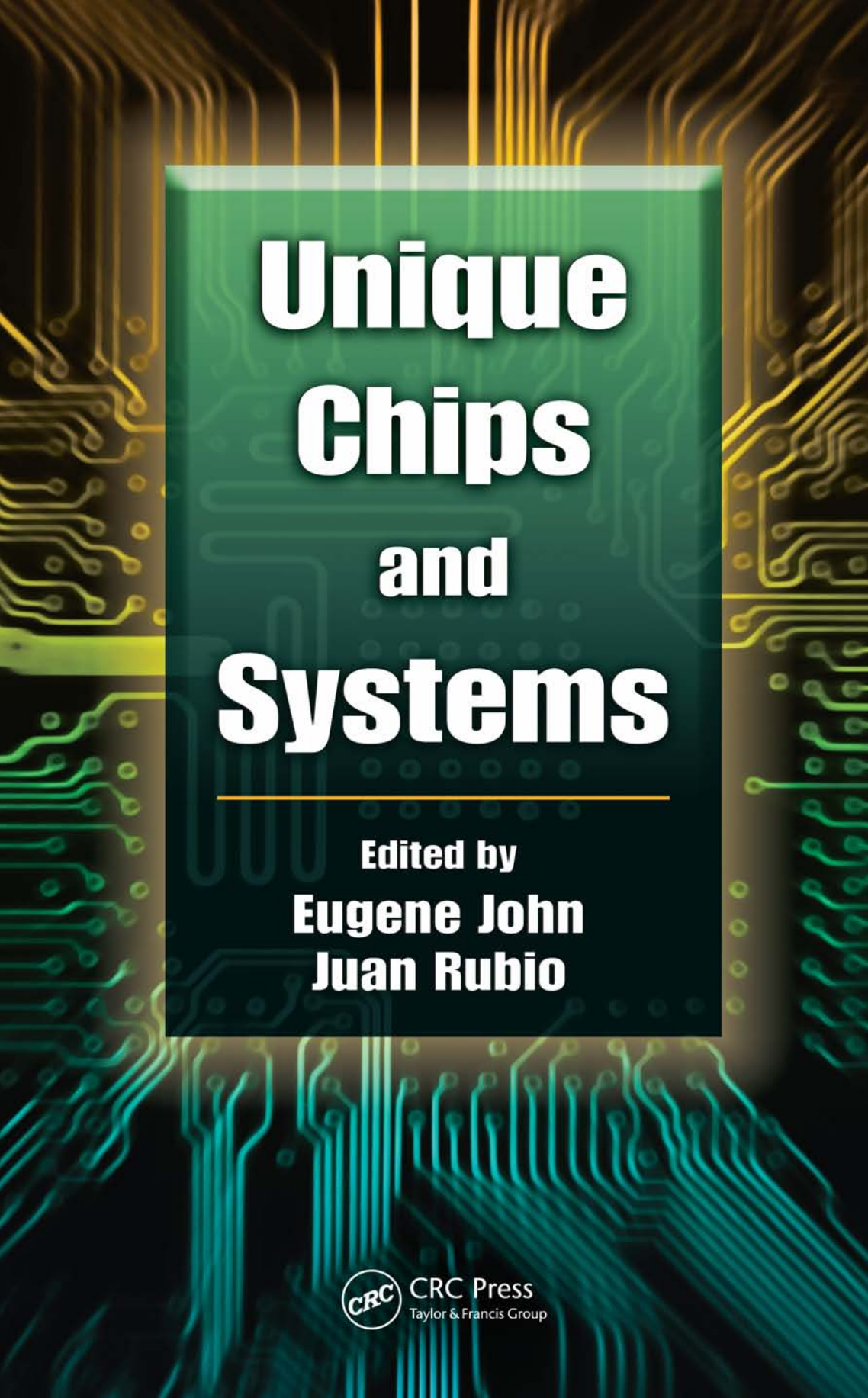

Unique
Chips
and
Systems
Edited by
Eugene John
Juan Rubio
CRC
CRC Press
Taylor & Francis Group

Unique Chips and Systems

Computer Engineering Series

Series Editor: Vojin Oklobdzija

Coding and Signal Processing for Magnetic Recording Systems
Edited by Bane Vasic and Erozan M. Kurtas

