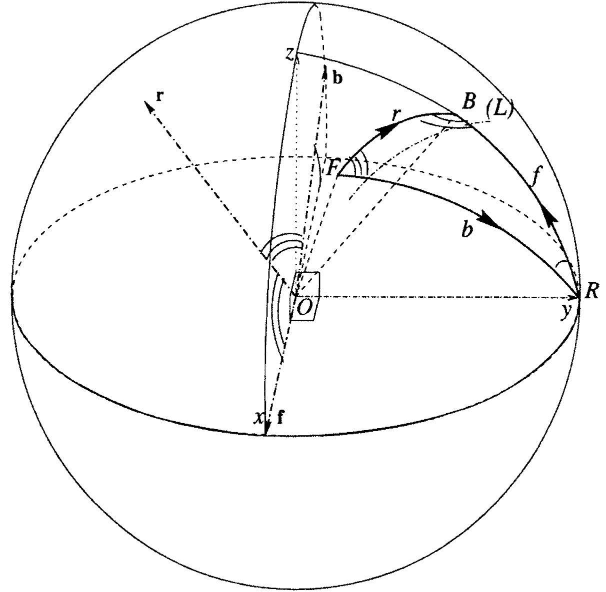


Figure 5. Geometric interpretation of the normalized ladder algorithm in terms of spherical trigonometry. From the information bRf, to the right of (L), we deduce the information FrB to the left of (L). This amounts to computing a rotation, from F to B, as the composition of two rotations, from F to R and from R to B.

showed how to link CORDIC rotations for that purpose. Following a similar vein, Lee et al. [10] proposed a way for linking the three types of CORDIC operations of Walther [7] to evaluate the expressions (1) (this way is also presented in [18], with a slight modification). Is this way optimal? Can our geometrical insight enable us to improve on it? Since the work [3] on quaternion CORDIC algorithms we know how to perform 3-D rotations in a CORDIC-like fashion by working simultaneously on all 3 components. Can this be exploited?

Geometrically, we are interested in the result of the composition of the "backward" rotation from F to R along b and the "forward" rotation from R to B along f; the cosine of the angle R between the sides b and f corresponds naturally to the normalized partial correlation. That result corresponds to the rotation from F to B along r, whose parameters are what we seek. Appendix A details this composition of 3-D rotations and actually obtains as a result a decomposition of the rotations in terms of 2-D CORDICs.

C. Implementation

Putting some flesh on the skeleton obtained in Appendix A, the computations are cast in terms of *pairs* of

المنارة للاستشارات
www.manaraa.com

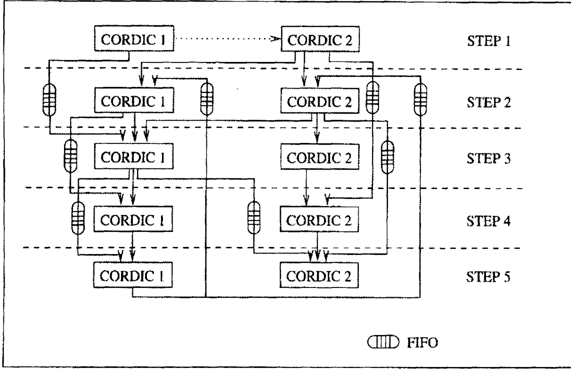


Figure 6. General dataflow of the proposed filter stage implementation.

2-D CORDIC operators (represented below by • *and* ⊙), where the operators do not require the "Z-factor" part that computes the angles explicitly. We obtain the architecture shown in Figure 6:

Step 1

- rotate $[\begin{smallmatrix}\rho\\ \bar{\rho}\end{smallmatrix}]$ to force the 2nd component to 0 and thus obtain the encoding (sign sequence) of the angle R,
- ⊙ simultaneously apply the rotation $\mathcal{R}(R) = [\begin{smallmatrix}\rho & -\bar{\rho}\\ \bar{\rho} & \rho\end{smallmatrix}]$, determined by the sign sequence for the angle R, to the vector $[\begin{smallmatrix}\bar{\eta}\\ 0\end{smallmatrix}]$ to obtain $[\begin{smallmatrix}\rho\bar{\eta}\\ \bar{\rho}\bar{\eta}\end{smallmatrix}]$.

Step 2

- apply the rotation $\mathcal{R}(f) = [\begin{smallmatrix}\nu & -\bar{\nu}\\ \bar{\nu} & \nu\end{smallmatrix}]$, determined by the encoding of f (obtained in Step 5 of previous update) to $[\begin{smallmatrix}\bar{\rho}\bar{\eta}\\ 0\end{smallmatrix}]$ to obtain $[\begin{smallmatrix}\nu\bar{\rho}\bar{\eta}\\ \bar{\nu}\bar{\rho}\bar{\eta}\end{smallmatrix}] = [\begin{smallmatrix}\nu^{\dagger}\\ \bar{\eta}\cdot\bar{\rho}\bar{\nu}\end{smallmatrix}]$.
- ⊙ apply the rotation $\mathcal{R}(f)$ to the vector $[\begin{smallmatrix}\eta\\ -\rho\bar{\eta}\end{smallmatrix}]$ to obtain $[\begin{smallmatrix}\rho+\\ \eta^{*}\end{smallmatrix}]$.

Step 3

- apply $\mathcal{R}(R)$ to $[\begin{smallmatrix}\eta^{*}\\ \nu^{\dagger}\end{smallmatrix}]$ to get $[\begin{smallmatrix}-\nu^{*}\\ \eta^{\dagger}\end{smallmatrix}]$, where $\eta^{\dagger} = \eta\cdot\bar{\rho}\bar{\nu}$,
- ⊙ employ the hyperbolic CORDIC mode and force to 0 the 2nd component of the vector $[\begin{smallmatrix}1\\ \rho+\end{smallmatrix}]$ to obtain $\bar{\rho}_{+}$ as 1st component.

Step 4

- rotate $[\begin{smallmatrix}\eta^{\dagger}\\ \bar{\eta}\cdot\bar{\rho}\bar{\nu}\end{smallmatrix}]$ to force the 2nd component to 0 and thus obtain as 1st component $\bar{\rho}\bar{\nu} = \bar{\rho}_{+}\bar{\nu}_{+}$ (and, as a byproduct, a not very accurate—when $\bar{\rho}\bar{\nu}$ is small—encoding of b, that we shall not use),
- ⊙ compute, in the linear mode, the encoding of $1/\bar{\rho}_{+}$ (non-restoring division) and, simultaneously, apply this sign sequence to $\bar{\rho}\bar{\eta}$ to get $\bar{\eta}_{+}$.

Step 5

- rotate $[\begin{smallmatrix}\nu^{*}\\ \bar{\rho}_{+}\bar{\nu}_{+}\end{smallmatrix}]$ to force the 2nd component to 0 and thus obtain the encoding (sign sequence) of the angle F to be used as encoding of the "angle" f in Step 2 for the next update.
- ⊙ apply in the linear mode the sign sequence encoding $1/\bar{\rho}_{+}$ both to ν^{*}, to get ν_{+}, and to η^{*}, to get η_{+}.

The second CORDIC at Step 4 is a modified version of the standard CORDIC architecture. It computes the sign sequence encoding of $1/\bar{\rho}_{+}$ and, simultaneously, applies this sign sequence to $\bar{\rho}\bar{\nu}$ to get $\bar{\nu}_{+}$. With $x_0 = 1$ and $y_0 = 0$, the recurrence is of the form (assuming $1/\bar{\rho}_{+}$ does not exceed 8, i.e., $|rho_{+}|$ does not exceed 0.992):

$$\begin{cases} x_{i+1} = x_i - \text{sign}(x_i)\cdot 2^{2-i}\cdot\bar{\rho}_{+} \\ y_{i+1} = y_i + \text{sign}(x_i)\cdot 2^{2-i}\cdot\bar{\rho}\bar{\nu} \end{cases} \quad (4)$$

From the recurrence relation, one can see that Eq. (4) fits on the shift-and-add resources of a CORDIC architecture.

The accuracy d needed for the computations will typically be about 16 to 20 bits. However, scaling plus additional iterations for convergence (in the hyperbolic case) impose slightly more than d pipeline stages within a CORDIC unit [4, 9].

Using the pipelined, parallel CORDIC presented before, we distinguish 4 basic architectures based on the number of CORDICs used:

1. minimal: 1–2 CORDICs
2. based on the 5 steps of computing 1 stage of the filter: $2\cdot 5$ CORDICs
3. based on the number of stages in the filter: $2\cdot n$ CORDICs
4. fully pipelineable, maximal performance: $2\cdot 5\cdot n$ CORDICs

All cases require some amount of memory, or shift-registers (FIFOs), to store intermediate values of the computation. Xilinx CLBs can be configured to 16-bit FIFOs enabling a very efficient implementation of the intermediate shift registers. Note that we do not require all three CORDIC equations, using only the

المنارة للاستشارات

www.manaraa.com

x and y pipes, we save 33% of area. Also each case has different requirements on reconfigurability on the CORDICs.

In all cases the delay for 1 result is $5 \cdot n \cdot d$ clock cycles. However, throughput differs with pipeline depth. In case 4, with $10 \cdot n$ CORDICs, throughput is 1 clock cycle between results. Case 3 with $2 \cdot n$ CORDICs results in 5 clock cycles between results. Case 2 with 10 CORDICs results in n clock cycles between results. Finally, case 1 requires $(1–2) \cdot 5 \cdot n$ clock cycles between results.

D. Discussion

In order to understand the advantages of the geometrical interpretation we compare the above implementation to an earlier implementation with CORDICs (see [10, 18]). Both implementations require 10 CORDICs. The earlier implementation employs all 3 pipelines (x, y, and z) as opposed to only (x, y) CORDICs in the proposed implementation. Thus, the geometrical interpretation gives us 17% reduction of latency with a 45% reduction of area for the fully pipelined case 4 (see above). However the earlier implementation could be modified to also use sign encodings of the z-quantities thus giving about the same latency and area. The geometrical approach has the advantage of being more systematic, less empirical, and therefore more apt to be used to create compilers for reconfigurable computing that can target CORDIC arithmetic units. While being a good starting point, our geometric viewpoint has probably not been fully exploited here and we still have hope for a more parallel computational scheme, operating on 3-D vectors.

An alternative way of using the geometrical insight could be derived from the general update equation (32 in [21]) which is not only valid in the scalar case but also in the multi-channel case. The idea in [21] is based on a block diagonalization of the singular-value decomposition type. This could be done for instance using an extension of the CORDIC idea to quaternion representation as in [3].

V. Conclusions

We have implemented high-throughput CORDICs for reconfigurable computing in our object-oriented hardware design environment—PAM-Blox—and optimized generic parallel CORDICs with state-of-the-art synthesis tools.

While commercial synthesis tools are very efficient in optimizing CORDICs for ASICs, FPGAs do not seem to lend themselves to these types of optimizations. At a higher level, in order to give an idea of what is involved in the automatic generation of CORDIC-like units for specific applications, we have decomposed the computations of an adaptive filter in terms of CORDIC operations, using geometric insights that could lead to high-level compilation to CORDICs.

For PAM-Blox, the natural extension is to integrate behavioral synthesis into module generation e.g. using synthesis from C [22, 23]. A behavioral method within the hardware object could be automatically transformed into multiple structural C++ methods.

CORDICs map well onto FPGAs. Due to their small area requirement, CORDICs—especially parallel forms—are most useful for certain highly parallelizeable and pipelineable applications which can take advantage of a large number of CORDIC units on a chip.

Appendix A

Compositions of rotations in 3-D space are best represented in terms of quaternions. Simply, and to facilitate the relation with [3], rotation by an angle u around an axis $\mathbf{u} = [u_x, u_y, u_z]^T$ with $u_x^2 + u_y^2 + u_z^2 = 1$ is evaluated by means of a multiplication by the matrix:

$$Q = \begin{bmatrix} w & -x & -y & -z \\ x & w & -z & y \\ y & z & w & -x \\ z & -y & x & w \end{bmatrix} \quad \text{where } w = \cos u$$

$$\text{and} \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} u_x \\ u_y \\ u_z \end{bmatrix} \cdot \sin u. \qquad \text{(A1)}$$

The product of the rotation by b around $\mathbf{b}$ (matrix Q_b) followed by the rotation by f around $\mathbf{f}$ (matrix Q_f) is given by the first column of $Q_f \cdot Q_b$, hence, in order to determine the resulting rotation angle r and direction $\mathbf{r}$, it is sufficient to multiply the first column of Q_b by Q_f. Exploiting the structure of Q_f and denoting by $\times$ the cross-product of two vectors, the evaluation of the

المنارة للاستشارات

www.manaraa.com

first column of the product yields

$$\begin{bmatrix} \cos r \\ \mathbf{r} \cdot \sin r \end{bmatrix} = \begin{bmatrix} \cos f \cos b - (\mathbf{f} \sin f) \cdot (\mathbf{b} \sin b) \\ (\mathbf{f} \sin f) \cos b + \cos f (\mathbf{b} \sin b) + (\mathbf{f} \sin f) \times (\mathbf{b} \sin b) \end{bmatrix} = \begin{bmatrix} \cos f \cos b - (\mathbf{f} \cdot \mathbf{b}) \sin f \sin b \\ \mathbf{f} \sin f \cos b + \mathbf{b} \cos f \sin b + (\mathbf{f} \times \mathbf{b}) \sin f \sin b \end{bmatrix}. \tag{A2}$$

Specifically (see Eq. (3) and Fig. 5)

$$\begin{cases} \cos f = \nu, & \sin f = \bar{\nu}, & \mathbf{f} = [1, 0, 0]^T, & \mathbf{f} \cdot \mathbf{b} = -\cos R = -\rho, \\ \cos b = \eta, & \sin b = \bar{\eta}, & \mathbf{b} = [-\cos R, 0, \sin R]^T = [-\rho, 0, \bar{\rho}]^T, \\ \cos r = \rho_+, & \sin r = \bar{\rho}_+, & \mathbf{f} \cdot \mathbf{r} = \cos B = \eta_+, & \mathbf{b} \cdot \mathbf{r} = \cos F = \nu_+. \end{cases} \tag{A3}$$

Thus, expressing the vectors **f**, **b** and **r** in terms of their components, to compose the rotations we compute the product

$$\begin{bmatrix} \nu & -\bar{\nu} & 0 & 0 \\ \bar{\nu} & \nu & 0 & 0 \\ 0 & 0 & \nu & -\bar{\nu} \\ 0 & 0 & \bar{\nu} & \nu \end{bmatrix} \begin{bmatrix} \eta \\ -\rho\bar{\eta} \\ 0 \\ \bar{\rho}\bar{\eta} \end{bmatrix} = \begin{bmatrix} \eta\nu + \rho\bar{\eta}\bar{\nu} \\ \eta\bar{\nu} - \rho\bar{\eta}\nu \\ -\bar{\rho}\bar{\eta}\bar{\nu} \\ \bar{\rho}\bar{\eta}\nu \end{bmatrix}. \tag{A4}$$

This equation provides the skeleton of the decomposition of the adaptive filter computations into CORDIC operations. Since Q_f is the composition of two independent plane rotations, this decoupling should be exploited: it is preferable here to employ 2-D CORDICs rather than a quaternion CORDIC (We outline an alternative approach that leads to the use of quaternion CORDIC at the end of Section 4.D), and apply two 2-D CORDICs in parallel for speed. The equation implies that $\rho_+ = \cos r$ may be obtained as the first component of

$$\begin{bmatrix} \nu & -\bar{\nu} \\ \bar{\nu} & \nu \end{bmatrix} \begin{bmatrix} \eta \\ -\rho\bar{\eta} \end{bmatrix}.$$

The second component of that vector is equal to the first component of $\mathbf{r} \sin r$, i.e., $\mathbf{f} \cdot (\mathbf{r} \sin r) = (\mathbf{f} \cdot \mathbf{r}) \sin r = \eta_+ \bar{\rho}_+$. We denote η^* this second component.

Similarly ν_+ can be obtained according to $\nu_+ \bar{\rho}_+ = (\mathbf{b} \cdot \mathbf{r}) \sin r = \mathbf{b} \cdot (\mathbf{r} \sin r) = -\rho\eta^* + \bar{\rho}\nu^\dagger$, where $\nu^\dagger$ is the second component of the vector

$$\begin{bmatrix} \nu & -\bar{\nu} \\ \bar{\nu} & \nu \end{bmatrix} \begin{bmatrix} 0 \\ \bar{\rho}\bar{\eta} \end{bmatrix}.$$

Thus, denoting $\nu^* = \nu_+ \bar{\rho}_+$, $-\nu^*$ may be obtained as the first component of

$$\begin{bmatrix} \rho & -\bar{\rho} \\ \bar{\rho} & \rho \end{bmatrix} \begin{bmatrix} \eta^* \\ \nu^\dagger \end{bmatrix}.$$

Hence, first we compute

$$\begin{bmatrix} \rho_+ \\ \eta^* \end{bmatrix} = \begin{bmatrix} \nu & -\bar{\nu} \\ \bar{\nu} & \nu \end{bmatrix} \begin{bmatrix} \eta \\ -\rho\bar{\eta} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} \nu^\dagger \\ \bar{\nu}\bar{\rho}\bar{\eta} \end{bmatrix} = \begin{bmatrix} \nu & -\bar{\nu} \\ \bar{\nu} & \nu \end{bmatrix} \begin{bmatrix} \bar{\rho}\bar{\eta} \\ 0 \end{bmatrix} \tag{A5}$$

then we evaluate

$$\begin{bmatrix} -\nu^* \\ \eta^\dagger \end{bmatrix} = \begin{bmatrix} -\nu^* \\ \eta \cdot \bar{\rho}\bar{\nu} \end{bmatrix} = \begin{bmatrix} \rho & -\bar{\rho} \\ \bar{\rho} & \rho \end{bmatrix} \begin{bmatrix} \eta^* \\ \nu^\dagger \end{bmatrix} \tag{A6}$$

and finally we obtain

$$\nu_+ = \nu^* / \bar{\rho}_+ \quad \text{and} \quad \eta_+ = \eta^* / \bar{\rho}_+. \tag{A7}$$

This way of decomposing the Eq. (1), guided by our geometric interpretation, leads to a computational structure different from that of [10, 18].

Proceeding with our geometrical approach we shall also use the relation between the sines of the angles and sides of a spherical triangle ("the law of sines"),

$$\frac{\bar{\rho}}{\bar{\rho}_+} = \frac{\bar{\nu}_+}{\bar{\nu}} = \frac{\bar{\eta}_+}{\bar{\eta}}$$

to update the sines $\bar{\nu}$ and $\bar{\eta}$:

$$\bar{\nu}_+ = \bar{\nu} \cdot (\bar{\rho}/\bar{\rho}_+) \quad \text{and} \quad \bar{\eta}_+ = \bar{\eta} \cdot (\bar{\rho}/\bar{\rho}_+) \tag{A8}$$

Acknowledgments

This research is supported by DARPA Grant No. DABT63-96-C-0106 and CNET under Grant 941B098. We thank M. Flynn and G. De Micheli for support and encouragement. We would also like to thank Synopsys and Xilinx for the synthesis tools used in this paper, and Compaq Systems Research Center for support of the PAM-Blox project.

References

1. W.H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. Prasanna, and H. Spaanenburg, "Seeking Solutions in Configurable Computing," *IEEE Computer Magazine*, Dec. 1997.

www.manaraa.com

2. J.M. Muller, *Elementary Functions: Algorithms and Implementation*, Birkhauser, 1997.
3. J.-M. Delosme and S.-F. Hsiao, "CORDIC Algorithms in Four Dimensions, Advanced Signal Processing Algorithms, Architectures and Implementations," *Proc. SPIE 1348*, San Diego, CA, July 1990, pp. 349–360.
4. H.M. Ahmed, "Signal Processing Algorithms and Architectures," Ph.D. Thesis, E.E. Dept., Stanford, June 1982.
5. R. Andraka, "A Survey of CORDIC Algorithms for FPGA Based Computers," *Sixth International Symposium on Field Programmable Gate Arrays*, Monterey, CA, 1998.
6. J.E. Volder, "The CORDIC Trigonometric Computing Technique," *IRE Trans. on Electronic Computers*, vol. EC-8, no. 3, Sept. 1959, pp. 330–334.
7. J.S. Walther, "A Unified Algorithm for Elementary Functions," *AFIPS Conf. Proc.*, vol. 38, 1971, pp. 379–385.
8. T.C. Chen, "Automatic Computation of Exponentials, Logarithms, Ratios and Square Roots," *IBM Journal of Research and Development*, July 1972.
9. J.-M. Delosme, "VLSI Implementation of Rotations in Pseudo-Euclidean Spaces," *Proc. 1983 Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, Boston, MA, April 1983, pp. 927–930.
10. D.T.L. Lee and M. Morf, "Generalized CORDIC for Digital Signal Processing," *Proc. 1982 Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, Paris, France, May 1982, pp. 1748–1751. See also Ph.D. Thesis, E.E. Dept., Stanford, Aug. 1980.
11. H. Al-Twaijry, "Area and Performance Optimized CMOS Multipliers," Ph.D. Thesis, Stanford, Aug. 1997.
12. O. Mencer, M. Morf, and M.J. Flynn, "PAM-Blox: High Performance FPGA Design for Adaptive Computing," *IEEE Symp. on FPGAs for Custom Computing Machines*, Napa Valley, CA, 1998. http://umunhum.stanford.edu/PAM-Blox/.
13. O. Mencer, M. Morf, and M.J. Flynn, "Hardware Software Tri-Design of Encryption for Mobile Communication Units," *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, Seattle, May 1998.
14. O. Mencer and M. Platzner, "Dynamic Circuit Generation for Boolean Satisfiability in an Object-Oriented Design Environment," *Hawaii Int. Conf. on System Sciences (ConfigWare Track)*, Jan. 1999.
15. Synopsys FPGA Express, http://www.synopsys.com/products/fpga_solution/fpga_express.html.
16. G. De Micheli, *Synthesis and Optimization of Digital Circuits*, Highstown, NJ: McGraw Hill, 1994.
17. Synopsys Products, http://www.synopsys.com/products.
18. J. Turner, Prentice-Hall Signal Processing Series, Ch. 5, 1985.
19. H.M. Ahmed, P.H. Ang, and M. Morf, "A VLSI Speech Anaysis Chip Set Utilizing Co-ordinate Rotation Arithmetic," *Proc. 1981 Int. Symp. on Circuits and Systems (ISCAS),* Chicago, Illinois, April 1981, pp. 737–741.
20. H.M. Ahmed, P.H. Ang, and M. Morf, "A VLSI Speech Analysis Chip Set Based on Square-Root Normalized Ladder Forms," *Proc. 1981 Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*, Atlanta, GA, March-April 1981, pp. 648–653.
21. M. Morf and D.T. Lee, "State-Space Structure of Ladder Canonical Forms," *Proc. 18th Conf. on Control and Design*, Dec. 1980, pp. 1221–1224.
22. L. Séméria and G. De Micheli, "Synthesis of Pointers in C: Application of Pointer Analysis to the Behavioral Synthesis from C," *Proceedings of the International Conference on Computer-Aided Design (ICCAD 1998)*, San Jose, Nov. 1998.
23. T.J. Callahan and J. Wawrzynek, "Instruction Level Parallelism for Reconfigurable Computing," *FPL'98*, Tallinn, Estonia, Sept. 1998, pp. 248–258. Publilshed in Springer-Verlag LNCS 1482.