Digital Image Sequence Processing, Compression, and Analysis
Edited by Todd R. Reed

Low-Power Electronics Design
Edited by Christian Piguet

Unique Chips and Systems
Edited by Eugene John and Juan Rubio

Unique Chips and Systems

Edited by
Eugene John
Juan Rubio

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
10 9 8 7 6 5 4 3 2 1

International Standard Book Number-13: 978-1-4200-5174-2 (Hardcover)

Library of Congress Cataloging-in-Publication Data

John, Eugene.
Unique chips and systems / Eugene John and Juan Rubio.
p. cm.
Includes bibliographical references and index.
ISBN 978-1-4200-5174-2 (hbk. : alk. paper)
1. Systems on a chip. 2. Computer networks--Equipment and supplies. I. Rubio, Juan, 1973- II. Title.

TK7895.E42J64 2007
621.3815--dc22 2007024835

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

Preface

Integrated circuits are the enabling technology for the modern information age. Advanced systems are built using state-of-the-art semiconductor chips. Computing, communication, and network chips fuel the information technology era. The demands of emerging software applications can be met only with unique chips and systems. The integration ability presented by modern semiconductor technology presents opportunities; however, the requirements posed by power consumption, reliability, and form factor present challenges. This book presents fourteen chapters dealing with several systems and chips that present unique approaches to designing future computing and communication chips and systems.

Chapter 1 presents the TRIPS processor architecture and microarchitecture. TRIPS is a unique architecture that seeks to better exploit uniprocessor-level concurrency by changing the way instruction-level concurrency is expressed to the hardware, thereby extending the scaling of uniprocessors and enabling more efficient multiprocessors. TRIPS uses an explicit data graph execution (EDGE) instruction set architecture to efficiently encode concurrency in its dataflow execution model. The TRIPS microarchitecture uses a distributed, tiled microarchitecture that supports dynamic out-of-order execution. It is partitioned for scalability and implements deep speculation and latency tolerance.

Chapter 2 describes the Centaur Technology x86 processor with several data security features. Centaur Technology (a part of VIA Technologies Inc.) integrated several security features into the x86 processor, with little increase in die size or development effort. The chapter presents the hardware security features, and describes the implementation of the AES encryption hardware, the secure hash algorithm (SHA) hardware and the Montgomery multiplier—all aimed at improving the security of the processor.

Chapter 3 presents the ARM Cortex-A8 processor, a sub-1 watt processor that provides high performance for general purpose and media applications. The processor performs superscalar execution; yet, it is designed to be energy efficient. The microarchitecture, machine efficiency, and operating frequency are decided with energy efficiency as a primary criterion. Multimedia and graphics applications are supported with a 64-bit SIMD unit.

Chapter 4 presents a highly parallel signal processor, the RACE-Hypercube processor, which achieves up to 1 trillion bytes/sec at a relatively low clock frequency of 250 MHz. The processor allows the selection of a variety of configuration parameters.

Chapter 5 presents an asynchronous FPGA design—the RASTER architecture. The challenges and limitation of a clocked design are overcome with a self-timed (asynchronous) design, resulting in higher performance per watt. The RASTER architecture consists of an FPGA logic cell that uses a unique method of intercell communication. Simulation shows data throughput rates of up to 1.3 GHz at the 90nm process on a benchmarking suite of small FPGA designs.

Chapter 6 presents another unique chip—the continuation-based Fuce multithreading processor. The Fuce processor from Kyushu University, Japan, is based on the dataflow computing model. The Fuce processor pursues parallel execution of threads with high parallel processing and compatibility. Fuce means "fusion of communication and execution." The Fuce processor executes multiple threads using the exclusive multithread execution model, which is derived from dataflow computing. The Fuce processor aims to fuse the interprocessor execution and interprocessor communication. The Fuce processor unifies processing inside the processor and communication with external processors using events and threads.

Chapter 7 is a study of a processor with dual thread execution modes. The authors present the use of additional cores on a processor for two purposes: (1) to execute subordinate threads, and (2) to execute speculative threads. Threads are spawned to the available processing cores to exploit thread-level parallelism. Performance analysis using SPEC CPU2000 benchmarks show higher improvement using subordinate threads rather than speculative threads. A processor that can switch execution modes between the two approaches is also investigated since many applications alternate between different types of phases during their execution. Such an adaptive processor is seen to be 17 percent better than the subordinate thread mechanism alone.