Oskar Mencer is a Ph.D. Candidate in the Department of Electrical Engineering at Stanford University. He received an M.S. degree in Electrical Engineering at Stanford and a B.S. degree at the Technion/Israel. He held summer positions at Rockwell, DIGITALs Systems Research Center, and Hitachi in Japan. His research interests are computer arithmetic, computer architecture and microarchitecture, and reconfigurable (custom) computing.
oskar@stanford.edu

Luc Séméria is a Ph.D. student in the Electrical Engineering Department at Stanford University. He received an Engineer Degree (ENST 96) and a M.S. degree in Electrical Engineering (Stanford University 98). While studying he held several summer positions at Synopsys Inc. His research interests include system-level design, hardware/software co-design and compiler techniques. He is currently working on the synthesis of hardware from C.
semeria@stanford.edu

المنارة للاستشارات
www.manaraa.c

Martin Morf received his Federal Diploma in Electrical Engineering, at the ETH-Zurich, and a Ph.D. in Electrical Engineering from Stanford University. He is currently Visiting Professor in the Computer Systems Laboratory at Stanford University. He is author of over 240 publications. His research interests include CORDIC arithmetic, X-Modulators, Quantum-Dots, Photonic Networks, Reconfigurable Radios, Computer Security, and Bio-computing.
morf@stanford.edu

Jean-Marc Delosme received the engineer's degree and the specialization in automatic control degree from the École Nationale Supérieure de l'Aéronautique et de l'Espace, France, and the M.S. and Ph.D. degrees in Electrical Engineering from Stanford University. Dr. Delosme joined the faculty of the Department of Electrical Engineering at Yale University in 1983 after a year as a research associate at the Information Systems Laboratory, Stanford University. Since 1996 he is Professor of Computer Science at the University of Evry, France. His research interests are in the general area of architecture and algorithm design for real-time and computationally intensive applications.
delosme@lami.univ-evry.fr

المنارة للاستشارات

www.manaraa.com

Journal of VLSI Signal Processing Systems 24, 223–240, 2000.
© 2000 Kluwer Academic Publishers. Printed in The Netherlands.

A Configurable Logic Based Architecture for Real-Time Continuous Speech Recognition Using Hidden Markov Models

PANAGIOTIS STOGIANNOS, APOSTOLOS DOLLAS AND VASSILIS DIGALAKIS

Technical University of Crete, 73100 Chania, Crete, Greece

Abstract. An architecture is presented for real-time continuous speech recognition based on a modified hidden Markov model. The algorithm is adapted to the needs of continuous speech recognition by efficient encoding of the state space, and logarithmic encoding of the weights so that products can be computed as sums. The paper presents the algorithm and its application related modifications, the mapping of the algorithm to a special purpose architecture, and the detailed design of this architecture using configurable logic. Emphasis is given on how the attributes of the algorithm are exploited in a configurable logic based design. A concrete design example is presented with a coprocessor engine having one large FPGA, 64 Mbytes of synchronous DRAM (SDRAM), a small FPGA as a SDRAM controller, and 2 Mbytes SRAM. This engine operating at 66 MHz performs roughly nine times as fast as a high end personal computer running a fully optimized version of the same algorithm.

1. Introduction

Speech recognition is widely considered as one of the main areas of growth in the near future [1], with new applications such as voice commerce (v-commerce), which is a variant of electronic commerce (e-commerce). In order for this growth to be realized, three problems need to be addressed:

- The recognition needs to be performed on continuous speech by systems which are speaker-independent and require no per-speaker training,
- The recognition rates need to be improved, and,
- The resulting solution from a computational point of view must be implementable in a real-time system (preferably embeddable).

The above three problems are interrelated, as some of the best algorithms for speaker-independent, continuous-speech recognition are also the most computationally demanding. Although the speed of general-purpose systems (usually personal computers) improves substantially at the rate described by Moore's law, the desire to have ever-increasing recognition rates and even multi-channel systems for commercial applications (e.g. for mail order sales) points towards the direction of specialized hardware.

Software approaches to speech recognition already exist. IBM's Via Voice and Dragon's Naturally Speaking are large-vocabulary dictation applications that run on PCs. There is a very large difference, however, between the large-vocabulary continuous-speech recognition (LVCSR) systems used in the DARPA benchmarks and products that must run at or even faster than real time. Traditionally, benchmark LVCSR engines run at tens or even hundreds times slower than real time, and perform significantly better than their product counterparts. Dictation products such as those mentioned above are characterized by fairly good performance with fairly inexpensive prices that are likely to drop dramatically in the future. An accelerator to replace dictation products would remove two of their most desirable features, simplicity and low cost (even high end software over time gets discounted or even bundled with standard software distributions). A hardware accelerator is especially useful in multi-port applications for telephony platforms, and in applications which require the lowest possible word error rate.

المنارة للاستشارات

www.manaraa.c

A recognition server running on a single host computer serves, with the aid of accelerators, a number of different channels connected to it.

Technology-wise, the fastest solution for hardware implementation of any system is with VLSI design, and such designs have already been made [2, 3]. These approaches, however, have not led to widespread usage because the design time for VLSI design, combined with its lack of flexibility once implemented, lead to rapid obsolesense of such systems. The case is especially aggravated by the quick pace of development of new algorithms and speed improvement of general computers. Therefore, an alternative approach is needed, in which the hardware does provide at least one order of magnitude speedup over software solutions, but at the same time remains flexible enough to run different variants of some class of algorithms, with its datapath and state-space customized according to the application. The usage of Field Programmable Gate Arrays (FPGA) [4] was thus considered as a good tradeoff between speed and flexibility, the latter being achieved through reconfiguration.

The usage of FPGA's for speech recognition is not entirely new. The implementation of hidden Markov Models, a highly successful model in continuous speech recognition was done using FPGA's in 1995 [5], for a 255 discrete word vocabulary (non continuous speech). A reconfigurable Viterbi decoder (which too is one of the key algorithms in speech recognition) was reported in 1996 [6]. To date, however, no system has appeared that partitions the entire continuous, large vocabulary speech recognition problem to software and reconfigurable hardware subsystems. Such a system is presented in this work. An architecture has been designed for the computational core of a state-of-the-art speech recognition algorithm. The new architecture is in effect a coprocessor, relying on the front end for preprocessing of data (e.g. Fourier transforms), which can be easily done in software, whereas speeds up by an order of magnitude the modified Hidden Markov Model computational core which accounts for 80% of the total computations (including preprocessing and grammar extraction).

The emphasis of this paper is on how the new architecture was developed in order to meet its performance goals for a specific set of parameter values, but the solution is general enough that through reconfiguration different systems can emerge with no board-level changes. Section 2 briefly presents the Hidden Markov Model (HMM) and modified HMM which are used in continuous speech recognition. Section 3 presents the new architecture, whereas Section 4 presents design-related issues, including usage of resources, and system speed. Finally, the status of the system and some conclusions of this work are summarized in Section 5. We note that although it is clear that such a coprocessor can form the basis for large, server type applications, this paper focuses on how a specific top performing algorithm which to date does not run in real time can be sped up to multi-channel real time operation. The specifics of its usage in multi-channel servers, however, is beyond the scope of the paper.

2. A Modified Hidden Markov Model for Continuous Speech Recognition

2.1. HMM-Based Speech Recognition

Today's state-of-the-art speech recognizers are based on statistical techniques, with the hidden Markov models being the dominant approach [7]. The typical components of a speech recognition and understanding system are the front-end processor, the decoder with its acoustic and language models, and the language understanding component. The latter component extracts the meaning of a decoded word sequence, and is an essential part of a natural language system. The remainder of this section briefly reviews the front end and the decoder.

The *front-end processor* typically performs a short-time Fourier analysis and extracts a sequence of observation vectors (or *acoustic* vectors) $\mathcal{O} = [O_1, O_2, \ldots, O_T]$. Many choices exist for the acoustic vectors, but the melwarped cepstral coefficients (MFCCs) have exhibited the best performance to date [8]. The sequence of acoustic vectors can either be modeled directly, or vector-quantized first and then modeled.

The *decoder* is based on a communication theory view of the recognition problem, trying to extract the most likely sequence of words $\mathcal{A} = [A_1, A_2, \ldots, A_N]$ given the set of acoustic vectors $\mathcal{O}$. This can be done using Bayes' rule:

$$\hat{\mathcal{A}} = \arg\max_{\mathcal{A}} \frac{\mathrm{P}(\mathcal{A})\mathrm{p}(\mathcal{O} \mid \mathcal{A})}{\mathrm{p}(\mathcal{O})}. \quad (1)$$

The discrete probability $\mathrm{P}(\mathcal{A})$ of the word sequence $\mathcal{A}$ is obtained from the *language model*, whereas the acoustic model determines the likelihood $\mathrm{p}(\mathcal{O} \mid \mathcal{A})$.

In HMM-based recognizers, the probability of an observation sequence for a given word is obtained by building a finite-state model, possibly by concatenating

المنارة للاستشارات
www.manaraa.com

models of the elementary speech sounds or phonemes. The state sequence $\mathcal{S} = [S_1, S_2, \ldots, S_T]$ is modeled as a Markov chain, and is not observed. At each state S_t and time t, an acoustic vector is observed based on the distribution $b_{s_t}(O_t) = \mathrm{p}(O_t \mid S_t)$, which is called *output distribution*. Because HMMs assume, for simplicity, that observations are independent of their neighbors, first- and second-order derivatives of the cepstral coefficients are included in the acoustic vector O_t. If the front-end vector quantizes the acoustic vectors, the output distributions take the form of discrete probability distributions. If the acoustic vector generated by the front end is passed directly to the acoustic model, then continuous-density output distributions are used, with the multivariate-mixture Gaussians being the most common choice:

$$b_S(O_t) = \sum_{i=1}^{K} \mathrm{P}(W_i \mid S)\mathcal{N}(O_t; \mu_{si}, \Sigma_{si}), \qquad (2)$$

where $\mathrm{P}(W_i \mid S)$ is the weight of the ith mixture component in state S, and $\mathcal{N}(O; \mu, \Sigma)$ is the multivariate Gaussian with mean μ and covariance Σ. The mixture weights are nonnegative and must sum to one. The continuous-density HMMs (CDHMMs) are used in most state-of-the-art large-vocabulary continuous-speech recognition systems, including the Decipher system of SRI International [9]. The main disadvantage of CDHMMs is their computational complexity, especially for large-vocabulary applications, with decoding times being much slower than real time when it is desired to achieve the best possible recognition performance. Decoding in real-time is achieved when the decoding time is less than or equal to the length of the recognized utterance.

In a HMM-based speech recognizer, HMMs represent the basic speech sounds or phonemes. To model coarticulation—the influence of neighboring phonemes on the pronunciation of a phoneme due to the inertia of the vocal tract—different HMMs are used for a particular phoneme based on the context in which it appears. This results to a large number of *context-dependent* phone models. A *phone* is the acoustic realization of a specific phoneme in a particular context. Each of these models consists of a number of states (typically three to five) that correspond to the beginning, middle and end of the phone.

The computation of the Gaussian-mixture output probabilities (2) for large-vocabulary applications cannot be performed in near-real time. A more efficient scheme was proposed in [9], by clustering different states together based on the similarity of their distributions. These groups of states use Gaussians from a common set, which is defined as a *genone*. Each state retains, however, its own mixture weights $\mathrm{P}(W_i \mid S)$.

Decoding in HMM-based speech recognizers is performed by searching for the most-likely word sequence $\mathcal{A}$. This is often approximated by finding the most-likely state sequence in a finite-state network that is built by connecting the basic phone-HMMs according to a set of grammar rules. Although many algorithms exist, the most common is the *Viterbi beam search* [10]. The time-synchronous beam search is a suboptimal version of the Viterbi[1] algorithm, in which only the most likely states survive (are *active*) at each time. The set of states that are active at each time constitutes the *search space*. Decoding time can be adjusted by controlling the beam width, that is, the maximum distance (in logarithmic scale) that a particular theory can have from the current best in order to survive. Reducing the beam width speeds up decoding, but can also introduce search errors.

2.2. Discrete-Mixture HMMs

In [11] we developed a novel encoding scheme for the transmission of the MFCCs in a client-server architecture for speech-enabled applications over the World Wide Web (WWW) and wireless channels. By using subvector quantization and a bit-allocation algorithm that was driven by speech recognition performance, we were able to encode the 13 MFCCs using as little as 20 bits in noise-free environments, while maintaining the recognition performance of a high-quality CDHMM recognizer. This was a rather surprising result, given that HMM-based state-of-the-art recognition systems today represent the MFCCs using floating-point arithmetic and model their distributions with Gaussian mixtures.

The possibility of representing the MFCCs with a small number of bits, instead of the 416 (=13 coefficients × 32 bits per coefficient) that are traditionally used in CDHMMs, in addition to being advantageous for transmission and storage, has serious implications in acoustic modeling. Using Gaussian mixtures to model a set of coefficients that can be represented with 20 bits is clearly overkill. In [12] we demonstrated that the high level of recognition performance of CDHMMs can be maintained with a far more efficient type of HMM, the discrete-mixture HMM (DMHMMs) with subvector quantization of the coefficient parameters.

المنارة للاستشارات
www.manaraa.com

Using subvector-quantization, we first partition the vector of MFCCs into L subvectors, $O_t = [O_{t1}, O_{t2}, \ldots, O_{tL}]$, and then we quantize each subvector using a separate VQ codebook,

$$\begin{aligned} X_t &= \mathrm{vq}(O_t) = [\mathrm{vq}(O_{t1}), \mathrm{vq}(O_{t2}), \ldots, \mathrm{vq}(O_{tL})] \\ &= [X_{t1}, X_{t2}, \ldots, X_{tL}]. \end{aligned}$$

Then, the Gaussian mixture output distribution (2) in CDHMMs can be replaced with a mixture of discrete probability distributions of the following form:

$$b_S(O_t) \doteq \mathrm{P}(\mathrm{vq}(O_t)) - \sum_{i=1}^{K} \mathrm{P}(W_i \mid S) \prod_{j=1}^{L} \mathrm{P}(X_{tj} \mid W_i, S), \tag{3}$$

where $\mathrm{P}(X_{tj} \mid W_i, S)$ is the probability of observing the discrete symbol $X_{tj} = \mathrm{vq}(O_{tj})$ for the jth subvector. The output distribution introduced above assumes that the indices of different subvectors are conditionally independent given the state and mixture index. Dependencies between the different subvectors for a given state are modeled through the mixture components.

When compared to the conventional CDHMMs, the discrete mixture HMMs replace a multivariate Gaussian density with the product of L discrete distributions, one for each subvector. The amount of computation can be reduced by decreasing L. At the same time, however, the number of bits required to represent each subvector increases, and this corresponds to an exponential increase in the amount of memory required to store the look-up tables of the discrete distributions. In [12] we found that a good compromise is to use $L = 15$ subvectors, and we showed that a speed-up of a factor of two to three can be achieved using this form of output distribution, while maintaining the level of performance of CDHMMs.

The DMHMM results are summarized in Fig. 1, where we plot the performance of two systems, a baseline CDHMM and a DMHMM with 15 subvectors, versus the decoding time. Performance is measured in *Word Error Rate* (WER), that is, the percentage of words that are erroneously recognized, including substitutions, deletions and insertions. We use the air-travel information domain (see Section 2.3), and the decoding time is adjusted by varying the beam width. The DMHMM system performs uniformly better than the baseline CDHMM system. Although the DMHMM is two to three times faster than the CDHMM, the WER increases (search errors are introduced) when the beam width is adjusted so that decoding is performed in less

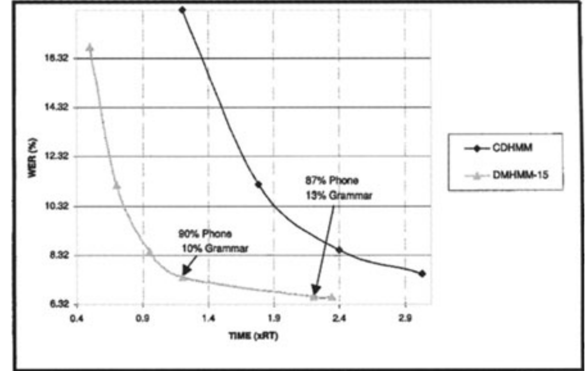


Figure 1. A comparison of CDHMM and DMHMM.

than two times real time. We also found, at different operating points, that roughly 90% of the decoding time is spent on the phone-processor, which performs the Viterbi search within a HMM phone model. 80% of this time is spent computing the output probabilities (3).

The previous experiments were performed on a P-II 266 MHz that was available at that time. Faster processors, which have been made available since then, can speed up the recognizer but cannot achieve real-time performance. In a recent test we performed in our lab we found that two, otherwise equivalently configured PCs, with P-II processors running at 266 and 333 MHz, respectively, had only a 7% difference in decoding time.

2.3. *A Realistic Continuous Speech Application*

So far, we have presented the DMHMM model as an appropriate modification of the general HMM for the problem at hand. In this section we constraint the problem size so that a meaningful FPGA-based architecture can be developed and compared with software-only methods. The application we selected for this work is the air-travel information (ATIS) domain [13]. In the ATIS domain, a user can get flight information and book flights across the United States using natural language. It consists of a vocabulary of approximately 1,500 words, with a moderate perplexity (a measure of difficulty). The recognizer for the ATIS domain is configured with 10,872 states, 1105 genones (that is, states are clustered in 1105 groups) and each genone is comprised by 32 Gaussians. In Section 3 we will present the complete reconfigurable architecture for this application, but we would like to emphasize that because it

www.manaraa.com

is reconfigurable, a broad set of different alternatives can be mapped on the same hardware with minimal design changes and no board-level modifications.

2.4. *System Partitioning in Software and Hardware*

Figure 1 shows the word error rate as a function of the processing time for the reference system. Even with the reduced computational requirements of the DMHMM model vs. the CDHMM model we see that the lowest WER requires over twice the speed of a general purpose computer. Even with newer models, it can be appreciated that the corresponding WER can be further reduced with specialized hardware, or, the personal computer resources can be freed up from performing speech recognition only. In the two points of the DMHMM curve the phone processing vs. grammar extraction times are presented. The phone processing accounts for 90% and 87% of the total times, and, not shown in the curve, 90% of these times are spent computing transition probabilities which thus account for 81% and 78% of the entire workload, including front end processing and grammar extraction. This leads to a natural software-hardware decomposition of the DMHMM model, in which the hardware architecture we will present speeds up roughly 80% of the entire workload. This part is the computation of the relation which gives the output probability $P(X_t \mid S)$ for a given set of $[S, X_t]$. The architecture was designed for a system that has 10872 states, 32 weights per state, 1105 genones and its observation vectors consist of 15 5-bit elements.

This breakdown allows for the reconfigurable engine to speed up the evaluation of the observation-distribution likelihoods (or output probabilities), which are the computational core of the application, whereas the front end processing, and the HMM search and grammar extraction are performed on the host processor.

3. An Architecture for Real-Time Computation of the DMHMM

The system architecture is dominated by two major facts that arise from the theoretical model. Those facts are:

- The relationship that needs to be computed for each set $[S, X_t]$ can be divided in two parts:
  1. The computation of the products $P(W_i \mid S) \prod_{j=1}^{15} P(X_{tj} \mid W_i, S)$, where S is the active state, $P(W_i \mid S)$ is the ith mixture weight of the active state, and X_{tj} is the jth observation vector element, and $P(X_{tj} \mid W_i, S)$ is the probability of observing the discrete symbol X_{tj} for the jth subvector.
  2. The computation of the sum $\sum_{i=1}^{32} P(W_i \mid S) \prod_{j=1}^{15} P(X_{tj} \mid W_i, S)$ *for each active state of the search space*, together with the comparison for the extraction of the maximum output probability. Those two parts can be designed separately as two hardware systems.
- The calculation of probability products can be converted to sums of logarithms. This approach has two benefits: it makes the design of the ALUs less complex (space reduction) and it speeds up the whole process (performing one addition and one log-to-fixed point transformation is computationally less expensive than performing one multiplication).

Those two facts led to an architecture with the following properties: deep pipeline, parallelism, partitioned ALUs and data restructuring FIFOs. The parts which form the entire system are shown in Fig. 2. These are:

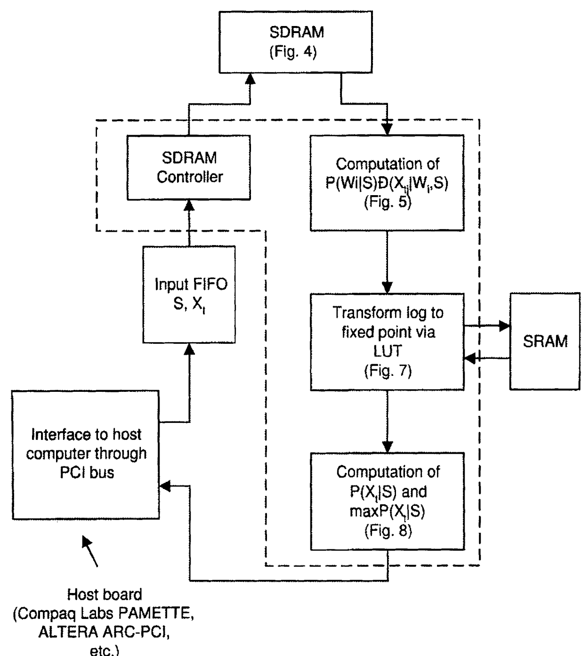


Figure 2. Block diagram of the system.

للاستشارات
www.manaraa.com

- The interface between the system and the host computer via the PCI bus of the host
- The system that computes the probabilities $P(W_i \mid S) \prod_{j=1}^{15} P(X_{tj} \mid W_i, S)$
- The look up table from logarithm to a fixed point real number
- The system that computes the output probabilities $\sum_{i=1}^{32} P(W_i \mid S) \prod_{j=1}^{15} P(X_{tj} \mid W_i, S)$
- The memory, SDRAM, which stores the data that is necessary for the calculations, and,
- the control of the system

Figure 2 shows how the new architecture communicates with the outside world via the PCI bus and input FIFO, as well as the SDRAM controller which has been designed to ensure correct operation and data streaming of the DRAM. The rest of this paper will focus on the computational aspects of the new architecture, but it is understood that the elements shown in Fig. 2 are all needed. The input FIFO has not been designed yet because as will be described below, for the first generation of the system it is envisioned that a commercial board which already has a PCI interface and a run time environment (e.g. the Compaq Labs PAMETTE [14]) will be used. The PAMETTE already has FPGA's which can be used to implement the FIFO and data buffering.

The system has three kinds of input and two kinds of output as Table 1 shows (together with their width). The inputs are: the index of the active genone (1105 possible values), the index of the active states (10872 possible values), and the observation vector elements. Those inputs produce the output probability for each of the active states and the maximum output probability for the active states.

The main architecture and data flow of the new system can be viewed in Fig. 3. As compared to Fig. 2, Fig. 3 does not show I/O but only the calculation subsystems and memories.

Table 1. Inputs and outputs of the system.

Information name	Kind of information	Values	Width (bits)
Genone index	Input	1105	11
State index	Input	10872	14
Observation element	Input	32	5
Output prob.	Output	–	16
Max. output prob.	Output	–	16

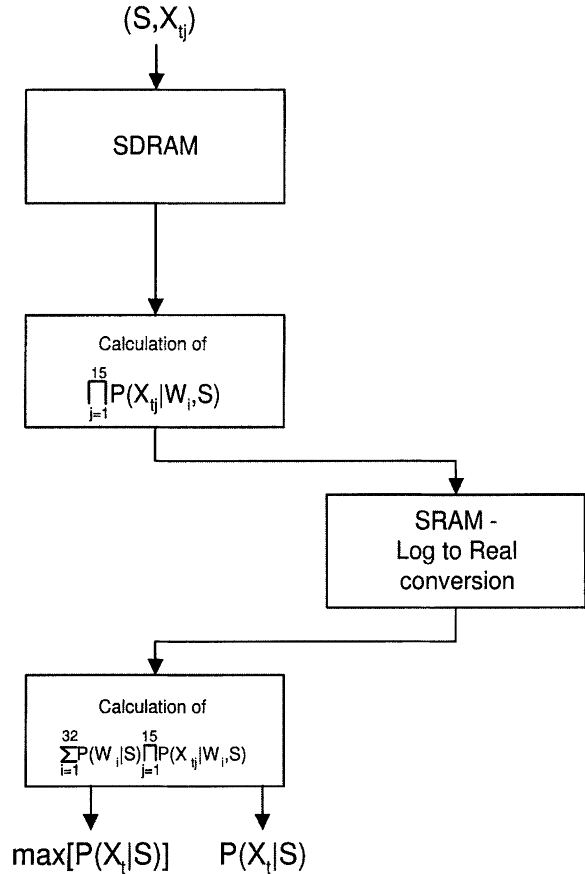


Figure 3. Block diagram of the architecture.

3.1. The SDRAM Subsystem

The first subsystem, which has all the data that is necessary to calculate the output probabilities is a 64 Mbyte SDRAM with a 64-bit wide datapath. Its pre-stored data includes:

- The probabilities $P(X_{tj} \mid W_i, S)$
- The probabilities $P(W_i \mid S)$

where X_{tj} are the observations, W_i are the weights of the states and S is the active genone. Those probabilities are stored in their logarithmic value and have a width of 16 bits, which means that each memory location has four values stored. In order to access sequentially a large amount of data, using the capabilities for bursty transfers that the SDRAM technology offers, the address of each memory location has a specific meaning. As Fig. 4 shows each address gives the

www.manaraa.com

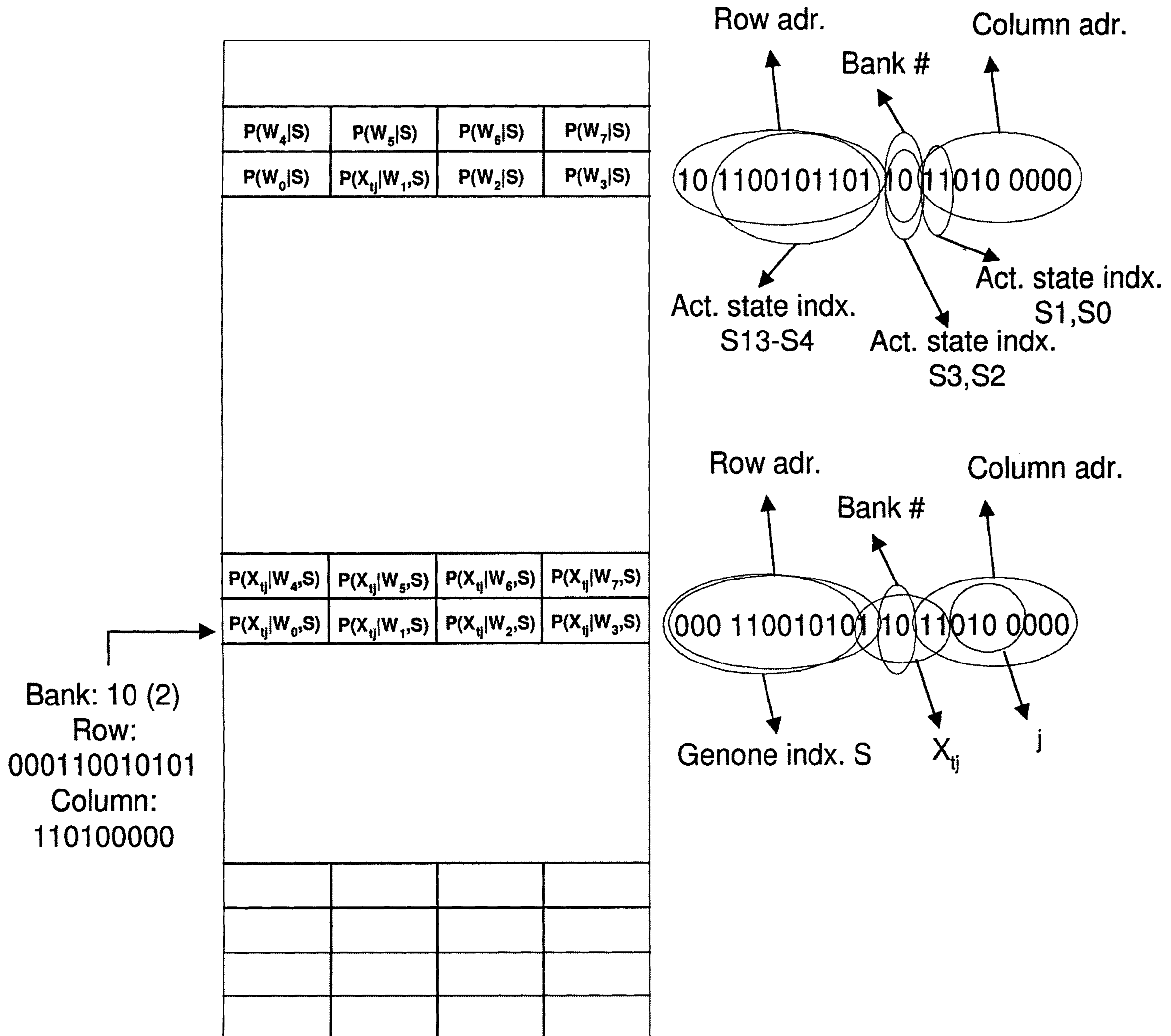


Figure 4. Organization of the SDRAM contents.

information of the genone, the active state, the observation and the weight to which the information refers. The Row, Bank, and Column of the full SDRAM address are formed as shown in Table 2.

According to this bit the rest of the address bits follow the rule that is shown in Fig. 4. For example, column 110110000 of row 000110010101 in bank 10 contains the probabilities $P(X_{tj} \mid W_i, S)$, where $S = 00011001010$, $X_{tj} = 11011$ and for the $j = 0100_2$-th subvector. With this kind of addressing the system can access all $P(X_{tj} \mid W_i, S)$ probabilities for a specific X_{tj}, a specific j, and its 32 weights with just one address when the SDRAM works in the burst mode [15], where it accesses 8 sequential addresses. This fact, together with the four internally interleaved banks of the SDRAM, is very important for the speed of the

Table 2. Address generation for the SDRAM.

Kind of probability	Row (12 bits)	Bank (2 bits)	Column (9 bits)
$P(X_{tj} \mid W_i, S)$	$S[10\ldots0]\ X_{tj}\ [4]$	$X_{tj}[3\ldots2]$	$X_{tj}[1\ldots0]\ J[3\ldots0]\ 000$
$P(W_i \mid S)$	$10(2624_{10} + S)\ [13\ldots4]$	$(2624_{10} + S)\ [3\ldots2]$	$(2624_{10} + S)\ [1\ldots0]\ 0000000$

المنارة للاستشارات
www.manaraa.com

Table 3. The opcodes of the subwords.

Opcode	Denotation
00	Command
01	Genone index
10	Active state index
11	Observation vector element

whole system as it can provide the system with data on every cycle. The address generation and the write/read cycels are being controlled by a FSM.

3.2. The I/O and Look Up Table Subsystems

The second and the fourth blocks shown in Fig. 3, which will be presented later, are the two systems that do all the calculations. The SRAM that is between them acts like a LUT that converts 16-bit wide logarithms to 16-bit fixed point real numbers. The system I/O with the host computer, for both command and data transactions, is through a 64-bit wide PCI interface, e.g. the Compaq Labs PAMETTE [14, 16] or the ALTERA ARC PCI. Each word that comes from the PCI consists of four 16-bit wide subwords. According to the opcode, shown in Table 3, which the two MSB of each subword form, the system knows what kind of information the rest of the subword carries. This information can be:

- The index of the active genone which is 11-bit wide (1105 genones)
- 5-bit wide elements, X_{tj}, of the observation vector X_t
- The 14-bit wide indices of the active states (10872 states)
- Command

The subwords are generated at the output of the FIFO that stores the incoming words. If it is a command it gets to the controller of the system. If not it is being used in order to build the address for the SDRAM.

3.3. The Datapath for the Probability Calculations

The procedure of computing the output probabilities for a given set of active states and a vector X_t of sampled speech begins with the restructuring of the row data that comes from the host computer. The controller

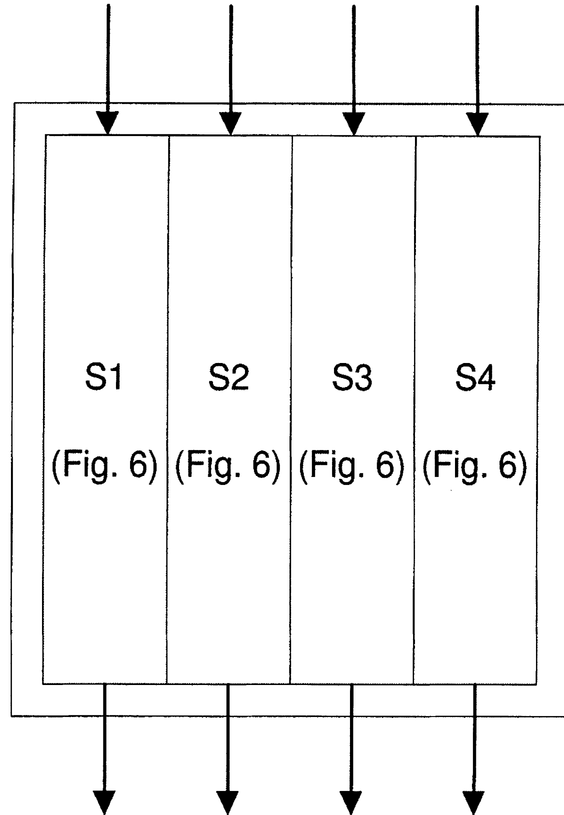


Figure 5. Parallelism in the addition datapath.

does the necessary actions and produces the addresses, which refer to the data associated with the currently active state and observation vector. The outputs of the SDRAM are ready to be processed.

The system which produces the $P(W_i \mid S) \prod_{j=1}^{15} P(X_{tj} \mid W_i, S)$ quantities consists of four equal slices, shown in Fig. 5. This way a parallelism of 4 is being achieved. Each slice does the computations in two parts: first it computes 8 out of the 32 quantities $\prod_{j=1}^{15} P(X_{tj} \mid W_i, S)$, each slice for 4 values of i, and then it produces the full product for the weights it holds. Such a slice (shown in Fig. 6) consists of two parallel FIFOs, which restructure the data from the SDRAM in order to feed the adder. Each cycle it gets new datum from the memory and puts the 8 LSBs in the small FIFO and the 8 MSBs in the large FIFO. At each cycle at the output of the FIFOs, the LSB part of input n comes out together with the MSB part of input $(n-1)$. This does not hold for the first 8 words of data at the beginning of the procedure. Those words bypass the FIFOs and the adder in order to fill the second set of FIFOs, which hold the sums. After filling this set it is restructured

المنارة للاستشارات
www.manaraa.com

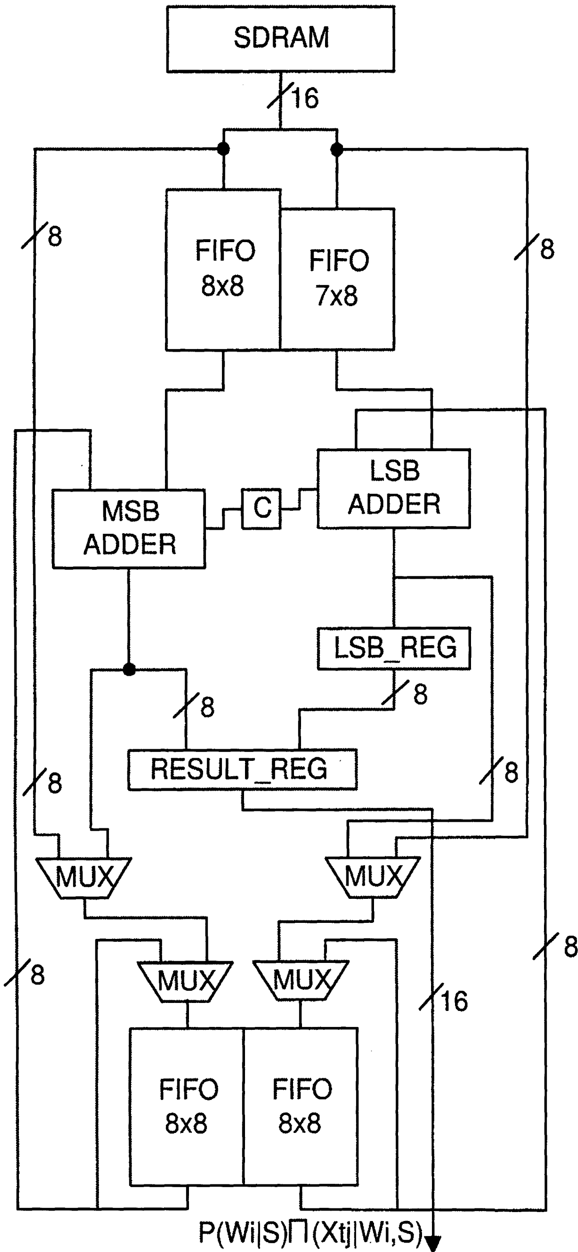


Figure 6. Datapath for addition.

by using the left FIFO only as a cyclic one (while the right FIFO is idle), and the same time the input FIFOs are being filled with the next set of data. This operation lasts for 7 cycles, at the end of which, the data in the second set of FIFOs have the same alignment as those in the first set. From this point the slice performs one addition per cycle. The algorithm that describes the production of the $\prod_{j=1}^{15} \mathrm{P}(X_{tj} \mid W_i, S)$ quantities is the following:

for $j = 1$ to 15 do
for $i = 0$ to 7 do
add $\mathrm{P}(X_{tj} \mid W_i, S)$ to $\prod_{j=1}^{j-1} \mathrm{P}(X_{tj} \mid W_i, S)$
store sum in FIFO
end for;
end for;

In order to describe the systems which do the computational work, first the partitioning of the ALUs has to be presented. Several models of adders were used as ALUs before deciding on a two stage pipeline. The problem was the addition speed. Even fast carry-lookahead adders, which normally give high speeds in VLSI, were a big bottleneck in the system performance because of the way they were mapped to FPGAs (the nature of the logic cells does not allow to implement them in two stage logic due to fanout and routing considerations). This point led to a partitioned adder, shown in Fig. 6, which gives a higher clock rate and works very fast with bursty input. Every addition needs two cycles to produce a result: one for the 8 LSB and one for the 8 MSB. But feeding in the second cycle the 8-bit adder of the first stage with the LSByte of the next input the system can keep both 8-bit adders busy every cycle. By restructuring the input this way, in each cycle the 8 LSB of the nth input are being processed by the first stage of the adder and the 8 MSB of the $(n-1)$th input by the second. This feature gives the 16-bit adder the capability of making n additions in $(n+2)$ cycles.

The results of the adder are being kept in the second set of FIFOs restructured, but can also be read in their actual form from the RESULT_REG. After computing all of the P's the second FIFO system begins to act as a cyclic one. The reason for this mode change, through the multiplexers in its inputs, is that the P's have to be used unchanged for the production of the $\mathrm{P}(W_i \mid S) \prod_{j=1}^{15} \mathrm{P}(X_{tj} \mid W_i, S)$ quantities. The algorithm for this second procedure is the following:

for $j = 1$ to (number of active states S) do
for $i = 0$ to 7 do
add $\mathrm{P}(W_i \mid S)$ to $\prod_{j=1}^{15} \mathrm{P}(X_{tj} \mid W_i, S)$
give the sum as output
store $\prod_{j=1}^{15} \mathrm{P}(X_{tj} \mid W_i, S)$ back in FIFO
end for;
end for;

It must be noted that the number of calculations in this procedure is not fixed, but it depends on the

المنارة للاستشارات
www.manaraa.com

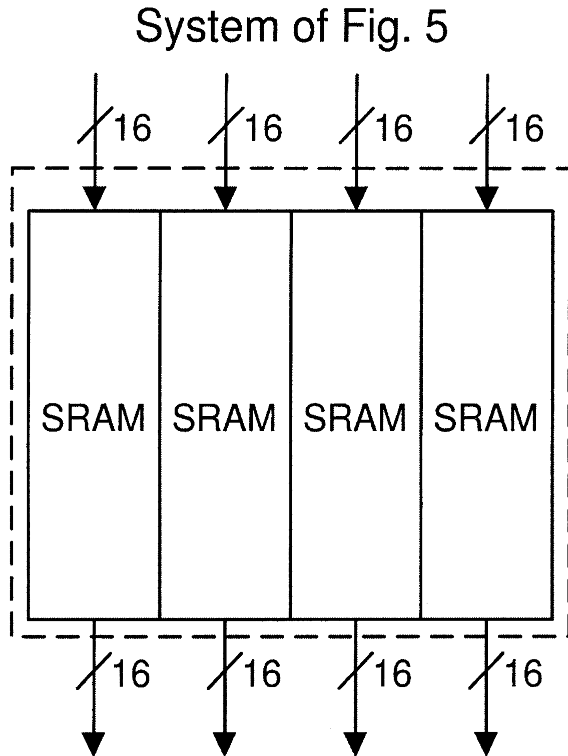


Figure 7. Logarithm to fixed point conversion look up table.

number of the active states, which vary from 1 to 57 per genone. Each such state produces 32 outputs (8 in each slice). Those results come out from each of the 4 RESULT_REG, in their regular form and not restructured, and have to be added together in order to give the output probability for the active states. In order to compute those sums all the results have to be converted to real numbers from logarithms. This is accomplished by 4 SRAMs, which act like a LUTs, and can be viewed as the interface between the two processing units, as shown in Fig. 7.