Adaptive power management of computer systems has become extremely important in recent years. Such techniques heavily rely on variation of power during execution of applications. Chapter 8 presents power phases in commercial and scientific workloads running on enterprise-class hardware. Power consumption of CPU, I/O, and disk subsystems is measured using power sensors and phase behavior of applications is studied.

Future chips are driven by emerging and future applications. A workload that is most demanding of computational power and speed is computer graphics and visualization. Gaming has driven this quest for function and speed to such a point that graphics chips, independent of the driving computer system, have more gates than the latest CPU and many times the arithmetic power. And yet, there are aspects of graphics that still overly consume the power of systems. In Chapter 9, example graphics applications that need enormous computing power are presented. The author seeks to provide compact geometric representations of shapes so that rendering (displaying on the screen) can be more efficiently performed. He shows a close relationship between quadratic Bezier curves (QBCs) and iterated function systems (IFSs) to manipulate 2D sets that resemble 3D sets in the real world. He also

demonstrates the value of segmenting 3D triangle meshes that represent human teeth, thus dramatically accelerating visualization processes.

In Chapter 10, the authors illustrate the use of hardware accelerators built from field programmable gate arrays (FPGAs), graphic processing units (GPUs), or SIMD processor arrays for high performance computing. Such a system can be considered as a two-level processing system, consisting of the conventional processing nodes and the acceleration hardware connected over a high-speed network. In this chapter, researchers from the Los Alamos National Laboratory describe the use of such systems for a class of applications that use wavefront algorithms. These algorithms are characterized by a specific order in which cells are processed. The improvement in performance from accelerators such as the Clearspeed CSX600 SIMD accelerator is presented.

In Chapter 11, characteristics of a bioinformatics application are presented. Computational biology has become an important workload for high performance computers. Multiple-sequence alignment applications are important bioinformatics applications. Twelve multiple sequence alignment programs with a variety of alignment approaches are analyzed for performance of the cache, trace cache, branch predictor, phase behavior, and so on.

Embedded systems are inherently real time systems—they must control and compute as demanded by events. And the larger systems they are part of may demand a significant number of parallel processes going; for example, the most lavishly outfitted BMW automobile has an excess of 100 microcontrollers in charge of its many operations. Ravenscar is a subset of the Ada programming language designed for real-time computing. In Chapter 12, the authors present a Ravenscar, hardware-implemented run-time kernel with delay queues that allows for accurate analysis of application timing behavior. Formal state models and their simulations as well as hardware implementation are presented. The authors describe the corresponding VHDL state machines and demonstrate that the required levels of parallelism, hardware requirements, and timing granularity can be achieved.

In Chapter 13, an error correction scheme for a network-on-chip (NOC) is presented. The increased susceptibility of on-chip networks to various sources of error necessitates strategies to handle errors. A forward error correction scheme employing a low density parity check code (LDPC) is presented in this chapter. The presented LDPC is a linear block code suitable for low latency, high gain, and low power design because of its streamlined forward-only data flow structure.

Chapter 14 presents silicon-based on-chip optical interconnects and their use in reducing thermal constraints in a high performance clustered multithreaded processor. Increased integration in modern semiconductor technologies often results in regions of the chip with very high power densities or hot spots. One technique to reduce the thermal concerns from the hot spots is to intermix hot and cold units, however, at the cost of increasing communication distances between blocks. Silicon-based optical interconnects are shown to

be very valuable for global communication paths in such chips. A significant reduction in thermal constraints without reducing performance is shown in connecting the common front-end with the distributed back-end of a clustered multithreaded processor.

We hope that the readers of this book enjoy the variety of unique systems and chips presented. Most of the chapters in this book are revised versions of selected papers presented at the first, second, and third Workshop on Unique Chips and Systems (UCAS). The first and second UCAS workshops were held in March 2005 and March 2006 in Austin, Texas, and the third UCAS workshop was held in San Jose, California, in April 2007. We would like to thank the authors of the chapters for their contributions. We also wish to thank all those who helped in the process, especially Nora Konopka and Jessica Vakili at CRC Press/Francis & Taylor.