3.4. *The Datapath for the Maximum Probability Calculation*

After the conversions the numbers enter the second processing unit, which uses 5 partitioned adders, shown in Fig. 8. Those adders are organized in three stages, shown in Fig. 9 with the necessary registers between them. The data that comes from the SRAMs has to be restructured again before it feeds the adders. This is accomplished using again the system of the two FIFOs with different depths. The first stage produces the sums of the data that comes from the SRAMs in

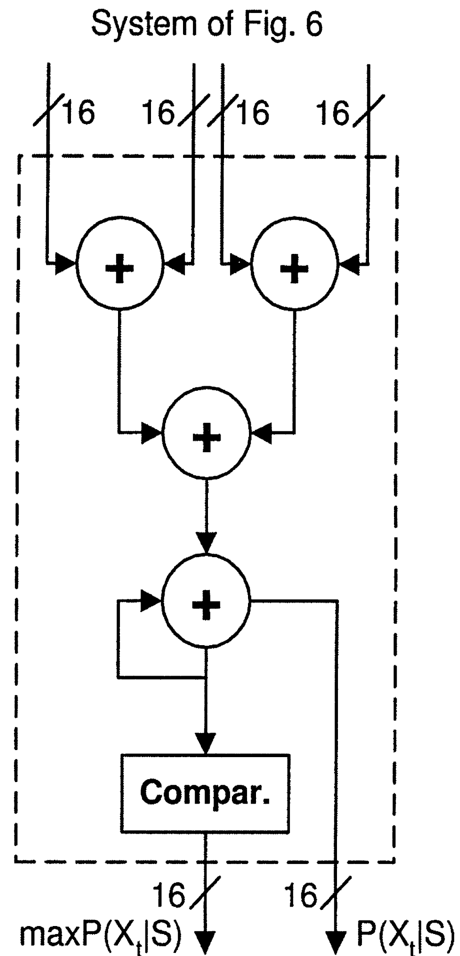


Figure 8. Adder tree and comparator structure to determine Max $P(X_t \mid S)$.

sets of two. Those sums are being added together in the second stage to give the sum of the 4 SRAM outputs. The third stage produces the output probability $P(X_t \mid S)$ by adding 8 sequential inputs. The intermediate sums are kept in a register, which contains the final sum. The next step is to find the maximum output probability for a given observation and the set of active states. For this purpose the register, which holds the output probabilities, is being compared with the register that holds the temporary maximum value. According to the output of this comparison the second register can be provided with the new or the previous maximum value through a multiplexer (shown in Fig. 8). Both the output probabilities and their maximum value feed a FIFO where they are merged in 64-bit words in order to be delivered to the host computer through the PCI bus. Figures 5 and 6, if put together in one, give

المنارة للاستشارات
www.manaraa.com

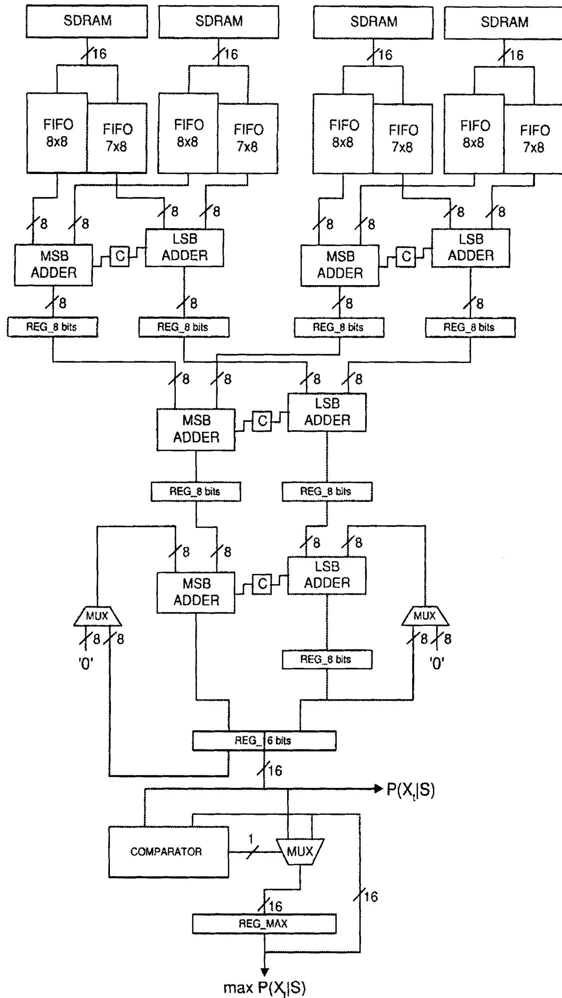


Figure 9. Datapath for adder tree and Max $P(X_t \mid S)$ determination.

the datapath of the architecture, which has 9 pipeline stages. Their control is being done by three Mealy-model FSMs [17, 18], which cooperate through control signals. The first controls the memory, the address generation and the interface with the PCI bus. The second controls the first processing unit. When the calculation of the P's finishes it sends a control signal to the third FSM, which controls the second processing unit in order to make it begin the procedure of producing output probabilities. The control is accomplished by loading the registers, the FIFOs, and the control bits of the multiplexers in both processing units. The adders do not have any control signals. All FSMs were designed to reduce the overhead cycles during the calculation of the output probabilities. According to the theoretical model, the number of operations needed in order to compute all the probabilities for an active genone that has N states and an observation vector X_t, is as follows:

- $(32 \times 15) + (32 \times N) = 480 + (32 \times N)$ memory accesses in order to get the data
- $32 \times 15 = 480$ additions for the calculation of $\prod_{j=1}^{15} P(X_{tj} \mid W_i, S)$
- $32 \times N$ additions for the calculation of $P(W_i \mid S) \prod_{j=1}^{15} P(X_{tj} \mid W_i, S)$
- $32 \times N$ accesses in the SRAM for the log-to-fixed point real number transformations
- $32 \times N$ additions for the calculation of $\sum_{i=1}^{32} P(W_i \mid S) \prod_{j=1}^{15} P(X_{tj} \mid W_i, S)$

This gives a total of $480 + (64 \times N)$ memory accesses and $480 + (64 \times N)$ additions. Using the capabilities of the architecture developed, this amount is achieved in $139 + (N \times 8)$ cycles including the wait states of the memory, from which 8 are needed for restructuring the FIFOs of the first processing unit, and 8 for the second unit input FIFOs to finish (the clearing of the pipeline at the end of the computation). The number of cycles each of the stages of the architecture is busy can be seen in Table 4.

In this table the effect of the pipeline and the parallelism can be seen very clear. The numbers show that the system has a constant number of wasted cycles, which does not depend on the number of active states.

Table 5 shows the number of cycles needed for all operations and for various values of N.

Table 4. Active cycles of the stages.

STAGE	# of active cycles
SDRAM	$120 + (8 \times N)$
First processing unit	$120 + (8 \times N)$
LUT (SRAM)	$(8 \times N)$
Second processing unit	$(8 \times N)$
Wasted cycles (wait states and FIFOs)	19

Table 5. Number of operations and cycles vs. number of N.

Value of N	Number of operations	Number of cycles	Speedup	μsec with 66 MHz clock
5	1280	179	7.15	2.71
10	1600	219	7.35	3.32
25	2560	339	7.55	5.14
40	3520	459	7.67	6.95
57	4608	595	7.74	9.02

للاستشارات المنارة
www.manaraa.com

If we compare this architecture with a different one which can perform one operation per cycle but has no parallelism and pipeline, even with the same adders, the architecture that was presented above is much faster for all the values of N as shown in Table 5. The comparison of the numbers gives an average speedup of 7.5. This speedup, although by coincidence is close to the speedup over running the algorithm in software, is in fact unrelated to the software execution of the algorithm, and it only reflects the effectiveness in the utilization of the system resources by the new architecture.

3.5. Reconfiguration Issues

The architecture which was described above, implements with great efficiency the modified Hidden Markov Model DMHMM shown in Section 2, with the specific parameters shown in Table 1. There are, however, issues which cannot be addressed in a traditional VLSI solution, but are readily addressable in this architecture due to the usage of reconfigurable hardware. The four most important of these issues are:

- State space encoding vs. depth of pipeline.
- Precision of arithmetic when computing the $P(W_i \mid S) \prod_{j=1}^{15} P(X_{tj} \mid W_i, S)$ products
- Vocabulary increase with SDRAM changes only
- Collection of statistics

The state space encoding vs. depth of pipeline referes to our choice of allocation of the 64-bit SDRAM to FPGA datapath. There is no constraint whatsoever that this datapath need to be broken down to four 16-bit quantities. We can arbitrarily change it to, e.g. three 20-bit quantities and leave four bits unused. If we do change the precision, we will need to change accordingly the pipeline depth of the addition datapath (which produces the sums of logarithms) in order to maintain a high addition throughput while the SDRAM is in burst mode. In this case the SDRAM subsystem need not be changed, but there will need to be some provision to keep the 16 MSB of the result for the lookup conversion from logarithm to integer fixed point.

The precision of the arithmetic when computing the $P(W_i \mid S) \prod_{j=1}^{15} P(X_{tj} \mid W_i, S)$ products (as sums) might need to be reconsidered. Indeed, if we consider that the arguments we add are 16-bit quantities and the result is also a 16-bit quantity, it is obvious that there might be overflow. In practice, however, it is understood that since the quantities in question correspond to probabilities that add up to one, and their representation is logarithmic, no overflow will occur. If we wanted to be conservative, we could hold the addition results in 20-bit registers, but then we would have one of two undesirable side effects: either the SRAM would become sixteen times larger (four more bits of address), or, if we only kept the sixteen MSB there would be a danger of using in practice only one quarter of the precision that the SRAM can offer us. As the architecture stands, it is *assumed* that in effect the most significant bits of the result are zeroes, and 16 bits suffice for the logarithm to fixed point real number conversion. Although simulations show that this is indeed the case, we may find in the future that there is indeed overflow. Due to the reconfigurable nature of the computational core, we can decide *what* is the desired precision, and *which* bits of the result should be used to address the SRAM, with no system level changes whatsoever. This aspect of the architecture alone signifies why FPGA's are not merely a rapid prototyping medium but also a means to change the implementation aspects of the algorithm through reconfiguration, without any hardware changes whatsoever. This form of possible reconfiguration is not dynamic, but is essential in circumventing problems which are normally associated with VLSI. Specifically, by allowing a change of the algorithm parameters, the precision of the arithmetic, and the structure of the pipeline, we can modify the operation of the system in ways that are not possible with VLSI implementations, even after the system has been produced and in operation.

Although it is obvious that with additional SDRAM we could increase the vocabulary, the organization of the information in the SRAM, as shown in Fig. 4 determines how the addresses need to be generated. The use of reconfigurable hardware means that such a design change is accomplished with a recompilation of the design according to the new desired sequencing.

Lastly, we have instrumented the computational core with statistics of interest. The potential for overflow in the additions when we compute the products is monitored, for the reasons that were stated above. Also, in order to help with algorithm development and state-space encoding, not only is the Maximum $P(X_t \mid S)$ returned to the host, but also all 32 potential maxima, i.e. all $P(X_t \mid S)$ that are returned from the SRAM. In practice that means that an utterance is recognized as one of 32 different possibilities. Alternative statistics,

المنارة للاستشارات
www.manaraa.com

including burst mode performance of the I/O and total utilization can be added too, while the system is operational.

The above examples regarding the usage of reconfigurable devices demonstrate why FPGA's are not merely a low cost prototyping alternative to VLSI design but they are a technology of choice in cases such as ours, in which the run-time characteristics of a system are not fully known. Whereas a lot more can be achieved through reconfiguration, and indeed this might be the case after the first generation of this system, for its first generation the use of FPGA's allows for a hardware platform which can achieve the desirable real time performance, while at the same time identifying through large scale field usage whether the chosen parameters of operation are optimal or not. Given that the algorithms evolve at a fairly rapid pace, it may turn out that the best solution in the long run is to evolve reconfigurable architectures rather than map a specific version into a VLSI implementation. The final choice will of course depend on issues such as volume of production, usage of the system (e.g. small number of large servers vs. a large number of plug-in cards), and the needs of the user community (a hardware system will have a larger vocabulary and a smaller WER than a software only system, but some version of the latter may be deemed "good enough" at some point).

4. Mapping the Architecture to Configurable Logic

One of the main targets of the architecture implementation was the flexibility and the design time. Those two facts, led to the use of configurable logic. The design was done using the ALTERA MAX+plus II ver. 9.0.1 CAD Tool for the design [19], and ALTERA FPGAs for the hardware implementation [20]. The architecture was described mainly in VHDL [21] and schematic capture.

4.1. Design of the Architecture

For this design two devices were selected: one for the SDRAM controller and one for the rest of the system. For the SDRAM controller a fast device (EPM7128SLC84-7) from the MAX7000S family of ALTERA EPLDs was selected [20, 22]. This solution was given because of the small propagation delays between inputs and outputs (7 nsec). For the rest of the system, which is the main architecture, the FLEX10KE was selected. Tables 6 and 7, below, show the numbers of inputs and outputs the architecture uses. The control inputs come mainly from the SDRAM controller.

Table 6. Signal pins of the main FPGA of the system.

Signal name	Type of signal	Number of pins
Data	Input	128
Control	Input	5
Data	Output	80
Control	Output	8
TOTAL	Input/output	221

Table 7. Use of FPGA resources.

Kind of resource	EPF10K50EFC484-1	Used by architecture
Input/output pins	248	215
Logic cells	2880	2507

This design fits in a single EPF10K50EFC484-1 device. This device has 248 I/O pins and 2880 logic cells. The architecture implemented uses the resources offered by this device efficiently, as shown in Table 7. As this table shows this implementation gives a fine utilization of the resources offered by the specific device. The operation frequency of the design is 70,92 Mhz. This leads to a system that can run with a clock of 66 Mhz (the PCI clock). For the design parameters given in Section 2, the processing time for a set of $[S, X_t]$, expressed in time and not in cycles, and for various numbers of active states is shown in Table 5.

The SDRAM is a single 64Mbyte DIMM module, organized internally as four interleaved banks, its datapath is 64 bits (thus the depth is 8Mwords) [23, 24]. The speed grade we have used is 100 MHz, and the part is exactly as used in personal computers. The SRAM module has been designed out of eight $64K \times 8$ SRAM chips, organized as four independently addressable banks of $64K \times 16$ bits each. In order to have the required speed, the chips that were used are 15 nsec SOJ package parts [25]. If we consider the usage of resources, the SRAM is 100% used, whereas the SDRAM is used 51% (this is the percentage of locations that contain useful information).

The statistical analysis of the recognizer implemented only in software, with a sample 100 recognized sentences, shows that with a sampling rate of 100 Hz, which gives one observation vector every

المنارة للاستشارات

www.manaraa.com

Table 8. SDRAM requirements on board.

Model	Number of states	State memory (MBytes)	Number of genones	Genone memory (MBytes)	Total memory (MBytes)
ATIS	10,872	0.7	1,105	32.3	64
Nuance v6.0	36,000	2.3	829	24.9	32
Nuance v6.2					
Single pass	42,000	2.7	926	27.8	64
Two pass	28,000	1.8	1,774	53.2	64

10 msec, the average number of active genones is 522, and the average number of active states per genone is 10. From those facts the average time available for the computation and I/O for one set of $[S, X_t]$ is 10 msec/522 = 19.15 μsec. The same computation can be done in the new architecture, described in this paper, in 3.32 μsec, leaving more than 15 μsec for system I/O. In this case the PCI passes to the coprocessor system seven 64-bit words and receives three 64-bit words. Even in the worst case scenario with all 57 states being used in each case, the processing time for the new architecture would become 9.02 μsec for processing. Given that the recognizer front end produces 100 observations/second for each channel, the above figures mean that under worst case conditions we have the capacity for two real-time channels and in a typical case we have the capacity for six real-time channels. By contrast, a 266 Pentium II PC has the capacity (for this part alone) for 0.7 real-time channels for the typical case, whereas the worst case has not been measured because it has not been observed. Therefore, the architecture is roughly 9 times faster than a personal computer for the corresponding part of the load, and given that I/O is done at 66 MHz (whereas the present design can operate at 71 MHz).

The number of real-time channels, two to six for this specific design, are useful for large server-type applications. Indeed, because the architecture is oblivious to the original source of the computation, can be partitioned among channels as long as the data are sent in sets of one observation and its active state space.

4.2. *The Effect of the Application on the SDRAM Subsystem*

In order to see the effects of alternative models on the SDRAM requirements of the system we have tabulated these requirements for several real-world applications: ATIS is the model that was used for the development of the architecture. In addition, we present data for a commercial system, namely, that of Nuance, and for versions 6.0 and 6.2 (both single and two pass). We note that all of the above systems can employ the existing 64 Mbytes of SDRAM, whereas the Nuance v6.0 system could very well fit in a board with 32 MBytes of SDRAM.

The calculations for the states were made as follows: each state has the same number of bytes. Specifically, each state has 32 Gaussians, and the mixture weight for each of them is represented by 2 bytes, meaning that each state requires 64 bytes.

Similarly, each genone has 32 Guassians and each Guassian is partitioned into 15 subvectors which are represented by 5-bits each, thereby requiring a probability vector of length 32 (of 2 bytes each, i.e. 64 bytes). The total product of these terms in $32 \times 15 \times 32 \times 2 =$ 30 KBytes. Therefore the total memory requirements can be determined by multiplying the number of genones by 30 KBytes and the number of states by 64 bytes. Whereas this total represents actual memory requirements, efficient encoding is not always possible, and as result, even in cases which would nominally fit in 32 MBytes of SDRAM, 64 MBytes might be required. A similar situation occurs with the ATIS application which requires slightly over 32 MBytes, thus necessitating 64 MBytes only 51% of which are used.

This memory calculation can also be used to demonstrate why reconfigurability of the system offers advantages which a VLSI implementation cannot have. Specifically, each time that we change the genones and the states, the *address bits* for these change too, which means that memory usage is tailored to the specific characteristics of the model which is used. Referring again to Fig. 4 we see how the address is formed according to where the genones and the states are. The ability to alter the organization of the SDRAM subsystem with a corresponding change of the address generation logic effictively mandates reconfigurability in the system.

المنارة للاستشارات
www.manaraa.co

4.3. PCI and User Application Performance Issues

In the design of a real time system, issues like the bus performance and application software performance are of major importance. It is noteworthy that in order to achieve real-time performance at the human scale, all processing has to be done *on the average* for the recognition of a sentence at the observation rate of 100 samples/sec, and the processing of an individual observation may deviate somewhat from this rate. It is thus permissible to use the average figure of 522 genones per observation in our calculation of I/O and application level processing (it turns out that we can meet worst case figure too, but this has not been experimentally observed). For each of these observations we have three elements of computation: the preprocessing, the search, and the grammar processing. As shown in Fig. 1, the search dominates the load with roughly 87% of the load in the case of the least error rate, and this is the aspect which gets sped up by our engine. All genones per observation are known and can be passed on to the coprocessor. Likewise, the results from each $[S, X_t]$ calculation can be buffered before they are passed to the processor.

For the remaining analysis we have used the *experimental* data on PCI bus performance measurments, which were reported by Moll and Shand [14], and specifically results derived from a 200 MHz Pentium Pro system with 32-bit 33 MHz PCI bus. We consider these figures to be very conservative in order to demonstrate that the real time performance of our system is not limited by I/O operations.

During a 10 msec period we write to the coprocessor $522 \times 10 \times 7 \times 8$ bytes = 292,320 bytes. The time needed to write all genones and active states through the PCI is calculated with the minimum measured DMA rate reported by Moll and Shand [14] of 129 Mbytes/sec. Although the initial latency is not reported, a conservative minimum is 100 μsec which corresponds to two system calls plus the time to buffer the I/O (in general, writes to the PCI are much faster than reads). For the 292,320 bytes passed to the coprocessor we thus need 2.16 msec transfer time and 0.1 msec initial latency, or 2.26 msec total time. The processing time for all these active states is 522×3.32 μsec = 1.73 msec.

The cost of a PCI bus transaction (PCI card interrupting and passing data to user process) has been experimentally determined by Moll and Shand [14] to be 30 μsec for the interrupt service routine and less than 200 μsec for the user thread in 98% of the cases in loaded systems for personal computers with 200 MHz Pentium Pro processors and 33 MHz 32-bit PCI bus. Therefore we consider these figures to be conservative. The same authors also report a *minimum* of 110 Mbytes/sec read rate (using DMA), which means that in the average case of 522 genones with an average number of 10 states each, and for which we return three 64-bit words (a total of 125,280 bytes) would require (in addition to the 200 μsec latency) 1.09 msec, bringing the total read time (including the interrupt overhead) to 1.29 msec.

From the above figures we conservatively estimate that with a 33 MHz 32-bit PCI bus and a 200 MHz Pentium Pro processor we need 2.26 msec + 1.73 msec + 1.29 msec = 5.28 msec. From Fig. 1 we see that even without the I/O cost the same aspect of the computation would take $0.87 \times 2.2 \times 10$ msec = 19.1 msec (here, the 2.2 factor over the required real time performance is applied on the 10 msec time budget for the calculation).

We now need to see what is the grammar and front end processing time budget. Again from Fig. 1 we see that the aspect which we now perform on software is $0.13 \times 2.2 \times 10$ msec = 2.86 msec for the grammar processing. The front end processing does not need to enter into this computation because it does not depend on the results of the coprocessor, and thus can be performed in parallel with the coprocessor performing the search. Thus, the critical path of the calculation, namely the search and the grammar processing require (under very conservative calculations) 5.28 msec + 2.86 msec = 8.14 msec which is within the 10 msec time budget for real time performance.

The real time performance estimates were calculated using the simplistic model of downloading all active genones and states, *then* doing all the calculations, *then* uploading all results, and *after* all results are loaded do the grammar processing. We do not suggest that this is by any means a desirable way to operate the coprocessor, because it makes poor usage of its resources, it necessitates large I/O buffers, and it uses poorly the CPU. In practice, the granularity of the I/O will be determined by actual system performance. It is clear that the data passed back and forth has to be for more than one state, since the 200 μsec interrupt-to-application process latency of the PCI bus alone is roughly ten times the 19.15 μsec processing window of one state. Given that each genone has a different number of active states associtated with it, it is desirable to send to the coprocessor a fixed number of states at a time. The only reason

المنارة للاستشارات
www.manaraa.com

why we performed the calculations in this manner was to demonstrate that even with conservative technology the real time requirements can be met with plenty of time left for the PC to run other applications too.

5. Present Status and Future Work

The present status of this project is the construction of an actual engine to perform DMHMM based continuous-speech recognition. Due to the difficulty in the acquisition of miscellaneous components (mostly, BGA sockets), the construction is done with more conservative technology, i.e. with multiple 84-pin PLCC packaged FPGA's instead of the larger model described in this paper. The design was readily partitioned among the smaller FPGA's by taking slices of the computation with the finite state machine controlling them into single FPGA's. The corresponding performance degradation of the system (as determined by post place and route times and calculated inter-chip communication delays) makes it appropriate for 33 MHz I/O rather than 66 MHz I/O, which means that even the smaller system will be able to handle multiple channels in real time. A small prototype board to prove the concept was constructed with one 84-pin EPF10K10LC84-4 ALTERA FPGA. On that board, *all* portions of the datapath and their corresponding control structures were tested, one at a time. The tests were performed at the full 33 MHz speed of the scaled down system, and all subsystems were found to be fully functional.

Due to the difficulty in acquiring PCI design cores for direct inferfacing to the PCI bus, high speed I/O will be performed with the COMPAQ Labs PAMETTE board, a copy of which has already been obtained on loan. One issue which has been studied but needs to be evaluated experimentally too is the I/O speed of the host system. Because speech recognition is tolerant to some initial latency (even a few tenths of a second), it is the *bandwidth* that will be of importance rather than the latency in accessing the coprocessor board, and with burst mode in PCI this is expected to be no problem.

In conclusion, an architecture has been presented such that with a single board system we can achieve multichannel real-time continuous speech recognition using the DMHMM algorithm. Such an architecture can be adopted to specific solutions of the DMHMM algorithm for different state space and processing constraints by simple design changes and reconfiguration of the computational core FPGA. This architecture can operate as a coprocessor for a personal computer which will perform the sampling, Fourier analysis, and grammar extraction using roughly half of its processing power. It is notable that even high-end PC's fully devoted to the task *cannot* perform real-time DMHMM continuous-speech recognition, whereas the improvement gained by using faster processors is not commensurate to the clock speed improvement (see previous section).

Following the implementation of this architecture, the system will be reconfigured so that it will be evaluated for a variety of state assignments, vocabulary sizes, and precision of computation, whereas a larger system might be implemented as a large server for multichannel applications.

Acknowledgments

The authors would like to acknowledge Laurent Moll and Mark Shand of COMPAQ Labs for the loan of a PAMETTE system, and, Stephen Smith and Joe Hanson of ALTERA Corporation for ALTERA's support with CAD tools and parts.

Note

1. The Viterbi algorithm is an instance of Dynamic Programming that is also used in communications for the decoding of convolutional codes.

References

1. Speech Technology is the Next Big Thing in Computing. *Business Week*, February 23, 1998.
2. H. Murveit, "An Integrated Circuit Based Speech Recognition System," Ph.D. Thesis, Department of Electrical Engineering, University of California at Berkeley, 1983.
3. S. Narayanaswamy, "Pen and Speech Recognition in the User Interface for Mobile Multimedia Terminals," Ph.D. Thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 1996.
4. S. Hauck, "The Roles of FPGA's in Reprogrammable Systems," *Proceedings of the IEEE*, April 1998, pp. 615–637.
5. H. Schmit and D. Thomas, "Hidden Markov and Fuzzy Controllers in FPGAs," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Computer Society Press, 1995, pp. 214–221.
6. D. Yeh, G. Feygin, and P. Chow, "RACER: A Reconfigurable Constraint-Length 14 Viterbi Decoder," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Computer Society Press, 1996, pp. 60–69.
7. S.J. Young, "A Review of Large-Vocabulary Continuous-Speech Recognition," *IEEE Signal Processing Magazine*, September 1996, pp. 45–57.
8. S.B. Davis and P. Mermelstein, "Comparison of Parametric Representations for Monosyllabic Word Recognition in

المنارة للاستشارات
www.manaraa.com

Continuously Spoken Sentences," *IEEE Trans. Acoustics Speech and Signal Processing*, vol. ASSP-28, no. 4, 1980, pp. 357–366.
9. V. Digalakis, P. Monaco, and H. Murveit, "Genones: Generalized Mixture Tying in Continuous Hidden Markov Model-Based Speech Recognizers," *IEEE Trans. Speech Audio Processing*, 1996, pp. 281–289.
10. F. Jelinek, *Statistical Methods for Speech Recognition*, MIT Press, 1997.
11. V. Digalakis, L. Neumeyer, and M. Perakakis, "Quantization of Cepstral Parameters for Speech Recognition Over the WWW," *IEEE Journal on Selected Areas in Communications*, 1999, pp. 82–90.
12. S. Tsakalidis, V. Digalakis, and L. Neumeyer, "Efficient Speech Recognition Using Subvector Quantization and Discrete-Mixture HMMs," *IEEE International Conference on Acoustics, Speech and Signal Processing*, submitted.
13. P. Price, "Evaluation of Spoken Language Systems: The ATIS Domain," *Proceedings of the Third DARPA Speech and Natural Language Workshop*, Morgan Kaufmann, June 1990.
14. L. Moll and M. Shand, "Systems Performance Measurement on PCI Pamette," *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, Computer Society Press, 1997, pp. 125–133.
15. Intel Corporation, *PC SDRAM Specification ver. 1.51*, 1997.
16. Digital Equipment Corporation, *PCI Pamette user-area Interface for Firmware v1.9*, 1997.
17. J.F. Wakerly, *Digital Design Principles and Practices*, Prentice Hall, 1990.
18. R.H. Katz, *Contemporary Logic Design*, The Benjamin/Cummings Publishing Com., 1994.
19. Altera Corporation, *ALTERA MAX+PLUS II VHDL*, Altera Corporation 101 Innovation Drive, San Jose, CA, USA, 1996.
20. Altera Corporation, *ALTERA Data Book 1998*, Altera Corporation 101 Innovation Drive, San Jose, CA, USA, 1998.
21. J.M. Berge, A. Fonkoua, S. Maginot, and J. Rouillard, *VHDL Designer's Reference*, Kluwer Academic Publishers, 1992.
22. Altera Corporation, *FLEX 10KE Embedded Programmable Logic Family*, Altera Corporation 101 Innovation Drive, San Jose, CA, USA, 1998.
23. IBM Corporation, *168 Pin SDRAM Registered DIMM Functional Description & Timing Diagrams*, 1998.
24. IBM Corporation, *8M x 64/72 Bank Unbuffered SDRAM Module*, 1998.
25. SAMSUNG Electronics, *KM68257C 32Kx8 Bit High Speed Static RAM(5V Operating)*, February 1998.

Panagiotis Stogiannos was born in 1974 in Corfu, Greece. He got his diploma in Electronic and Computer Engineering in 1997, and his MS degree in Computer Engineering in 1999 from the ECE Dept. of the Technical University of Crete. At present he is pursuing his PhD degree at the same Department and he is employed in Intracom in Athens, Greece. His research interests include: design and implementation of embedded systems, design of application specific coprocessors using reconfigurable logic and design of real-time systems.

Apostolos Dollas received his B.S., M.S., and Ph.D. degrees in Computer Science from the University of Illinois at Urbana-Champaign in 1982, 1984 and 1987 respectively. From 1986 until 1994 he was Assistant Professor of Electrical Engineering and Assistant Professor of Computer Science at Duke University. Since 1993 he is Associate Professor of the Electronic and Computer Engineering Department at the Technical University of Crete in Chania, Greece, where he is also the Director of the Microprocessor and Hardware Laboratory. He has served one term as Department Chairman (1995–1997) and one term as Associate Department Chairman (1997–1999). He conducts research and teaches in the areas of rapid system prototyping, computer architecture, reconfigurable computing, embedded systems, and application specific high-performance digital systems. In all of these areas Dollas places emphasis on the development of fully functional prototypes.

Dollas is a senior member of IEEE and the IEEE Computer Society. He belongs to the Eta Kappa Nu and Tau Beta Pi and has received the IEEE Computer Society Golden Core Member award, the IEEE Computer Society Meritorious Service Award, and twice the Department of Computer Science Award for Outstanding Teaching at the University of Illinois at Urbana-Champaign.

Vassilios V. Digalakis was born in Chania, Greece, on February 2, 1963. He received the Diploma in electrical engineering from the National Technical University of Athens, Greece, in 1986, the M.S. degree in electrical engineering from Northeastern University, Boston, MA, in 1988, and the Ph.D. degree in electrical and systems engineering from Boston University, Boston, MA, in 1992.

المنارة للاستشارات

www.manaraa.com

From January 1992 to February 1995 he was with the Speech Technology and Research Laboratory of SRI International in Menlo Park, CA. At SRI, he was a principal investigator for ARPA research contracts and he developed new speech recognition and speaker adaptation algorithms for the DECIPHER speech recognition system. He also developed language education algorithms using speech recognition techniques.

He is currently with the department of Electronic and Computer engineering of the Technical University of Crete in Chania, where he holds an assistant professor position. He teaches undergraduate and graduate courses on Speech Processing and on Digital and Analog Communications. His research interests are in pattern and speech recognition, information theory and digital communications.

المنارة للاستشارات

www.manaraa.co

Journal of VLSI Signal Processing Systems 24, 241–262, 2000.
© 2000 Kluwer Academic Publishers. Printed in The Netherlands.

The CAM-Brain Machine (CBM): Real Time Evolution and Update of a 75 Million Neuron FPGA-Based Artificial Brain

HUGO DE GARIS

Evolutionary Systems Dept., ATR—Human Information Processing Research Laboratories, 2-2 Hikari-dai, Seika-cho, Soraku-gun, Kyoto 619-0288, Japan

MICHAEL KORKIN

Genobyte, Inc., 1503 Spruce Street, Suite 3, Boulder CO 80302, USA

Abstract. This article introduces ATR's "CAM-Brain Machine" (CBM), an FPGA based piece of hardware which implements a genetic algorithm (GA) to evolve a cellular automata (CA) based neural network circuit module, of approximately 1,000 neurons, in about a second, i.e. a complete run of a GA, with 10,000 s of circuit growths and performance evaluations. Up to 65,000 of these modules, each of which is evolved with a humanly specified function, can be downloaded into a large RAM space, and interconnected according to humanly specified artificial brain architectures. This RAM, containing an artificial brain with up to 75 million neurons, is then updated by the CBM at a rate of 130 billion CA cells per second. Such speeds should enable real time control of robots and hopefully the birth of a new research field that we call "brain building". The first such artificial brain, to be built by ATR starting in 2000, will be used to control the behaviors of a life sized robot kitten called "Robokoneko".

1. Introduction

This article introduces ATR's "CAM-Brain Machine" (CBM) [1], a Xilinx XC6264 FPGA [2] based piece of hardware that is used to evolve 3D cellular automata based neural network [3] circuit modules at electronic speeds, that is in about a second per module. 65,000 of these modules can then be assembled into a large RAM space according to humanly specified artificial brain architectures. This RAM is updated by the CBM fast enough (130 billion CA cell updates/sec) for real time control of robots. ATR's CBM should be built and delivered by the fourth quarter of 1999.

The CBM is the essential tool in ATR's "Artificial Brain (CAM-Brain) Project" [4, 5], which at the time of writing (Summer 1999), has been running for 6.5 years. Although the focus of this article is on the functional principles and design of the CBM, a certain background needs to be provided so that the motivation for its construction is understood.

The basic (and rather ambitious) aim of the CAM-Brain Project as first stated in 1993 was to build an artificial brain containing a billion artificial neurons by the year 2001. The actual figure in 1999 will be maximum 75 million, but the billion figure is still reachable if we really want. The ATR Brain Builder team is hoping that the CBM will revolutionize the field of neural networks (by creating neural systems with tens of millions of artificial neurons, rather than just the conventional tens to hundreds), and will create a new research field called "Brain Building". The CBM will make practical the creation of artificial brains, which are defined to be assemblages of tens of thousands (and higher magnitudes) of evolved neural net modules into humanly defined artificial brain architectures. An artificial brain will consist of a large RAM memory space, into which individual CA modules are downloaded once they have been evolved. The CA cells in this RAM will be updated by the CBM fast enough for real time control of a robot kitten "Robokoneko" (Japanese for "robot kitten").

Since the neural net model used to fit into state-of-the-art evolvable electronics has to be simple, the signaling states of the neural net were chosen to be 1 bit

www.manaraa.com

binary. We label this model "CoDi-1Bit" [6] (CoDi = Collect & Distribute). This article will summarize the principles of this 1 bit neural signaling model, since the CBM is an electronic implementation of it. We realize that limiting ourselves to only 1 bit per neural signal (to fit into the Xilinx XC6264 chips), is rather severe (although nature uses a 1 bit signal scheme with its evoked potentials, i.e. the spikes in the axons), so it is possible that future versions of the CBM may use multibit neural signaling to obtain higher "evolvability" of neural module functionality.

The remainder of this article is structured as follows. Section 2 gives an explanation of the "CoDi-1Bit" neural net model that is implemented by the CAM-Brain Machine (CBM). Section 3 discusses briefly the representation that our team has chosen to interpret the 1 bit signals which are input to and output from the CoDi modules (we call this representation "SIIC" = Spike Interval Information Coding). This representation is important because the CBM measures the "fitness" (i.e. the performance measure of the evolving circuit) using analog output values obtained by convolving the binary outputs of the module with a digitized convolution function. Section 4 shows how analog time-dependent signals can be converted into spike trains (bit strings of 0 s and 1 s) to be input into CoDi modules using the so-called "HSA" (Hough Spiker Algorithm). The SIIC (spiketrain to analog signal conversion) and the HSA (analog signal to spiketrain conversion) allow users (evolutionary engineers) to think entirely in analog terms when specifying input signals and target (desired) output signals, which is much easier than thinking in terms of spike intervals (the number of 0 s between the 1 s). This analog thinking for evolutionary engineers simplifies the evolution of modules, and overcomes the limitation to some extent of the 1 bit binary signaling of the CoDi modules (and hence the CBM). Section 5, the heart of this article, provides a detailed summary of CBM design and functionality, using the ideas already discussed in the earlier sections. Since an artificial brain without a body (such as a robot) seems rather pointless, Section 6 introduces early work on the behavioral repertoire and mechanical design of the kitten robot "Robokoneko" that our artificial brain will control. Section 7 presents a (software simulated) sample of what evolved CoDi modules will be able to do, once the CBM is complete and delivered. Our Brain Builder team will then be evolving thousands of such modules. Section 8 discusses ideas for interesting future modules and multi-module systems to be evolved. Section 9 talks about some related work, and Section 10 concludes.

2. The CoDi-1Bit Neural Network Model

The CBM implements the so called "CoDi" (i.e. Collect and Distribute) [6] cellular automata based neural network model. It is a simplified form of an earlier model developed at ATR (Kyoto, Japan) in the summer of 1996, with two goals in mind. One was to make neural network functioning much simpler and more compact compared to the original ATR model, so as to achieve considerably faster evolution runs on the CAM-8 (Cellular Automata Machine), a dedicated hardware tool developed at Massachusetts Institute of Technology in 1989.

In order to evolve one neural module, a population of 30–100 modules is run through a genetic algorithm [7] for 200–600 generations, resulting in up to 60,000 different module evaluations. Each module evaluation consists of—firstly, growing a new set of axonic and dendritic trees, guided by the module's chromosome (which provide the growth instructions for the trees). These trees interconnect several hundred neurons in the 3D cellular automata space of 13,824 cells ($24 \times 24 \times 24$). Evaluation is continued by sending spiketrains to the module through its afferent axons (external connections) to evaluate its performance (fitness) by looking at the outgoing spiketrains. This typically requires up to 1000 update cycles for all the cells in the module.

On the MIT CAM-8 machine, it takes up to 69 minutes to go through 829 billion cell updates needed to evolve a single neural module, as described above. A simple "insect-like" artificial brain has hundreds of thousands of neurons arranged into ten thousand modules. It would take 500 days (running 24 hours a day) to finish the computations.

Another limitation was apparent in the full brain simulation mode, involving thousands of modules interconnected together. For a 10,000-module brain, the CAM-8 is capable of updating every module at the rate of one update cycle 1.4 times a second. However, for real time control of a robotic device, an update rate of 50–100 cycles per module, 10–20 times a second is needed. So, the second goal was to have a model which would be portable into electronic hardware to eventually design a machine capable of accelerating both brain evolution and brain simulation by a factor of 500 compared to CAM-8.

المنارة للاستشارات
www.manaraa.com

The CoDi model operates as a 3D cellular automata (CA). Each cell is a cube which has six neighbor cells, one for each of its faces. By loading a different phenotype code into a cell, it can be reconfigured to function as a neuron, an axon, or a dendrite. A neuron is a brain cell. An axon is the branching of a neuron which carries a neural signal away from the neuron to other neurons. A dendrite is the branching of the neuron which carries a neural signal towards the neuron from other neurons. Neurons are configurable on a coarser grid, namely one per block of $2 \times 2 \times 3$ CA cells. Cells are interconnected with bidirectional 1-bit buses and assembled into 3D modules of 13,824 cells ($24 \times 24 \times 24$).

Modules are further interconnected with 188 1-bit connections to function together as an artificial brain. Each module can receive signals from up to 188 other modules and send its output signals to up to 64,640 modules. These intermodular connections are virtual and implemented as a cross-reference list in a module interconnection memory (see below).

In a neuron cell, five (of its six) connections are dendritic inputs, and one is an axonic output. A 4-bit accumulator sums incoming signals and fires an output signal when a threshold is exceeded. Each of the inputs can perform an inhibitory or an excitatory function (depending on the neuron's chromosome) and either adds to or subtracts from the accumulator. The neuron cell's output can be oriented in 6 different ways in the 3D space. A dendrite cell also has five inputs and one output, to collect signals from other cells. The incoming signals are passed to the output with an 5-bit XOR function. An axon cell is the opposite of a dendrite. It has 1 input and 5 outputs, and distributes signals to its neighbors. The "Collect and Distribute" mechanism of this neural model is reflected in its name "CoDi". Blank cells perform no function in an evolved neural network. They are used to grow new sets of dendritic and axonic trees during the evolution mode.

Before the growth begins, the module space consists of blank cells. Each cell is seeded with a 6-bit chromosome. The chromosome will guide the local direction of the dendritic and axonic tree growth. Six bits serve as a mask to encode different growth instructions, such as grow straight, turn left, split into three branches, block growth, T-split up and down etc. Before the growth phase starts, some cells are seeded as neurons under genetic control. As the growth starts, each neuron continuously sends growth signals to the surrounding blank cells, alternating between "grow dendrite" (sent in the direction of future dendritic inputs) and "grow axon" (sent towards the future axonic output). A blank cell which receives a growth signal becomes a dendrite cell, or an axon cell, and further propagates the growth signal, being continuously sent by the root neuron, to other blank cells. The direction of the propagation is guided by the 6-bit growth instruction, described above. This mechanism grows a complex 3D system of branching dendritic and axonic trees, with each tree having one neuron cell associated with it. The trees can conduct signals between the neurons to perform complex spatio-temporal functions. The end-product of the growth phase is a phenotype bitstring which encodes the type and spatial orientation of each cell.

Thus there are two main phases—neural net growth and neural net signaling. In the CoDi-1Bit model, the signal states contain only 1 bit. With an 8 bit signal for example (as was the case in the old CAM-Brain Project model) one simply looks at the signal state to see the signal value. With 1 bit signaling, one needs to choose an interpretation of the signals, e.g. frequency based (count the number of spikes (1 s) in a given time), or interpret the spacing between the spikes as containing information etc. These interpretation issues will be taken up in the next section.

3. The Spike Interval Information Coding Representation, "SIIC"

3.1. Choosing a Representation for the CoDi-1Bit Signaling

The constraints imposed by state-of-the-art programmable (evolvable) FPGAs in 1998 were such that the CA based model (the CoDi model) had to be very simple in order to be implementable within those constraints. Consequently, the signaling states in the model were made to contain only 1 bit of information (as happens in nature's "binary" spike trains). The problem then arose as to interpretation. How were we to assign meaning to the binary pulse streams (i.e. the clocked sequences of 0 s and 1 s which are a neural net module's inputs and outputs? We tried various ideas such as a frequency based interpretation, i.e. count the number of pulses (i.e. 1 s) in a given time window (of N clock cycles). But this was thought to be too slow. In an artificial brain with tens of thousands of modules which may be vertically nested to a depth of 20 or more (where the outputs of a module in layer n get fed into a module in layer $n + 1$, where n may be as large as

المنارة للاستشارات

www.manaraa.com

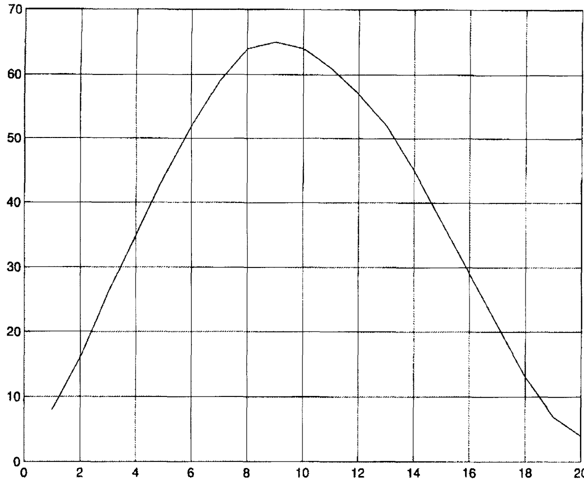


Figure 1. The convolution function used in the "SIIC" representation.

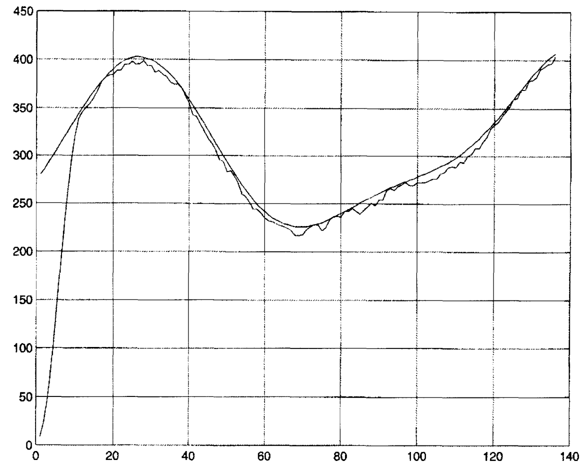


Figure 2. An analog (smooth) curve and its deconvolved/convolved approximation (jerky) curve.

20 or 30) then the cumulative delays may end up in a total response time of the robot kitten being too slow (e.g. if you wave your finger in front of its eye, it might react many seconds later). We wanted a representation that would deliver an integer or real valued number at each clock tick, the ultimate in speed. The first such representation we looked at we called "unary". If N neurons on an output surface are firing at a given clock tick, then the firing pattern represented the integer N, independently of where the outputs were coming from. We found this representation to be too stochastic, too jerky. Ultimately we chose a representation which convolves the binary pulse string with the convolution function shown in Fig. 1. We call this representation "SIIC" (Spike Interval Information Coding) which was inspired by [8].