Eugene John
University of Texas at San Antonio

Juan Rubio
IBM Austin Research Laboratory

Editors

Eugene John is a professor in the department of electrical and computer engineering at the University of Texas at San Antonio. He received his Ph.D. in electrical engineering from Pennsylvania State University in 1995. His current research interests include low power circuits and systems, VLSI design, power estimation and optimization, multimedia and network processors, computer architecture, performance evaluation, and biometrics. He is a senior member of the IEEE, IEEE Computer Society, and IEEE Circuits and Systems Society. He is also a member of Eta Kappa Nu, Tau Beta Pi, and Phi Kappa Phi Honor societies.

Juan Rubio is a research staff member at the IBM Austin Research Lab. His interests include computer system architecture, performance analysis, and control systems. At IBM, he explores techniques to model, monitor and manage power; temperature and performance in computer servers; and data centers. His research contributed directly to the development of PowerExecutive™ and EnergyScale™ technology used in IBM systems.

Rubio received a B.S. in electrical engineering from Universidad Santa Maria La Antigua, Panama, in 1997, and an M.S. and Ph.D. in computer engineering from the University of Texas at Austin in 2004.

Contributors

David H. Albonesi Cornell University, Ithaca, New York

Makoto Amamiya Kyushu University, Fukuoka, Japan

Satoshi Amamiya Kyushu University, Fukuoka, Japan

Frank Barry Onward Communications Inc. and Appalachian State University, Boone, North Carolina

Praveen Bhojwani Texas A&M University, College Station, Texas

W. Lloyd Bircher University of Texas at Austin, Texas

Gregory Briggs University of Rochester, Rochester, New York

Doug Burger University of Texas at Austin, Texas

Guoqing Chen University of Rochester, Rochester, New York

Hui Chen University of Rochester, Rochester, New York

Gwan Choi Texas A&M University, College Station, Texas

Tom Crispin Centaur Technology Inc., Austin, Texas

Rajagopalan Desikan University of Texas at Austin, Texas

Saurabh Drolia University of Texas at Austin, Texas

Philippe M. Fauchet University of Rochester, Rochester, New York

Manoj Franklin University of Maryland, College Park, Maryland

Eby G. Friedman University of Rochester, Rochester, New York

Brian C. Gaide University of Texas at Austin, Texas

M. S. Govindan University of Texas at Austin, Texas

Paul Gratz University of Texas at Austin, Texas

Divya Gulati University of Texas at Austin, Texas

Heather Hanson University of Texas at Austin, Texas

Mikhail Haurylau University of Rochester, Rochester, New York

G. Glenn Henry Centaur Technology Inc., Austin, Texas

Adolfy Hoisie Los Alamos National Laboratory, Los Alamos, New Mexico

Masaaki Izumi Kyushu University, Fukuoka, Japan

Chand T. John Stanford University, Stanford, California

Lizy Kurian John University of Texas at Austin, Texas

Stephen W. Keckler University of Texas at Austin, Texas

Darren J. Kerbyson Los Alamos National Laboratory, Los Alamos, New Mexico

Changkyu Kim University of Texas at Austin, Texas

Tao Li University of Florida, Gainesville, Florida

Haiming Liu University of Texas at Austin, Texas

Kristina Lundqvist Massachusetts Institute of Technology, Cambridge, Massachusetts

Rabi Mahapatra Texas A&M University, College Station, Texas

Rania Mameesh University of Maryland, College Park, Maryland

Takanori Matsuzaki Kyushu University, Fukuoka, Japan

Robert McDonald University of Texas at Austin, Texas

Ramadass Nagarajan University of Texas at Austin, Texas

Nicholas Nelson University of Rochester, Rochester, New York

Terry Parks Centaur Technology Inc., Austin, Texas

Gerald G. Pechanek Lightning Hawk Consulting Inc. Cary, North Carolina, and Priest & Goldstein, PLLC, Durham, North Carolina