This representation delivers a real valued output at each clock tick, thus converting a binary pulse string into an analog time dependent signal. Our team has already published several papers on the results of this convolution representation work [9]. Figure 2 shows the result of deconvolving an arbitrary analog curve (that is, converting an analog signal into a spike train (binary string) as explained in Section 4), and then convolving it back (i.e. converting a spike train into an analog signal) to the original analog curve. The smooth curve is the original curve, and the spikey curve is the result of the two conversions. The percentage errors obtained between the original curve and the result of the two conversions were only about 2%, so we thought these two conversions were very useful. Of course, it is one thing to have accurate conversions from analog signals to spike trains and vice versa. It is another that a CoDi-1Bit neural net module can evolve a spike train that when convolved can produce a desired analog output. Figure 3 shows just such an example (of a target 3 period sine curve) which evolved quite successfully, showing that the basic idea is sound. (The solid curve is the target curve, and the dashed curve is the evolved and convolved result. The actual spikes (i.e. the 1 s in the binary string output from the CoDi module) are shown beneath the curves.) Figure 4 shows two outputs of a "halver" circuit which was evolved to take a constant analog input (e.g. 600 or 400) and to output half its value (300 or 200). This case is a good example of how an evolutionary engineer can think entirely in analog terms when evolving modules. The analog input is automatically converted to a spike train, which enters the neural net module, and the spike train output of the module get automatically converted to an analog signal whose values are compared with a target curve to evaluate the fitness (performance) of the evolving circuit. Further examples of evolved modules (although using only binary I/O), are to be found in Section 7.

3.2. *The SIIC Convolution Algorithm*

The convolution algorithm we use takes the output spiketrain (a bit string of 0 s and 1 s), and runs the pulses (the 1 s) by the convolution function shown in the simplified example below. The output at any given time t is defined as the sum of those samples of the convolution filter that have a 1 in the corresponding spiketrain positions. The example below should clarify what is meant by this.

المنارة للاستشارات
www.manaraa.com

Figure 3. A 3 period sine curve resulting from convolution of an evolved CoDi-1Bit. The lower figure shows the actual spikes that generated the waveform.

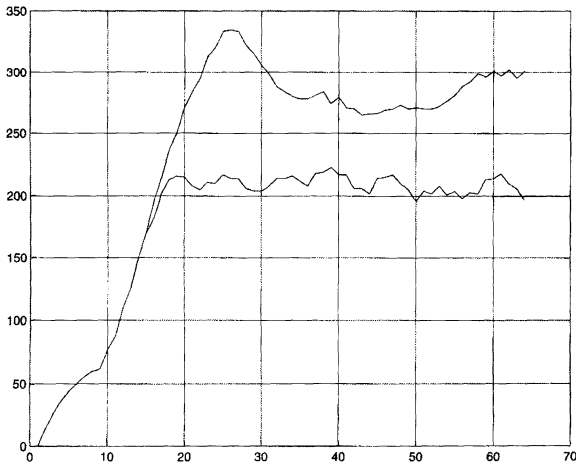


Figure 4. Outputs of a halver circuit (with inputs 600 and 400) using fully analog I/O.

Simplified Example. Convolve the spiketrain 1101001 (where the left most bit is the earliest, the right most bit, the latest) using the convolution filter values {1 4 9 5 −2}. The spiketrain in this diagram moves from left to right across the convolution filter.

Alternatively, one can view the convolution filter (window) moving across the spiketrain. The number to the right of the colon shows the value of the convolution sum at each time t.

```
time-shifted spike train :  1 0 0 1 0 1 1
  ---> (moves left to right)
convolution filter :  1 4 9 5 -2

  1 0 0 1 0 1 1
              0 0 0 0  0 :  0  t = -1

    1 0 0 1 0 1 1
              1 0 0 0  0 :  1  t = 0

      1 0 0 1 0 1 1
              1 4 0 0  0 :  5  t = 1

        1 0 0 1 0 1 1
              0 4 9 0  0 : 13  t = 2

          1 0 0 1 0 1 1
              1 0 9 5  0 : 15  t = 3
```

المنارة للاستشارات

www.manaraa.com

```
1 0 0 1 0 1  1
    0 4 0 5 -2 :  7   t = 4

  1 0 0 1 0  1
    0 0 9 0 -2 :  7   t = 5

    1 0 0 1  0
    1 0 0 5  0 :  6   t = 6

      1 0 0  1
    0 4 0 0 -2 :  2   t = 7

        1 0  0
    0 0 9 0  0 :  9   t = 8

          1  0
    0 0 0 5  0 :  5   t = 9

             1
    0 0 0 0 -2 : -2   t = 10
```

Hence, the time-dependent output of the convolution filter takes the values (0, 1, 5, 13, 15, 7, 7, 6, 2, 9, 5, −2). This is a time varying analog signal, which is the desired result.

4. The "Hough Spiker Algorithm" (HSA) for Deconvolution

Section 3 above explained the use of the SIIC (Spike Interval Information Coding) Representation which provides an efficient transformation of a spike train (string of bits) into a (clocked) time varying "analog" signal. We need this interpretation in order to interpret the spike train output from the CoDi modules to evaluate their fitness values (by comparing the actual converted analog output waveforms with user specified target waveforms). However, we also need the inverse process, namely, an algorithm which takes as input, a clocked (digitized, binary numbered) time varying "analog" signal, and outputs a spike train. This conversion is needed as an interface between the motors/sensors of the robot bodies (e.g. a kitten robot) that the artificial brain controls, and the brain's CoDi modules. However, it is also very useful to users, i.e. evolutionary engineers to be able to think entirely in terms of analog signals (at both the inputs and outputs) rather than in abstract, visually unintelligible spiketrains. This will make their task of evolving many CoDi modules much easier. We therefore present next an algorithm which is the opposite of the SIIC, namely one which takes as input, a time varying analog signal, and outputs a spiketrain, which if later is convolved with the SIIC convolution filter, should result in the original analog signal.

A brief description of the algorithm used to generate a spiketrain from a time varying analog signal is now presented. It is called the "Hough Spiker Algorithm" (HSA) and can be viewed as the inverse of the convolution algorithm described above in Section 3.

To give an intuitive feel for this deconvolution algorithm, consider a spiketrain consisting of a single pulse (all 0 s with one 1). When this pulse passes through the convolution function window, it adds each value of the convolution function to the output in turn.

A single pulse: (100000 . . . $\rightarrow t = +\infty$) will be convolved with the convolution function expressed as a function of time. At $t = 0$ its value will be the first value of the convolution filter, at $t = 1$ its value will be the second value of the convolution filter, etc. Just as a particular spiketrain is a series of spikes with time delays between them, so too the convolved spiketrain will be the sum of the convolution filters, with (possibly) time delays between them. At each clock tick when there is a spike, add the convolution filter to the output. If there is no spike, just shift the time offset and repeat.

The same example.

```
spike train        1 1 0 1 0 0 1
convolution filter 1 4 9 5 -2

     t -> 0 1  2  3   4  5 6   7 8 9 10
 out:
  1       1 4  9  5  -2
  1         1  4  9   5 -2
  0            0  0   0  0 0
  1               1   4  9 5 -2
  0                   0  0 0   0 0
  0                      0 0   0 0 0
  1                        1   4 9 5 -2
          -----------------------------
          1 5 13 15   7  7 6   2 9 5 -2
```

In the HSA deconvolution algorithm, we take advantage of this summation, and in effect do the reverse, a kind of progressive subtraction of the convolution function. If at a given clock tick, the values of the

المنارة للاستشارات
www.manaraa.c

convolution function are less than the analog values at the corresponding positions, then subtract the convolution function values from the analog values. The justification for this is that for the analog values to be greater than the convolution values, implies that to generate the analog signal values at that clock tick, the CoDi module must have fired at that moment, and this firing contributed the set of convolution values to the analog output. Once one has determined that at that clock tick, there should be a spike, one subtracts the convolution function's values, so that a similar process can be undertaken at the next clock tick. For example, to deconvolve the convolved output (using the same value of the convolution function as in the simple example of the previous section.

The resulting analog output (the jerky curve) should be very close to the original solid line as Fig. 2 shows it to be. The HSA seems to work well when the values of the waveforms are large and do not take values close to zero, and do not change too quickly relative to the time width of the convolution filter window. It may be possible to simply add a constant value to incoming analog signals before spiking them and to ensure that the analog signal does not change too rapidly.

Note however, that the HSA deconvolution algorithm was only discovered fairly recently, so the neural net module evolution that is discussed in Section 7 below, does not use it. The I/Os to these modules as specified by the evolutionary engineer were in binary, not analog.

```
          1  5 13 15  7  7  6  2  9  5 -2
compare:  1  4  9  5 -2   conv.vals<analog sig vals, so spike:1
          0  1  4 10  9  7  6  2  9  5 -2 subtract (time++)
compare:     1  4  9  5 -2                less, so spike:11
          0  0  0  1  4  9  6  2  9  5 -2 subtract (time++)
compare:        1  4  9  5 -2             not less, so no spike:110
          0  0  0  1  4  9  6  2  9  5 -2 (time++)
compare:           1  4  9  5 -2          less, so spike:1101
          0  0  0  0  0  0  1  4  9  5 -2 subtract (time++)
compare:              1  4  9  5 -2       not less:11010
          0  0  0  0  0  0  1  4  9  5 -2 (time++)
compare:                 1  4  9  5 -2    not less:110100
          0  0  0  0  0  0  1  4  9  5 -2 (time++)
compare:                    1  4  9  5 -2 less, so spike:1101001
          0  0  0  0  0  0  0  0  0  0  0 subtract (time++)
```

It is assumed that spiking will irreversibly raise the value of the convolved output. If the convolution filter value at a given clock tick is less than that of the target waveform, spiking will bring the two values closer together. If the waveform value is still too low after a spike has occurred, a near future spike will bring the two closer together.

Figure 5 shows an example of an HSA spiketrain output. It is the spike train corresponding to Fig. 2 in fact. The original input analog signal is the solid line in Fig. 2. The spiketrain resulting from each analog input is sent into the SIIC convolver (shown in Fig. 1).

5. The CAM-Brain Machine (CBM)

5.1. CBM Overview

The CAM-Brain Machine (CAM stands for Cellular Automata Machine) is a research tool for the creation of artificial brains. An original set of ideas for the CAM-Brain project was developed by Dr. Hugo de Garis at the Evolutionary Systems Department of ATR HIP (Kyoto, Japan), and is currently being implemented as

```
( time ---> )
111100010001101111110100010111110110100010101110100100010011010100
100010101010100101001010110001101010011001101011010101011101110101101
```

Figure 5. The spiketrain output of Fig. 2, as generated by the Hough Spiker Algorithm (HSA).

للاستشارات
www.manaraa.com

a dedicated research tool by Genobyte, Inc. (Boulder, Colorado). Genobyte is licensed by ATR International and Japan's Key Technologies Center to manufacture and sell CBMs to third parties.

An artificial brain, supported by the CBM, consists of up to 64,640 neural modules, each module populated with up to 1,152 neurons, a total of 74.5 million neurons. Within each neural module, neurons are densely interconnected with branching dendritic and axonic trees in a three-dimensional space, forming an arbitrarily complex interconnection topology. A neural module can receive afferent axons from up to 188 other modules of the brain, with each axon being capable of multiple branching in three dimensions, forming hundreds of connections with dendritic branches inside the module. Each module sends efferent axon branches to up to 64,640 other modules.

A critical part of the CBM approach is that the detailed dendritic/axonal tree structure of the neural modules is not "manually designed" or "engineered" to perform a specific brain function, but rather evolved directly in hardware, using genetic algorithms, in the spirit of the growing research field of evolvable hardware [9–12].

Genetic algorithms operate on a population of chromosomes, which represent neural networks of different topologies and functionalities. Better performers for a particular function are selected and further reproduced using chromosome recombination and mutation. After hundreds of generations, this approach produces very complex neural networks with a desired functionality. The evolutionary approach can create a complex functionality without any a priori knowledge about how to achieve it, as long as the desired input/output function is known.

5.2. CBM Architecture

We begin the description of the CBM with a brief overview, followed by several paragraphs giving a somewhat greater level of detail. These paragraphs also attempt to justify to some extent the architectural decisions we made. Note that we have compromised here between a need for corporate secrecy (Genobyte, Michael Korkin's company [13], has a licensing agreement with ATR to build and sell CBMs, hopefully free from imitators for several years) and academic openness, so the description below is somewhat lacking in critical details.

In the CBM we have implemented what is called "function-level" evolvable hardware, as opposed to "gate-level" evolvable hardware, which directly operates on a sea of Boolean gates. Our functions take the form of cellular automata cells, which are manually designed and configured in Xilinx XC6264 FPGA chips. (Note that Xilinx removed the XC6200 family of chips from the market. We managed to salvage the few remaining XC6264 chips from Xilinx, enough to build approximately 8 CAM-Brain Machines (CBM) in the next few years.) Each of these cellular automata cells contains a 6-bit register and some additional logic, which allow it to exchange signals with its neighboring cells. The contents of the register is the subject of evolution. So, instead of using FPGA configuration memory space to instantiate different circuits, our design utilizes our own "configuration" space made up of multiple 6-bit registers in CA cells, which are preloaded into the FPGAs. In fact, the CBM design uses three different cell functions for three different phases of operation (i.e. growth, signaling, and genetic), so we reconfigure the entire FPGA chips multiple times in the process of cycling through the CBM phases. A high reconfiguration speed and direct access to the user-level registers in the XC6264 chips allow us to achieve high overall throughput.

The following provides further details of our CBM implementation.

The CBM architecture is designed around the architectural features of Xilinx's XC6264 FPGA chips. These SRAM-based FPGAs allow rapid reconfiguration logic at the rate of 60 Mbytes/s. A full CBM array of 72 FPGAs forms a cellular automata cubic space of $24 \times 24 \times 24$ cells. Each FPGA holds a subspace of $8 \times 6 \times 4$ CA cells, a total of 192. These FPGAs are further interconnected to provide a continuous, uninterrupted space. Each FPGA has 208 bidirectional connections with its neighboring FPGAs in a three-dimensional logical space. Each FPGA is located on a separate PCB, which also carries a tightly coupled 16 Mbyte DRAM SIMM and control logic CPLD. Interconnections are made via a large backplane panel carrying all 72 FPGA module PCBs. The cellular space is wrapped around all three axes of the CA cube, forming a toroidal cube. All 72 FPGA functions are accomplished in parallel for the complete array under central control, while each FPGA has its own data to work with in its own 16 Mbytes memory space. Thus, the CBM architecture is of the SIMD (single instruction multiple data) type.

المنارة للاستشارات
www.manaraa.com

The FPGA array is time shared between multiple neural modules during an evolution run, or during brain run mode, by rapid instantiation of each module for a period of 12 microseconds, during which time the CA space is clocked 96 times at 9.47 MHz. At the end of this period, the status of the cells is saved in the 16 Mbytes of DRAM, while the next module configuration is uploaded into the CA space from the DRAM. The resultant cellular update rate in the CBM's array of 72 FPGAs is on the order of 114 billion cells/second.

Each CA cell contains function logic and control registers which determine its operation. A cell typically occupies a rectangular FPGA subspace of 64 fine-grain function units, and a control register typically contains 7 to 35 bits. Cell registers can be written or read through a 32-bit FPGA data interface in the same manner as the FPGA configuration space is accessed, which is a distinctive feature of the XC6264. Cells are interconnected inside the FPGA with their neighboring cells using internal routing resources. Those cells which form the external surface of the CA subspace connect to cells inside the neighboring FPGAs in the array, a total of 208 connections. All inter-chip connections in the CBM have an open-drain configuration with external pull-ups to protect them from potential damage resulting from certain configuration patterns in the connected CA cells belonging to different FPGAs.

Each CA cell's internal control registers are implemented as dual pipeline registers. The first stage is used to upload new bitstrings into all 192 cells in an FPGA through the 32-bit data interface, while the second stage holds the current cell configuration of the functioning cellular automata space. The first stage register's contents can be loaded into the second stage register for all cells in parallel using a global signal. This accomplishes complete CA space reconfiguration in a matter of nanoseconds as well as simultaneous execution of the CA states with a background reconfiguration for the next neural module instantiation. Thus, the hardware core of the CBM is continuously utilized without any considerable idle time.

For each of the three operational phases of the CBM, during every generation of a genetic algorithm (growth phase, signal phase, genetic phase), the full array of the 72 FPGAs is rapidly reconfigured with a completely different set of CA cell functions. In the growth phase, the CA cells perform a network growth algorithm, while their control registers are uploaded with the neural module's chromosomes. The result of the growth phase is the neural module phenotype to be saved at the end of the growth phase. The phenotype is further used to configure the signal phase cells during the signal phase. In the genetic phase, the function of the cells is to create an offspring chromosome from two parent chromosomes using crossover and mutation masks.

Reconfiguration is accomplished by loading the configuration data from the DRAM SIMM via the 32-bit FPGA data interface. Complete FPGA reconfiguration takes less than one millisecond. All 72 FPGAs are reconfigured in parallel. An alternative to reconfiguring an FPGA for each operational phase would have been implementing more complex CA cells capable of functioning in all phases. This would have resulted in a significantly smaller cellular space fittable into the FPGA. The rapid reconfiguration capability of the XC6264 provided a solution which allows a large number of cells with a high functional diversity, in exchange for a small additional operation time. This additional time is less than 3 seconds per 1000 generations of evolution.

In addition to the main FPGA array, the CBM utilizes four XC6264 FPGAs for spiketrain buffer logic and for a fitness evaluation unit. The fitness evaluation unit holds eight separate 24-tap convolution filters for output/target spiketrain deviation computation during the evolution runs.

The CBM consists of the following six major blocks:

1. Cellular Automata Module
2. Genotype/Phenotype Memory
3. Fitness Evaluation Unit
4. Genetic Algorithm Unit
5. Module Interconnection Memory
6. External Interface

Each of these blocks is discussed in detail below, followed by some further architectural points in Section 5.3. A summary of CBM capacities can be found in Table 1.

Cellular Automata Module. The cellular automata module is the hardware core of the CBM. It is intended to accelerate the speed of brain evolution through a highly parallel execution of cellular state updates. The CA module consists of an array of identical hardware logic circuits or cells arranged as a 3D structure of $24 \times 24 \times 24$ cells (a total of 13,824 cells). Cells forming the top layer of the module are recurrently connected with the cells in the bottom layer. A similar recurrent connection is made between the cells on the

المنارة للاستشارات

www.manaraa.com

Table 1. Summary of CBM technical specifications.

Cellular automata update rate (max.)	130 billion cells/s
Cellular automata update rate (min.)	114 billion cells/s
Number of supported cellular automata cells (max.)	893 million
Number of supported neurons (max., per module)	1,152
Number of supported neurons (max., per brain)	74,465,244
Number of supported neural modules	64,640
Data flow rate, neuronal level (max.)	13.5 Gbytes/s
Data flow rate, dendrite level (estimated average)	40.8 Gbytes/s
Data flow rate, intermodular level (max.)	74 Mbytes/s
Number of FPGAs	72
Number of FPGA reconfigurable function units	1,179,648
Phenotype/genotype memory	1.18 Gbytes
Chromosome length	91,008 bits
Power consumption	1.5 KWatt (5 V, 300 A)

north and south, east and west vertical surfaces. Thus a fully recurrent toroidal cube is formed. This feature allows a higher axonic and dendritic growth capacity by effectively doubling each of the three dimensions of the cellular space.

The CBM hardware core is time-shared between multiple modules forming a brain during brain simulation. Only one module is instantiated at a time. The FPGA firmware design is a dual-buffered structure, which allows simultaneous configuration of the next module while the current module is being run (i.e. signals are propagated through the dendrites and axons between neurons). Thus, the FPGA core is run continuously without any idle time between modules for reconfiguration.

The surfaces of the cube have external connections to provide signal input from other modules. Each surface has a matrix of 64 signals, which is repeated on the opposite surface due to wrap around connections. Thus, a total of 192 different connections is available. Four connections, i.e. one on each of the surfaces, and one at one of the 8 corner cells of the cube, are used as output points. Due to wrap around, any corner cell has 3 wrap-around faces, so it is within two cells maximum of any other corner cell, including the opposite corner, and at the same time equidistant from the three other outputs. The fourth output is equivalent to the center of the cube, so the set of all 4 outputs looks nice and symmetric.

The CA module is implemented with Xilinx FPGA devices XC6264. These devices are fully and partially reconfigurable, feature a new co-processor architecture with data and address bus access in addition to user inputs and outputs, and allow the reading and writing of any of the internal flip-flops through the data bus. An XC6264 FPGA contains 16,384 logic function cells [2], each cell featuring a flip-flop and Boolean logic capacity, capable of toggling at a 220 MHz rate. Logic cells are interconnected with neighbors at several hierarchical levels, providing identical propagation delay for any length of connection. This feature is very well suited for a 3D CA space configuration. Additionally, clock routing is optimized for equal propagation time, and power distribution is implemented in a redundant manner.

To implement the CA module, a 3D block of identical logic cells is configured inside each XC6264 device, with CoDi specified 1-bit signal buses interconnecting the cells. Given the FPGA internal routing capabilities and the logic capacity needed to implement each cell, the optimal arrangement for a XC6264 is $4 \times 6 \times 8$ (192 cells). This elementary block of cells requires 208 external connections to form a larger 3D block by interconnecting with six neighbor FPGAs on the south, north, east, west, top, and bottom sides in a virtual 3D space. A total of 72 FPGAs, arranged as a $6 \times 4 \times 3$ array are used to implement a $24 \times 24 \times 24$ cellular cube.

The CBM implements interconnections between 72 FPGAs, each placed on a small individual printed circuit board, in the form of one large backplane board, carrying all 72 FPGA daughter boards.

The CBM clock rate for cellular update is selected between 8.25 MHz, 9.42 MHz, and 11 MHz. At this

المنارة للاستشارات
www.manaraa.com

rate all 13,824 cells are updated simultaneously, which results in the update rate of 114 to 130 billion cells/s. This rate exceeds the CAM-8 update rate by a factor of 570 to 650 times.

Genotype and Phenotype Memory. Each of the 72 FPGA daughter boards includes 16 Mbytes of EDO DRAM to be used for storing the genotypes and phenotypes of the neural modules, a total of 1,180 Mbytes. The genotype is the set of genes in a cell and the phenotype is the final product of the genotype, the body and behavior that the genotype builds/generates. There are two modes of CBM operation, namely evolution mode and run mode. The evolution mode involves the growth phase and signaling phase. During the growth phase, memory is used to store the chromosome bitstrings of the evolving population of modules (module genotypes). For a module of 13,824 cells there are over 91 Kbits of genotype memory needed. For each module the genotype memory also stores information concerning the locations and orientations of the neurons inside the module, and their synaptic masks.

During the run mode, memory is used as a phenotype memory for the evolved modules. The phenotype data describes the grown axonic and dendritic trees and their respective neurons for each module. The phenotype data is loaded into the CA module to configure it according to the evolved function. The genotype/phenotype memory is used to store and rapidly reconfigure (reload) the FPGA hardware CA module. Reconfiguration can be performed in parallel with running the module, due to a dual pipelined phenotype/genotype register provided in each cell. This guarantees the continuous running of the FPGA array at full speed with no interruptions for reloading in either evolution or run modes. The phenotype/genotype memory can support up to 64,640 interconnected neural modules at a time. An additional memory will be based in the main memory of the host computer (Pentium 500 MHz) connected to the CBM through a PCI bus, capable of transferring data at 132 Mbytes/s.

Fitness Evaluation Unit. Signaling in the CBM is accomplished with 1-bit spiketrains, a sequence of ones separated by intervals of zeros, similar to those of biological neural networks. Information, representing external stimuli, as well as internal waveforms, is encoded in spiketrains using a so-called "Spike Interval Information Coding (SIIC)". This method of coding is implemented by nature in animal neural networks, and is very efficient in terms of information capacity per spike. Conversion from spiketrains into "analog" waveforms representing external stimuli, or internal signaling, is accomplished by convolving the spiketrain with a special multi-tap linear filter.

When a module is being evolved, it must be evaluated in terms of it's fitness for a targeted task. During the signaling phase, each module receives up to 188 different spiketrains, and produces up to four different output spiketrains, which are compared with a target array of spiketrains in order to guide the evolutionary process. This comparison gives a measure of performance, or fitness, of the module.

Fitness evaluation is supported by a hardware unit which consists of an input spiketrain buffer, a target spiketrain buffer, and a fitness evaluator. During each clock cycle an input vector is read from its stack and fed into the module's inputs. At the same time, a target vector is read from its buffer to be compared with the current module outputs by the evaluator. The fitness evaluator performs a convolution of the spiketrains with the convolution filter, and computes the sum of the waveform's absolute deviations for the duration of the signaling phase. At the end of the signaling phase, a final measure of the module's fitness is instantly available.

Genetic Algorithm Unit. To evolve a module, a population of modules is evaluated by computing every module's fitness measure, as described above. A subset of the best modules are then selected for further reproduction. In each generation of modules, the best are mated and mutated to produce a set of offspring modules to become the next generation. Mating and mutation is performed by the CBM hardware core at high speed, configured for the genetic phase. During this phase, each cell's firmware implements crossover and mutation masks, two parent registers and an offspring register. Thus, each offspring chromosome is generated in nanoseconds, directly in hardware. Crossover is performed in parallel in hardware by all of a module's 14K CA cells. One crossover act takes about 100 ns for two parent chromosomes, each of which is 91 Kbit long, using a 91 Kbit crossover mask and a 91 Kbit mutation mask. The selection algorithm is performed by the host computer in software, using access to the CBM via a PCI interface.

Module Interconnection Memory. In order to support the run mode of operation, which requires a large

www.manaraa.com

number of evolved modules to function as one artificial brain, a module interconnection memory is provided. Each module can receive inputs from up to 188 other modules. A list of these source modules referenced to each module is stored in a CBM cross-reference memory (3 Mbytes) by the host computer. This list is compiled by CBM software using a module interconnection netlist in EDIF format. This netlist reflects the module interconnections as designed by the user, using off-the-shelf schematic capture tools.

The length of module interconnections is 96 cells (clock cycles). For each of the 64,640 modules, a Signal Memory stores up to three 96-bit long output spiketrains.

During the run mode, at the time each module of a brain is configured in the CA hardware core (by loading its phenotype), a signal input buffer is also loaded with up to 188 spiketrains according to the netlist in the module interconnection memory. The spiketrains are the signals saved from the previous instantiation and signaling of the 188 sourcing modules. At the same time, the four output spiketrains of the currently instantiated module are saved back to the Signal Memory. This repetitive cycling through all the modules which form the brain, results in a repetitive saving and retrieving of the spiketrains to/from the Signal Memory. It provides the signaling between modules according to the brain interconnection structure reflected in the schematics, designed by the user.

In a maximum brain with 64,640 modules, the CBM update rate is such that each cell propagates approximately 288 bit-long spiketrains per second. A 288 bit-long spiketrain can carry on the order of 72 bytes of signal information, using the SIIC coding method. Each neuron receives up to 5 spiketrains, so there are up to 188 million spiketrains being processed by neurons in the brain. Thus the maximum information processing rate by all neurons in the brain is of the order of 13.5 Gbytes/s.

Additional spiketrain processing in multiple dendritic branches can be estimated by assuming 50% of the total cellular space to be occupied by dendrite cells, each cell on average having 2.5 branches out of 5 possible. Informational throughput of dendrite cells is then of the order of 40.8 Gbyte/s.

External Interface. The CBM architecture can receive and send spiketrains not only from/to the Signal Memory, but also from/to the external CBM interface. Any module can receive up to 188 incoming spiketrains and send up to 4 spiketrains to an external device, such as a robot, a speech processing system, etc. In a brain with 16,384 modules, the information rate, as measured at the external interface is up to 4.5 Kbytes/s per each module, or up to 74 Mbyte/s overall. In a smaller brain with less number of modules, the external information rate is higher, for example, a brain with 4,000 modules provides quadruple the external information rate for each module (18 Kbyte/s).

5.3. Further CBM Architectural Points

The CBM core is implemented as a large 12-layer backplane with 72 FPGA module boards plugged in. Each FPGA module board contains one Xilinx XC6264 BG560 FPGA, one Xilinx XC95216 BG352 CPLD, and a 16 Mbyte EDO DRAM module. (Each of the 72 FPGAs has a tightly coupled unshared 16 Mbyte EDO DRAM that it is connected via the FastMap interface to the FPGA to provide the fastest possible speed for FPGA reconfiguration, as well as loading and saving neural module configurations in signal and growth phase.) Each FPGA contains 16K reconfigurable function units. Memory is used under CPLD control to load and save FPGA configurations to accomplish time sharing of the fast FPGA hardware. The datapath between memory and an FPGA is 32-bits wide and provides a data transfer rate of 66 Mbyte/s. An FPGA is thermally coupled with a temperature sensor circuit which is pre-programmed to shut-off the main clock when a temperature limit is exceeded.

The backplane serves primarily as a means to interconnect all 72 FPGAs. Each FPGA has 208 bidirectional connections to six other FPGAs arranged as a three-dimensional array of 6 by 3 by 4 FPGAs. In addition, the backplane's opposite side hosts several other boards used for overall sequencing and control of the system, implementing an SIMD (Single Instruction Multiple Data) architecture. Overall, there are 7.2 million reconfigurable gates in the CBM. To accomplish this connectivity, a High Density Metric connector system is used with press-fit contacts, providing over 30,000 connections.

The CBM is connected as a PCI target to a Pentium III computer which initializes the system and performs some background auxiliary control.

Although the CBM has been developed primarily to implement a specific neural network model based on cellular automata, its architecture is quite universal

المنارة للاستشارات

www.manaraa.com

and very flexible. In fact, the CBM can be used for a large variety of applications which benefit from a high speed and fast reconfigurability of its hardware. Hardware-based implementations of a variety of algorithms have been shown to exceed the computational speed of high-cost super computers, as is the case with the CAM-Brain algorithm. The maximum computational power of the CBM is estimated to be equivalent to ten thousand Pentium III 500 MHz computers in the CAM-Brain algorithm implementation. Since the figure of 10,000 may be surprising to some readers, a quick justification is given. CBM updates 13,824 cells every 106 nanoseconds, or 7.7 picoseconds per cell update. An equivalent software algorithm requires 13 instructions per cell update, each instruction 3 clocks on average, with pipelining, a total of 78 nanoseconds (for Pentium III 500 MHz). Hence, the ratio is roughly 10,000.

In particular, one application supported by the CBM architecture is gate-level and function-level evolvable hardware, which is based on applying a genetic algorithm to evolve complex digital circuits for a specific task. With 7.2 million gates, the resulting circuit complexity is likely to exceed human ability to design, debug, or even understand the dynamics of such a circuit. The CAM-Brain algorithm itself is an example of function-level evolvable hardware, where a basic unit of evolution is a function of a cellular automata cell, implemented as a specific (non-evolvable) logic circuit. This circuit can implement a number of different functions selectable by loading a chromosome bit string into the cell's genotype register which switches the cell to perform a specific function.

A summary of the CBM technical specifications can be found in Table 1.

6. "Robokoneko", the Kitten Robot

An artificial brain with nothing to control is rather useless, so we chose a controllable object that we thought would attract a lot of media attention, i.e. a cute life-size robot kitten that we call "Robokoneko". We did this partly for political and strategic reasons. Brain building is still very much in the "proof of concept" phase, so we want to show the world something that is controlled by an artificial brain, that would not require a PhD to understand what it is doing. If the kitten robot can perform lots of interesting behaviors, this will be obvious to anyone simply by observation. The more media attention the kitten robot gets, the more likely

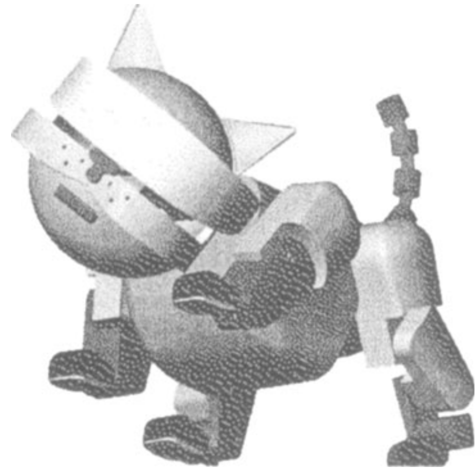


Figure 6. "Robokoneko", the life-sized kitten robot to be controlled by our artificial brain.

our brain building work will be funded beyond 2001 (the end of our current research project).

Figure 6 shows the mechanical design our team has chosen for the kitten robot. Its total length is about 25 cms, hence roughly life size. Its torso has two components, joined with 2 degrees of freedom (DoF) articulation. The back legs have 1 DoF at the ankle and the knee, and 2 DoF at the hip. All 4 feet are spring loaded between the heel and toe pad. The front legs have 1 DoF at the knee, and 2 DoF at the hip. With one mechanical motor per DoF, that makes 14 motors for the legs. 2 motors are required for the connection between the back and front torso, 3 for the neck, 1 to open and close the mouth, 2 for the tail, 1 for camera zooming, giving a total of 23 motors.

In order to evolve modules which can control the motions of the robot kitten, we thought it would be a good idea to feed back the state of each motor (i.e. a spiketrain generated from the pulse width modulation PWM output value of the motor) into the controlling module. Since each module can have up to 188 inputs, feeding in these 23 motor state values will be no problem. We may install acceleromotors and/or gyroscopes which may add another 6 or more inputs to each motion control module. It can thus be seen that the mechanical design of the kitten robot has implications on the design of the CBM modules. There need to be sufficient numbers of inputs for example.

The motion control modules will not be evolved directly using the mechanical robot kitten. This would be hopelessly slow. Mechanical fitness measurement is impractical for our purposes. Instead we will soon be

المنارة للاستشارات

www.manaraa.com

simulating the kitten's motions using an elaborate commercial simulation software package called "Working Model-3D". This software will allow output from an evolving module to control the simulated motors of the simulated kitten. This software simulation approach negates to some extent the philosophy of the CAM-Brain Machine and the CAM-Brain Project, i.e. the need for hardware evolution speeds. This compromise was felt to be a necessary evil. In practice, the proportion of modules concerned with motion control will be very small compared to the total. Potentially, we have 64K modules to play with. Probably most of them will be concerned with pattern recognition, vision, audition, etc. and decision making. Designing the kitten robot artificial brain remains the greatest research challenge of the CAM-Brain Project and will occupy us through 1999, and beyond.

Work on the evolution of the motions of the robot kitten has already begun and a journal article on this work has been submitted for publication [14]. The strategy employed was as follows. The evolution of the kitten's behaviors will not occur at electronic speeds. The kitten's motions were simulated with Working Model 3D (WM3D) software, which is very elaborate, incorporating gravity, moments of inertia, frictions, etc. This software is used by major companies to simulate their new designs before fabrication. Our team wrote a software interface to WM3D which allows a genetic algorithm (GA) to be performed on the simulated motions. The user can specify a "fitness" (performance criterion) definition which is then used to evolve the desired motion. In practice, this evolution was very slow, taking several days per motion. We were thus motivated to find ways to accelerate this process. We took several options. One was to hand code "ball park" motion vectors (of the form of an angular acceleration per motor per clock tick, for all motors, for a given interval of clockticks) for a given desired motion. This hand coded ballpark solution served as an initial population in the genetic algorithm, and the GA was then used to "fine tune" the motion which was often jerky at first. Another option is to use a cluster of work stations with one GA chromosome (motion vector) per machine, resulting in an Nfold speedup with *N* machines. Once a motion vector is evolved under simulation it becomes the target vector that the CBM uses to evolve a corresponding module giving the same time dependent output. This evolved motion control module (actually one module per motor for all motors, in a set we call a "cluster") is then downloaded into the RAM of the artificial brain. (At least that is the plan. We have not tested this yet.) Evolving motions may be slow, but it is a "once off" affair, and although a major compromise to the "evolution of neural net modules at electronic speeds" philosophy, it is seen as being unavoidable, but not a crippling handicap.

Perhaps some clarification may be useful at this point. One may wonder that once the GA-using-simulation has evolved the needed motion vector, what is gained by then evolving a neural net that produces this same motion vector, which is, in effect, merely using the CBM to convert from the motion vector to the neural net. This is done for several reasons. One is to take advantage of the generalization properties of neural nets. Another is to have modules that can then be run efficiently on the CBM. Thus it may appear that the CBM is being used only as a "motion vector to neural net converter", and as a runtime environment for the set of 10,000 s of modules. It appears as though the FPGA is not being used for the real learning. This is true for the motion control modules, but is not true for the vast majority of modules, such as the aural and visual pattern detector modules, the logic control modules, etc. We estimate that the motion control modules will constitute less than a few percent of the total. Since the non motion control modules will be evolved directly in the CBM, the claim that the (evolutionary) learning does not occur in the CBM is not true in most cases.

We do not know yet how well we can get modules to interact together. This question remains unexplored. Without a CBM's speed, the evolution and updating of many interconnected modules remains impractical. No one has tried to build an artificial brain before on the scale attempted in this project. Of course, we can define an "artificial brain" to be whatever we like (in the case of the CAM-Brain project, the definition is "an assemblage of evolved CA based neural net modules"). With the CBM evolving a module in about 1 second, and with hundreds of evolutionary engineers helping out, maybe it is realistic to build an artificial brain with 10,000 s of modules in a few years. We don't know. That is the research challenge.

7. A Sampler of CoDi-1Bit Evolved Neural Net Modules

Since the whole point of using the CBM is to attain a high evolution speed, it is useful if the representation chosen to interpret the 1 bit signals which enter and

المنارة للاستشارات www.manaraa.com

leave the CoDi modules can be unique, otherwise several representations would need to be implemented in the electronics. (For the CBM to be efficient, i.e. to evolve CoDi modules in about 1 second, fitness measurements need to be performed at electronic speeds, which implies that the representation chosen for the signals be implemented directly in the hardware.) We chose the SIIC to be our unique representation. However, as mentioned at the bottom of Section 5, most of the evolutionary experiments presented here were already undertaken before the SIIC representation was chosen. Since the results of these earlier experiments are interesting in their own right, we report on them here. They show to what extent that CoDi modules are evolvable and the power of their functionality. The evolution of SIIC-representation-based and HSA-based modules will be the subject of work in the very near future, given that both algorithms are now ready. So is the CBM multi-module simulation code, so progress should be rather rapid in the coming months prior to the delivery of the CBM itself. Once the CBM is delivered, multi-module systems should be built as fast as we can dream them up. The bottleneck in building large scale multi-module systems will become human creativity lag, not module evolution lag (as was the case with software evolution speeds in the "pre-CBM era".) We now provide a sample of evolved CoDi neural net modules, their specified functionalities, and their actual performances, to give a feel for what they can do.

7.1. *Multiple Timer Module*

Since a 100% fitness score does not test the limits of evolvability of a module, a more demanding output function was tried. The target output (similar to the above pattern) and the actual evolved output (placed immediately under the target pattern for comparison) were as follows:

Target
000000000000000000000000000000
11111111111111111111
Evolved
000000000000000000000000000000
00011111111111111111
Target ctd.
000000000000000000000000 1111111111111111
00000000000000000000
Evolved ctd.
100000000000000000000000 0111111111111111
10000000000000000000

The fitness definition was as follows. If a 0 appeared in the first (0) block, score 12 points. If a 1 appeared in the second (1) block, score 7 points. If a 0 appeared in the third block (0), score 3 points. If a 1 appeared in the fourth block (1), score 2 points. If a 0 appeared in the fifth block (0), score 1 point. Hence a perfect score would be $30 \times 12 + 20 \times 7 + 24 \times 3 + 16 \times 2 + 20 \times 1 = 624$. These weightings were chosen so as to encourage the earlier outputs to be correct before the later outputs. Population size was 30. No crossover. This result converged after 100 generations with a fitness value of 0.957.

It is interesting to note that these good results were evolving in 100 generations, and yet the chromosome length is very large. The standard CBM chromosome length is of the order of 90K bits. One might think that such a long chromosome would be very slow in evolving, but this was not the case. One possible explanation for this is that there may be so many possible solutions, that (any reasonable) one is quickly found.

7.2. *Pattern Detector Module*

With a slight modification of the code used to evolve the above module, a pattern detector module was evolved which was capable of distinguishing between two square wave inputs, of 111000111000... and 11111000001111100000... In this case, no switch input was used. Two experiments were run. In the first, the input was the 6 clocktick cycle square wave input, applied at the fixed input point $(8, 8, 0)$. In the second experiment, the circuit was regrown with the same chromosome and the 10 clocktick cycle square wave input was applied to the same fixed input point. The fitness definition was the same as above. Results are shown below. Over 90 clockticks, the first output had 48 more 1 s than the second output.

Square wave input 111000111000...
Output
000000000000000000000000100110111011111111
111111111111111111111111111111111111111111
111111
Square wave input 11111000001111100000...
Output
000000000000000000000000000010001000100010
001000100010001000100010000000000100010001
00010

Since the CoDi modules seem capable of evolving such detectors, it may be possible to evolve modules

www.manaraa.com

which are capable of detecting a specific phoneme analog input (e.g. the spike train (bit string) which when convolved with a particular convolution function gives the time dependent analog signal). In a manner similar to the above, one could input the signal in the first experiment, and a random signal in the second, in a multi-test experiment, and evolve the phoneme detector. Maybe one could evolve a set of detectors, one for each phoneme. By using the SIIC and HSA digital/analog conversions, this kind of thing may become quite practical.

For the evolution of the pattern detector module and the module in the next section, the population size was 15. Fitness values reached saturation after about 200 generations or less. After the evolution, observation of the occupancy of the CA cells showed that most of them were filled (more than 95%), with roughly equal numbers of axons as dendrites, which is to be expected given that the initial growth instruction at a neuron's face could be an axon or dendrite with a 50/50 chance depending on whether a bit was set to 1 or 0 in the module's chromosome. Since it took an overnight run to evolve a single module, usually only one run was undertaken. Once the CBM is functional, more robust statistics can be undertaken by averaging over many runs.

7.3. Hubel-Wiesel Line Motion Detector Module

The results of the following experiment were significant for the CAM-Brain Project as a whole, we felt. It involved the evolution of a Hubel-Wiesel type line motion detector. Hubel and Wiesel won a Nobel prize for discovering that certain neural cells in the visual region of the cat's brain fired strongly when lines of light at particular orientations and speeds were shone onto a screen that the cats were observing. These cells (neurons) were detecting the motion of lines at a particular orientation. The evolution of this "Hubel-Wiesel" module used the same fitness definition and a similar methodology as in the above case. In the first experiment, a square 12×12 neuron input grid was used. At the first clock tick, the top horizontal 12 neurons were made to fire—at the second clock tick, the second horizontal row of 12 neurons was made to fire, etc., for 12 clock ticks, then the cycle was repeated. This input firing pattern simulated the motion of a line of light moving horizontally down the visual field on the retina of a cat. In the second experiment, 12 randomly positioned input neurons were fired at each clock tick. These 12 positions were randomly generated for each clock tick. This second input firing pattern simulated input noise, to be contrasted with the line motion input. Output results are shown below.

Line Motion Input Case
Output
000000000000000000000000001001101001101111
11111111111111111111111111111111111111111
1111111
Random Input Case
Output
0000000000000000000000000000001000000000000
000100010101010101010101010101010101010101
010101

There were 35 more 1 bit outputs in the first case than the second. Since the inputs to the second case are positioned randomly, the same neural net module will generate a different fitness value depending on the input. Nevertheless the evolution still improved over time, developing a fairly robust net giving fitness values corresponding to over 30 1-bit differences (between the two experiments) in most cases (e.g. the top 5 fitness chromosomes were saved for each generation and not crossed over or mutated. The fitness values (1-bit difference count) of these top 5 were 31, 34, 35, 30, 29 after several hundred generations). Thus it was possible to evolve a Hubel-Wiesel Line Motion Detector. Actually, the statement that it was possible to evolve a Hubel-Wiesel line motion detector may be disputed here. For example, the target function could be stated as: "Output 1 if and only if at least one of the horizontal rows of inputs contains all 1's." This can be done with a multi-input AND gate (one neuron perhaps) for each row, followed by an OR (one dendrite). A more interesting conclusion might be that Hubel-Wiesel type systems can be more naturally achieved because of the crucial element of spatial layout in the CAM-Brain, as distinct from purely topological networks. Of course, we have no idea how the circuit does what it does. This is the great strength of "evolutionary engineering". Evolved circuits can at times achieve performance levels beyond what human engineers can achieve with traditional top-down design techniques, i.e. attain superior engineering performance levels, but the price is that one loses scientific understanding, due to the overwhelming structural and dynamical complexity of these CoDi circuits. The reason why many evolutionary engineers feel that evolutionary engineering can

المنارة للاستشارات

www.manaraa.com

potentially evolve a level of functionality superior to what human beings can design, is due to the so-called "complexity independence" of the genetic algorithm. This notion of complexity independence is informal. It means that the GA does not care about the inherent structural and dynamic complexity of the system it evolves. All that matters to a GA is that its (scalar) fitness values keep increasing. Hence a GA can evolve an extremely complex system (in structure and/or dynamics) which may surpass the complexity level limit that a human brain can comprehend. That extra complexity can provide an extra level of functionality. This feature is thought to be one of the great advantages of evolutionary engineering. Thus "evolutionary engineering" can sometimes provide a superior form of engineering. In practice, once evolutionary engineers can generate tens of thousands, even millions of modules, only a few die-hard analysts will want to know how an individual module functions. For the most part, no one will care how a particular module amongst millions actually does what it does.