Nikos Pitsianis Duke University, Durham, North Carolina

Nitya Ranganathan University of Texas at Austin, Texas

Karthikeyan Sankaralingam University of Texas at Austin, Texas

Simha Sethumadhavan University of Texas at Austin, Texas

Sadia Sharif University of Texas at Austin, Texas

Premkishore Shivakumar University of Texas at Austin, Texas

Rohit Singhal Texas A&M University, College Station, Texas

Mihailo Stojancic ViCore Technologies Inc., Palo Alto, California

David Williamson ARM Inc., Austin, Texas

1

Architecture and Implementation of the TRIPS Processor

Stephen W. Keckler, Doug Burger, Karthikeyan Sankaralingam, Ramadass Nagarajan, Robert McDonald, Rajagopalan Desikan, Saurabh Drolia, M. S. Govindan, Paul Gratz, Divya Gulati, Heather Hanson, Changkyu Kim, Haiming Liu, Nitya Ranganathan, Simha Sethumadhavan, Sadia Sharif, and Premkishore Shivakumar

The University of Texas at Austin

CONTENTS

1.1 Introduction

Growing on-chip wire delays, coupled with complexity and power limitations, have placed severe constraints on the issue-width scaling of centralized superscalar architectures. As a result, recent microprocessor designs have backed away from powerful uniprocessors, instead favoring multiple simpler cores on a single die. Partitioning the chip into a collection of processors communicating via a common memory system mitigates some of the technology scaling challenges, but increases the burden on software to provide multiple threads to execute concurrently across the cores.

An alternative is to pursue more powerful uniprocessors, but design them so that they are scalable and tolerant of technology and complexity scaling. Ideally, such wide-issue processors would be *tiled* [30], meaning composed of multiple replicated, communicating design blocks. Because of multicycle communication delays across these large processors, control must be distributed across the tiles. We advocate the use of microarchitectural networks (or *micronets*) for routing control and data among the tiles. Micronets provide high-bandwidth, flow-controlled transport for control or data in a wire-dominated processor by connecting the multiple tiles, each of which is a client on one or more micronets. Higher-level microarchitectural protocols direct global control across the micronets and tiles in a manner invisible to software.

In this chapter, we describe the architecture and implementation of the Tera-op, Reliable, Intelligently-adaptive Processing System (TRIPS) processor—a distributed, tiled microarchitecture. In particular, we discuss TRIPS tile partitioning, micronet connectivity, and distributed protocols that provide global services in the TRIPS processor, including distributed fetch, execution, flush, and commit. Although some of our prior publications have described the TRIPS approach to exploiting parallelism as well as high-level performance results [20,3], this chapter examines in detail the intertile connectivity and protocols that have resulted from reducing the high-level design to silicon. The key concepts that differentiate TRIPS from other tiled architectures such as RAW [30] are the dynamic scheduling and execution which require distributed dynamic hardware protocols to provide the means to extract both irregular and regular concurrency.

To understand the design complexity, timing, area, and performance issues of this dynamic tiled approach, we implemented the TRIPS design in a 170M transistor, 130 nm ASIC chip. This prototype chip contains two processor cores, each of which implements an EDGE instruction set architecture [3], is up to four-way multithreaded, and can execute a peak of 16 instructions per cycle. Each processor core contains five types of tiles communicating across seven micronets: one for data, one for instructions, and five for control used to orchestrate distributed execution. TRIPS prototype tiles range in size from 1–9 mm^2. Four of the principal processor elements—instruction and data caches, register files, and execution units—are each subdivided into replicated copies of their respective tile type; for example, the instruction cache is composed of five instruction cache tiles, and the computation core is composed of 16 execution tiles.

The tiles are sized to be small enough so that wire delay within the tile is less than one cycle, and so can largely be ignored from a global perspective. Each tile interacts only with its immediate neighbors through the various micronets, which have roles such as transmitting operands between instructions, distributing instructions from the instruction cache tiles to the execution tiles, or communicating control messages from the program sequencer. By avoiding any global wires or broadcast buses—other than the clock, reset tree, and interrupt signals—this design is inherently scalable to smaller processes, and is less vulnerable to wire delays than conventional designs. Preliminary performance results on the prototype architecture using a cycle-accurate simulator show that compiled code outperforms an Alpha 21264 on half of the benchmarks, and we expect these results to improve as the TRIPS compiler and optimizations are tuned. Hand optimization of the benchmarks produces IPCs ranging from 1.5–6.5 and performance relative to Alpha of 0.6–8.