8. Ideas for Interesting Future CoDi Modules to be Evolved

8.1. Multi-Test Modules

The CBM hardware automatically performs a fitness measurement on the assumption that the 1 Bit signals which leave the evolving module into the fitness measurement circuit are interpreted with the SIIC approach, i.e. the hardware actually implements the SIIC convolution algorithm. We have implemented the CBM having a single very general fitness measurement methodology, to simplify the electronics. Hence evolutionary engineers using the CBM will need to specify the functions of the modules they want to evolve using the SIIC methodology. However, there is a problem with this unified approach, namely how to give the same circuit several tests, i.e. several sets of different inputs in a single run. For example, imagine one aims to evolve a module which detects a time dependent input pattern P. One inputs the pattern P for T_p clocks. One wants the module to respond strongly when the pattern in detected, and weakly if any other pattern is presented. Hence the same circuit needs to be tested for several pattern inputs, i.e. P and others. The pattern P is called the positive case, while the others are called the negative cases. (It is also possible that there may be several positive cases (P_i).) One does not want a module which responds well to any pattern. It must discriminate.

How does one test all these cases (positive and negative) in a single run? By concatenating them, i.e. sandwiching them over time. For example, imagine there are 2 positive examples and 4 negative examples to be input to the same circuit. Hence there will be 6 time periods in which the patterns are presented sequentially at the input in one long run. Between each input signal presentation, the signal states in the circuit are cleared out, ready for the next signal input. This the CBM actually does. This resetting of the signal states is part of the CBM fitness measuring approach that we call "multitest" fitness measurement. The 6 input pattern periods can be represented as "P_1, P_2, N_1, N_2, N_3, N_4". The periods last P_i and N_i clock ticks each. So that the total number of clock ticks for the positive periods is more or less equal to the total of the negative periods, the durations of the P_i can be lengthened. This should increase the evolvability of the positive responses, otherwise the evolution may favor the negative cases too heavily. The target output patterns one wants for these 6 periods can be represented as "high, high, low, low, low, low".

Clearing the signal states between individual inputs in multi-test runs is needed because it is possible that self sustaining reverberating loops will be set up once an initial input is switched off. Such self sustaining loops may in fact be very useful, since they can be looked upon as a form of memory, and hence may be used to make CoDi modules capable of learning, i.e. adapting to their experience. The next subsection elaborates on this idea.

The CBM evaluates each partial fitness (one for each test in the multi-test case) and then sums the partial fitnesses to get the total fitness for the circuit (the module).

8.2. Modules Which Learn

Until recently, we have always thought that the CAM-Brain Project would produce neural circuits that would be INcapable of learning, i.e. they would not modify themselves based on their run time experience. The rationale was that it would be complicated enough dealing with tens of thousands of non learning modules all interacting with each other, let alone having tens of thousands of learnable modules. Also, we saw no way of having CoDi modules which could learn. Lately

المنارة للاستشارات

www.manaraa.com

however, we have begun to think that learnable CoDi modules might be evolvable. The essence of learning in a system is that some event in the past leaves some trace or memory in the system. In a CoDi module, that could take the form of reverberating internal signaling after an initiating input arrives. In some modules, once the input stops, the resulting 1 Bit signals could die away, i.e. be transient. Alternatively, the reverberating signals could persist and hence constitute a form of memory. Thus CoDi modules may be evolvable which generate reverberating signals.

8.3. From Multi Module Systems to Artificial Brains

Once our group and others have gained a lot of experience in evolving single modules, the next obvious step is to start to design multi-module systems, since the ultimate goal of the CAM-Brain Project is to put many many modules together (up to 64,460 of them in the current design of the CBM) to make artificial brains. Obviously, no CAM-Brain team will try to build a 64k module brain (with maximum 75 million artificial neurons) all at once. Instead, as a first step, small multi-module systems will be built, with tens of modules. Once experience is gained in how to do this successfully, larger systems will be undertaken, e.g. with 100 s of modules, then 1000 s, and later 10,000 s. Note that at the time of writing (Spring 1999) the authors make no pretense of knowing how to design a 64k module artificial brain. The whole point of the CAM-Brain project is to provide a tool which renders artificial brain building practical. Now that the tool exists, it is quite possible that the theory and the practice of brain building will advance rapidly. Just how to design a module artificial brain remains the major research challenge for the authors for the next few years.

Over time, artificial nervous systems should grow in complexity, until they can be called artificial brains. The robot kitten that our team is currently designing will be controlled by an artificial brain with up to 64k modules. Since this kitten robot contains a CCD TV camera, microphones for ears, touch sensors, 22 motors for the legs and body, etc, it should offer plenty of scope for brain building. This is a huge amount of work, which will need to be distributed over many CAM-Brain teams across the planet. With modern (almost cost free) internet telephone technology, coordinating such a large management effort is less expensive.

9. Related Work

This section deals with a sample of research work performed by others, which is related to the CAM-Brain Project, the CAM-Brain Machine (CBM), and the kitten robot Robokoneko that our artificial brain will control. Three aspects of our work have been chosen for comparison with comparable work by others, namely, the cellular automata machine (CAM) aspect, the runtime reconfigurable hardware aspect, and the pet robot aspect. In each of these three subsections, an initial brief summary of the related work is given, followed by a comparison with our work. The three related works we chose to discuss are:

(a) Margolus and Toffoli's CAM-8 Cellular Automata Machine
(b) Eldredge and Hutchings' Runtime Reconfigurable Neural Net Hardware
(c) Sony's Pet Dog Robot 'Aibo'.

9.1. Margolus and Toffoli's CAM-8 Cellular Automata Machine

The Information Mechanics Group at MIT has been concerned for the past decade (until they transfered recently to Boston University) with the physics of computation, including such topics as quantum computing, crystalline (3D) computing, and the hardware acceleration of cellular automata based modeling. Margolus and Toffoli have designed 8 versions of their Cellular Automata Machine (CAM) over the years [15]. Our group purchased their 8th version CAM-8 in 1994 and used it to obtain the graphics of our evolving neural circuits, some with 10 million artificial neurons. See de Garis's home page for these images. The title of our research project, "CAM-Brain Project" was based on the idea of putting an artificial brain inside a large cellular automata space inside a Cellular Automata Machine, hence CAM-Brain. The CAM-8 is essentially a dual RAM based lookup table hardware device. A 16 bit entry address for the LUT is obtained from the state of the central cell at a given position and its 4 neighbors. If the maximum number of states is 8, i.e. 3 bits, then the 5 states in the order (center, north, east, south, west) generate a $15 + 1$ bit string (with an extra zero). This address points to the next state of the center cell. With two such RAM memories, the RAM-1 at time T can be used to generate the states of the CA space at time

المنارة للاستشارات
www.manaraa.com

$T+1$, which are stored in RAM-2. At time $T+1$, the RAM-2 is used to generate the states of the CA space in RAM-1, overwriting the old contents. Thus the two RAMs ping-pong, updating each other. This is done at a rate of 200 million CA cells a second.

The CAM-Brain Machine (CBM) is a more specialized device, devoted to the evolution of CA based neural network circuit modules which are downloaded into a gigabyte of RAM. Once all the (maximum 64k) modules are downloaded, the CBM is used to update this space at a rate of 130 billion CA cells a second, which is some 650 times faster than the CAM-8. The CAM-8 is a general CA hardware accelerator. The CBM is exclusively for neural nets, unless one reprograms the FPGAs it contains. The CAM-8 has been used mainly in applications of CA simulated fluid flows, electromagnetic wave simulations etc. The CBM has been constructed with the specific aim of building artificial brains, although one company, Belgium's Lernout and Hauspie (L&H) has bought one for speech processing feasibility studies.

9.2. *Eldredge and Hutchings' Runtime Reconfigurable Neural Net Hardware*

Eldridge and Hutchings use a run time reconfigurable (FPGA) hardware system (called RRANN) to execute a backpropagation learning algorithm in a feed forward neural net [16]. They divide the learning task into three phases (feed-forward, backpropagation, and update), each with its own circuitry, which is configured consecutively into the FPGAs during run time. They achieve a greater efficiency in silicon use this way, since without the reconfiguration, far more silicon would be needed and for most of the time would not be used.

There are several similarities and contrasts which can be made between RRANN and the CBM. Both use run time reconfiguration, although in the case of RRANN, the reconfiguring occurs only twice (between the three phases) whereas the CBM is constantly reconfiguring its FPGAs into (neural net) growth mode and signaling mode, for each generation for hundreds of generations in the genetic algorithm. The CBM deals with millions of artificial neurons, whereas the RRANN deals with far fewer, the nature of the RRANN task being quite different. RRANN uses a non evolutionary approach, as distinct from the CBM. RRANN limits itself to neural nets that use feed forward signaling (characteristic of the backprop algorithm). CBM uses an evolutionary approach for which the internal complexity of the neural circuitry that is evolving is irrelevant, and hence can be much more complex in its structure and dynamics, and hopefully, because of that, more performant than feedforward networks.

9.3. *Sony's Pet Dog Robot 'Aibo'*

Our team thought that an artificial brain without a body for it to control would be rather useless, so we decided it would be a good idea to have it control a cute lifesized kitten robot called Robokoneko. However, after we conceived Robokoneko, SONY Corporation of Japan, unveiled its plans to make a similar robot pet, which they eventually called "Aibo" which is Japanese for "pal, mate, partner". It is about the size of a living Chiwawa dog, whose image can be seen at [17]. It has a limited number of behaviors (a dozen or so) which include, walking, turning, following and kicking a ball, getting on its feet, wagging its tail, etc. It is controlled by a few onbody microchips and costs a few thousand dollars.

The kitten robot is rather similar in concept, except that it will be controlled by an artificial brain, which will be orders of magnitude more sophisticated and performant than Aibo's microprocessors. Since the first generation artificial brain controlled by CBM-1 can contain 64k evolved neural net modules, we can afford to be ambitious. (We plan by about 2001, to have a second generation machine CBM-2, to handle a billion neuron artificial brain with a million modules, i.e. 16 times more). We can give the kitten robot hundreds of behaviors, thousands of pattern recognizers etc., and have it switch between these many behaviors depending upon its moods, its drives, its internal states (such as curiosity, hunger, boredom), its external stimuli etc. To the casual observer, the difference in the behavioral repertoire and general intelligence levels of Aibo and Robokoneko should be marked. However, Robokoneko is still a concept, whereas Aibo is already a product, due to the greater human resources SONY was able to give to its development. Robokoneko is much more of a research project, as nobody really understands yet how to build an artificial brain.

10. Conclusions

This article has provided an overview of ATR's CAM-Brain Machine (CBM) and the Artificial Brain ("CAM-Brain") Project of which the CBM is the project's

المنارة للاستشارات
www.manaraa.com

fundamental tool. The CBM should be delivered to ATR in the third quarter of 1999. The CBM will update 130 billion 3D CA cells a second and evolve a CA based neural net module in about 1 second. This speed should make practical the assemblage of tens of thousands of evolved neural net modules into humanly defined artificial brain architectures, and hopefully create a new research field that we call simply "Brain Building". This article has discussed the neural net model ("CoDi-1Bit") which is implemented by the CBM. Also presented were discussions on how to convert back and forth between analog time dependent signals and spiketrains (bit strings of 0 s and 1 s), thus enabling users to think entirely in terms of analog input and target output signals. A sample of evolved neural network modules using the CoDi-1Bit model was given. Once the CBM is delivered and sufficient experience with it enables the construction of large neural systems, with tens of thousands of modules, an artificial brain will be designed and built to control the behavior of a robot kitten called "Robokoneko". The challenges which remain in the CAM-Brain Project are to fully test the limits of the evolvability of the CoDi-1Bit modules (using the CBM), so as to gain experience in what can be readily evolved and what cannot, and then to assemble large numbers of them to make Robokoneko's brain. The biggest challenge will probably be creating the brain's architecture, our main task for 1999, and beyond.

The CBM should be fast enough for many multi-module tests to be undertaken. Multi-module systems can be evolved, assembled into the RAM, and then tested as a functional unit. Once a system has been built successfully it can be used as a component in a larger system, ad infinitum. The challenges of the CAM-Brain Project are not only conceptual in nature, but managerial as well. A back of the envelope calculation says that if an evolutionary engineer (i.e. someone who evolves a neural net module using a CBM) takes half an hour of human thinking time to dream up the fitness definition (i.e. the performance criterion) of a module, to specify the module's input signal(s), its target output signal, its input and output links with other modules, etc., then 8 evolutionary engineers would be needed to complete the design of a 64k module artificial brain within 2 years. Thus one needs to speak in terms of brain builder teams. If one wants to be a lot more ambitious and build a million module artificial brain in 2 years, then 120 evolutionary engineers are needed. Such a large team would need managers to control them. One can imagine higher level "brain architects" handing out module specifications to lower level evolutionary engineers who actually evolve them on their CBMs and report back to the brain architects with the results. The brain architects and evolutionary engineers need not be located in one place. Modern internet telephone technologies, which we use successfully on a daily basis, make globally distributed "virtual teams" practical.

If artificial brains can be made to work reasonably successfully, e.g. by making interesting robot pets, or simple household cleaner robots, etc., then a new artificial brain based computer industry will probably be created. However, this will only be possible if machines such as the CBM can deliver sufficient "evolvability" to make it happen. By evolvability is meant the degree to which some desired functionality is evolvable by a given model and implementation. As evolutionary engineers quickly learn, not all neural net modules evolve as one would wish. For example, it is quite possible that the decision to limit the CoDi model to 1 bit neural signaling (in order to implement the model in the Xilinx XC6264 chips) has limited the evolvability of the CoDi neural net modules. The first author (de Garis) evolved neural net modules (in software) with 8–10 bit neural signals for his PhD a decade ago [18], and obtained a remarkable level of evolvability, but even then there were limits. Section 7 above has provided a taste of what CoDi-1Bit modules can do. Once our team has the CBM, we will be able to broaden rapidly our experience in CoDi module evolution and hence obtain a feel for its evolvability, within the constraints of 1 bit signaling and the CA based neural nets. We will then be more able to design an artificial brain based on modules that are evolvable in practice.

As Moore's law provides more powerful evolvable chips in future years, later versions of the CBM will be able to implement more complex neural net models, with multi bit signaling, with more realistic neuron models, etc., and hence provide a greater level of evolvability, a concept fundamental to the effort of building artificial brains. As an example of evolvability, consider the attempt to evolve a module whose output should be as close as possible to some time dependent wave form. In a recent paper submitted for publication [19], we evolved a module which followed a sinusoidal curve very closely (with 2–5% error) for about 40 clockticks, and then diverged. We then changed the neural net model (in simulation) by adding more bits to the model's GA chromosome. The evolved curve then followed the target curve for about 80 clocks before divergence. This increase in the module's "MEC"

المنارة للاستشارات
www.manaraa.com

("Modular Evolvable Capacity") was due presumably to the increase in the number of bits in the chromosome, giving the module a greater potential to generate a desired behavior for longer. Any model with a finite number of bits will obviously have a limit to how extensively it can be evolved to generate some desired function. We suspect that this concept of the MEC will play a fundamental role in future evolutionary engineering and particularly in brain building. Perhaps by combing modules in some way, it may be possible to extend the MEC of the whole indefinitely. This remains a challenge for future research. We feel that future generations of the CBM, using future generations of evolvable chips, will generate a steady increase in the MECs of the modules they evolve. Evolutionary engineers should be aware of the limitations of the evolutionary approach. They should be conscious of the concept of the MEC and the drive to increase their values (e.g. the number of clockticks during which the evolved curve follows closely the target curve before diverging).

Acknowledgments

The authors acknowledge the financial, contractual and managerial support of Katsunori Shimohara of ATR/NTT. Shimohara believed in the possibility of building an artificial brain since de Garis proposed it to ATR in 1992. The authors also acknowledge the anonymous reviewers who did a fine job in providing suggestions to improve this paper.

References

1. M. Korkin, H. de Garis, F. Gers, and H. Hemmi, "CBM (CAM-Brain Machine): A Hardware Tool Which Evolves a Neural Net Module in a Fraction of a Second and Runs a Million Neuron Artificial Brain in Real Time," *Genetic Programming 1997: Proceedings of the Second Annual Conference*, J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba, and R.L. Riolo (Eds.), July 1997.
2. Xilinx, Inc., *The Programmable Logic Data Book 1996*, 1996.
3. D. Rumelhart and J.L. McClelland, *Parallel Distributed Processing: Explorations in the Microstructure of Cognition*, Cambridge, MA: MIT Press/Bradford Books, 1986.
4. H. de Garis, "An Artificial Brain: ATR's Cam-Brain Project Aims to Build/Evolve an Artificial Brain With a Million Neural Net Modules Inside a Trillion Cell Cellular Automata Machine," *New Generation Computing Journal*, vol. 12, no. 2, 1994.
5. H. de Garis, F. Gers, M. Korkin, A. Agah, and N. Eiji Nawa, "Building an Artificial Brain Using an FPGA Based 'CAM-Brain Machine'," *Artificial Life and Robotics Journal*, to appear.
6. F. Gers, H. de Garis, and M. Korkin, "CoDi-1 Bit: A Simplified Cellular Automata Based Neuron Model," *Proceedings of AE97, Artificial Evolution Conference*, Oct. 1997.
7. D.E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, 1989.
8. F. Rieke, D. Warland, R. de Ruyter van Steveninck, and W. Bialek, *Spikes: Exploring the Neural Code*, Cambridge, MA: MIT Press/Bradford Books, 1997.
9. M. Korkin, N. Eiji Nawa, and H. de Garis, "A 'Spike Interval Information Coding' Representation for ATR's CAM-Brain Machine (CBM)," *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES'98)*, Springer-Verlag, Sept. 1998.
10. E. Sanchez and M. Tomassini, (Eds.), *Towards Evolvable Hardware: The Evolutionary Engineering Approach*, Springer-Verlag, 1996. Lecture Notes in Computer Science, No. 1062.
11. T. Higuchi, M. Iwata, and W. Liu (Eds.), *Evolvable Systems: From Biology to Hardware*, Springer-Verlag, 1997. Lecture Notes in Computer Science, No. 1259.
12. A. Thompson and P. Layzell, "Analysis of Unconventional Evolved Electronics," *Communications of the ACM*, vol. 42, no. 4, pp. 71–79, 1999.
13. An Evolutionary Engineering Consultancy Company based in Boulder, Colorado, at URL `http://www.genobyte.com`.
14. H. de Garis, N. Petroff, M. Korkin, and G. Fehr, "Simulation and Evolution of the Motions of a Life Sized Kitten Robot 'Robokoneko' to be Controlled by a 32000 Neural Net Module Artificial Brain," Submitted to publication, available on website `http://www.hip.atr.co.jp/~degaris/papers/JCG.html`.
15. T. Toffoli and N. Margolus, *Cellular Automata Machines*, Cambridge, MA: MIT Press, 1987.
16. J.G. Eldredge and B.L. Hutchings, "Rrann: The Run-Time Reconfiguration Artificial Neural Network," *Proceedings of the Custom Integrated Circuits Conference*, May 1994. Available at http://splish.ee.byu.edu/docs/cicc94.rrann.ps.
17. MSNBC, "Sit, Aibo, Sit!, MSNBC, Technology Report. Available on website `http://www.msnbc.com/news/268649.asp`.
18. H. de Garis, "Genetic Programming: GenNets, Artificial Nervous Systems, Artifical Embryos," Ph.D. Thesis, Brussels University, Jan. 1992. Available at http://www.hip.atr.co.jp/~degaris.
19. H. de Garis, A. Buller, M. Korkin, F. Gers, N. Eiji Nawa, and M. Hough, "ATR's Artificial Brain (CAM-Brain) Project: A Sample of What Individual CAM-Brain Modules Can Do With Digital and Analog I/O," *Proceedings of the First NASA/DoD Workshop on Evolvable Hardware*, Pasadena, California, July 19–21, 1999, IEEE Computer Society, ISBN 0-7695-0256-3.

المنارة للاستشارات
www.manaraa.com

Prof. Dr. Hugo de Garis is head of the Brain Builder Group at ATR Labs in Kyoto, Japan. He is the father of the rapidly growing research field "evolvable hardware", a concept he got off the ground in 1992. He uses evolvable hardware techniques to evolve neural network circuit modules at electronic speeds using FPGA based hardware. He is assembling 64000 of these modules in RAM to build a 75 million neuron artificial brain. He obtained his Ph.D. in artificial nervous systems in 1991 from the University of Brussels (ULB), Belgium in 1991. From February 2000, he will be continuing his artificial brain work at Starlab, in Brussels (http://www.starlab.org).
degaris@hip.atr.co.jp

Dr. Michael Korkin received his M.S. degree in Computer Systems Engineering from MIIT, Moscow, Russia, in 1982, and his Ph.D. degree in Digital Image Processing from MPI, Moscow, in 1988. In 1991–1997 he worked as a Senior Hardware Engineer at a medical imaging firm in Denver, Colorado, USA. He founded his company Genobyte Inc. in 1997 in Boulder, Colorado. His primary research interests are evolvable hardware, artificial brain building, and neuroscience.
korkin@genobyte.com

للاستشارات المنارة

www.manaraa.c