The rest of the chapter is organized as follows. Section 1.2 describes the TRIPS ISA. Section 1.3 describes the microarchitecture of the various tiles that compose the processor. This is followed by Section 1.4 which describes TRIPS microarchitectural protocols. Section 1.5 describes the physical design of the TRIPS prototype and also discusses various overheads of a distributed design. This is followed by Section 1.6, which is about related work. Finally, Section 1.7 concludes.

1.2 ISA Support for Distributed Execution

TRIPS implements an Explicit Data Graph Execution (EDGE) instruction set architecture (ISA). We conceived it with the goal of high-performance, single-threaded, concurrent but distributed execution, by allowing compiler-generated dataflow graphs to be mapped to an execution substrate by the microarchitecture. The two defining features of an EDGE ISA are (1) block-atomic execution and (2) direct communication of instructions within a block, which together enable efficient dataflow-like execution.

The TRIPS ISA aggregates up to 128 instructions into a single block that obeys the block-atomic execution model, in which a block is logically fetched, executed, and committed as a single entity. This model amortizes the per-instruction bookkeeping over a large number of instructions and reduces the number of branch predictions and register file accesses. Furthermore, this model reduces the frequency at which control decisions about what to execute must be made (such as fetch or commit), providing the additional latency tolerance to make more distributed execution practical.

1.2.1 TRIPS Blocks

The compiler constructs TRIPS blocks and assigns each instruction to a location within the block. Each block is divided into between two and five 128-byte chunks, as shown in Figure 1.1. Every block includes a header chunk that encodes up to `32 read` and up to `32 write` instructions that access the 128 architectural registers. The read instructions pull values out of the

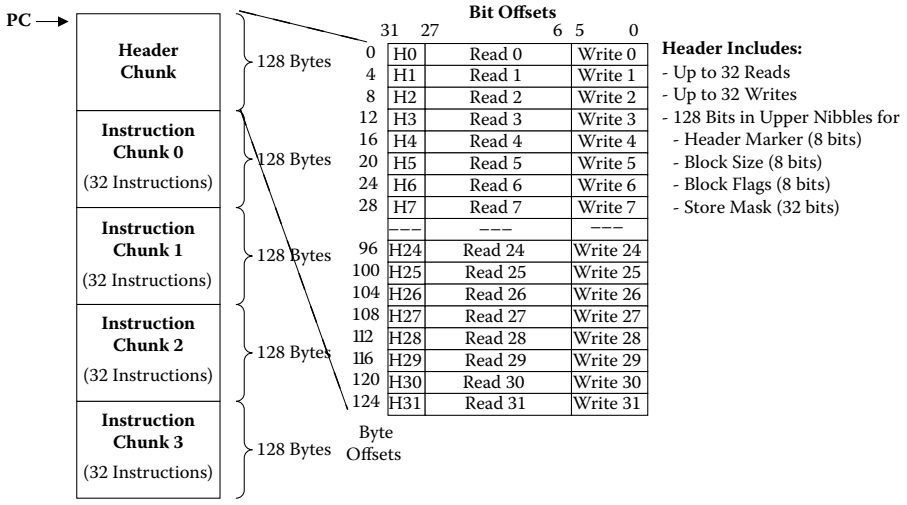


FIGURE 1.1
TRIPS Block Format.

registers and send them to compute instructions in the block, whereas the write instructions return outputs from the block to the specified architectural registers. In the TRIPS microarchitecture, each of the 32 read and write instructions are distributed across the four register banks, as described in the next section.

The header chunk also holds three types of control state for the block: a 32-bit "store mask" that indicates which of the possible 32 memory instructions are stores, block execution flags that indicate the execution mode of the block, and the number of instruction "body" chunks in the block. The store mask is used, as described in Section 1.4, to enable distributed detection of block completion.

A block may contain up to four body chunks—each consisting of 32 instructions—for a maximum of 128 instructions, at most 32 of which can be loads and stores. All possible executions of a given block must emit the same number of outputs (stores, register writes, and one branch) regardless of the predicated path taken through the block. This constraint is necessary to detect block completion on the distributed substrate. The compiler generates blocks that conform to these constraints [25].

1.2.2 TRIPS Instruction Formats

With *direct instruction communication*, instructions in a block send their results directly to intrablock dependent consumers in a dataflow fashion. This model supports distributed execution by eliminating the need for any intervening shared centralized structures (e.g., an issue window or register file) between intrablock producers and consumers.

Figure 1.2 shows that the TRIPS ISA supports direct instruction communication by encoding the consumers of an instruction's result as targets within the producing instruction. The microarchitecture can thus determine precisely where the consumer resides and forward a producer's result directly to its target instruction(s). The nine-bit target fields (T0 and T1) each specify the target instruction with seven bits and the operand type (left, right, predicate) with the remaining two. A microarchitecture supporting this ISA maps each of a block's 128 instructions to particular coordinates, thereby determining the distributed flow of operands along the block's dataflow graph. An instruction's coordinates are implicitly determined by its position in its chunk, as shown in Figure 1.1.

The instruction set also includes two additional nontraditional fields: `PR` and `LSID`. The `PR` field is included in nearly all of the instructions and encodes a predicate to control conditional execution of the instruction. This two-bit field determines whether the instruction is unpredicated, predicated on a true condition, or predicated on a false condition. Predicated instructions must wait for a predicate operand to arrive before executing. The TRIPS predicated execution model is integrated with the datagraph execution model, enabling efficient programming and new opportunities for compile-time optimization [26]. The `LSID` field is five bits and is included in every

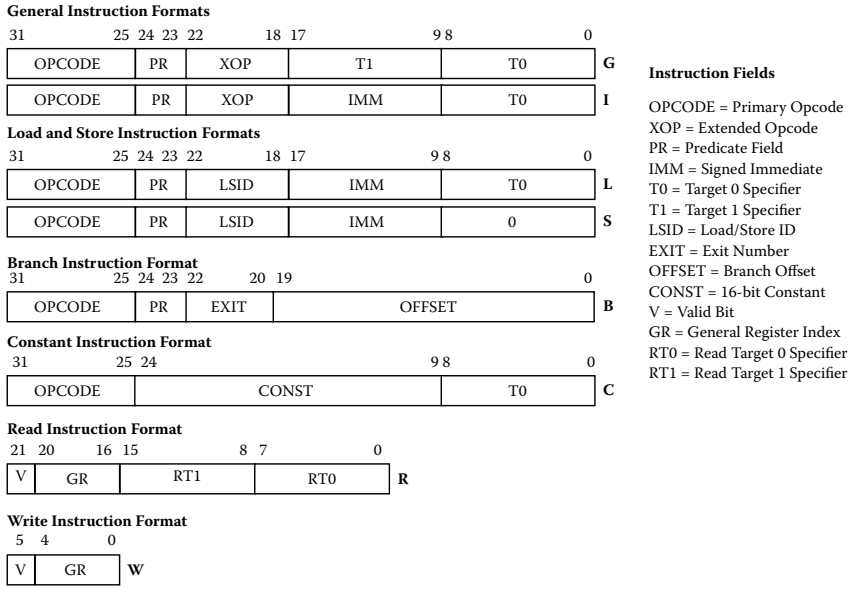


FIGURE 1.2
TRIPS Instruction Formats.

load and store instruction. This field encodes the original program order of load and store instructions within the block so that the TRIPS memory disambiguation hardware can determine when load instructions have illegally executed before store instructions on which they actually depend.

1.2.3 Code Generation

The TRIPS compiler accepts sequential programs and produces explicit datagraphs for execution by the TRIPS distributed hardware resources. Front-end compilation for an EDGE instruction set is similar to that of a conventional ISA, including standard scalar optimizations. The internal representation of the code after these optimizations is similar to standard three-operand RISC code. At this stage, the EDGE blocks are merely basic blocks, and are typically quite small.

The next stage of compilation is TRIPS block formation, which uses a number of techniques to grow the block size. These optimizations include but are not limited to loop unrolling, function inlining, and if-conversion using predication. TRIPS block formation, however, is constrained by the block-size restrictions discussed above. After block formation, a TRIPS program can be represented using our TRIPS intermediate language (TIL). This language appears similar to RISC assembly code but reflects the organization of the instructions into blocks and includes the predication and load–store identifiers. Further details on overall structure of the compiler and block formation can be found in [25, 17].

Finally, the TIL code is processed by the scheduler, which is in the back end of the compiler, to produce TRIPS assembly code (TASL). The scheduler is aware of the topology of the TRIPS core, including number of execution units and latencies between execution units. The schedule determines where each of the TRIPS instructions will execute in order to minimize the overall critical path of the block and the program. The placement of an instruction is implicitly encoded by the location of the instruction in the block chunks of Figure 1.1. Details on the algorithms employed in the scheduler can be found in [5].

Figure 1.3 shows the transformation of a simple sequence of RISC instructions into the TRIPS TIL and TASL representations. After TRIPS block formation, the basic block of Figure 1.3a is translated into the TIL code of Figure 1.3b. The TIL code clearly delineates the block boundaries with `begin` and `end` directives. The block also shows the read and write instructions that direct values in and out of the persistent architecture register file. The TIL code further includes the predication of the branch instructions that cause execution of either block 2 or block 3 after block 1 completes. The instructions in the TIL code still appear in *operand* format in which each instruction encodes named source and destination operands.

Figure 1.3c shows the final TRIPS assembly code after block scheduling. The instructions are the same as Figure 1.3b, but each instruction has been assigned a unique identifier indicated in brackets. This unique identifier is interpreted by the hardware as the physical coordinates of a reservation

(a) RISC basic block

```
ld R3,4(R2)
add R4,R1,R3
st R4,4(R2)
addi R5,R4,#2
beqz R4,Block3
```

(b) TRIPS TIL code

```
.bbegin block1
 read $t1,$g1
 read $t2,$g2
 ld $t3,4($t2)
 add $t4,$t1,$t3
 st $t4,4($t2)
 addi $t5,$t4,2
 teqz $t6,$t4
 b_t<$t6> block3
 b_f<$t6> block2
 write $g5,$t5
.bend block1

.bbegin block2
...
```

(c) TRIPS TASL code

```
[R1] $g1  [2]
[R2] $g2  [1] [4]
[1]  ld   L[1] 4 [2]
[2]  add  [3] [4]
[3]  mov  [5] [6]
[4]  st   S[2] 4
[5]  addi 2 [W1]
[6]  teqz [7] [8]
[7]  b_t  block3
[8]  b_f  block2
[W1] $g5
```

FIGURE 1.3
Transformation of RISC code to TRIPS assembly code.

station. The instructions are now in *target* format in which each instruction specifies the identifiers of the consumers of its result, but instructions do not encode from where their operands are coming. The load and store instructions now include the load–store sequence numbers (LSID) and the instruction stream includes a new `mov` instruction. This `mov` serves to fan out the result of the previous add instruction to a total of three targets. The `add` cannot complete this task itself because instruction encodings are limited to two targets for arithmetic instructions.

1.3 A Distributed Microarchitecture

The goal of the TRIPS microarchitecture is a processor that is scalable and distributed, meaning that it has no global wires, is built from a small set of reused components on routed networks, and can be extended to a wider-issue implementation without recompiling source code or changing the ISA. Figure 1.4 shows the tile-level block diagram of the TRIPS prototype that meets these specifications. The three major components on the chip are two processors and the secondary memory system, each connected internally by one or more micronetworks.

Each of the processor cores is implemented using five unique tiles: one global control tile (GT), 16 execution tiles (ET), four register tiles (RT), four data tiles (DT), and five instruction tiles (IT). The main processor core micronetwork is the operand network (OPN), shown in Figure 1.5. It connects all of the tiles except for the ITs in a two-dimensional, wormhole-routed, 5×5 mesh topology. The OPN has separate control and data channels, and can deliver one 64-bit data operand per link per cycle. A control header packet is launched one cycle in advance of the data payload packet to accelerate wakeup and select for bypassed operands that traverse the network.

Each processor core contains six other micronetworks, one for instruction dispatch—the global dispatch network (GDN)–and five for control: global control network (GCN), for committing and flushing blocks; global status network (GSN), for transmitting information about block completion; global refill network (GRN), for I-cache miss refills; data status network (DSN), for communicating store completion information; and external store network (ESN), for determining store completion in the L2 cache or memory. Links in each of these networks connect only nearest neighbor tiles and messages traverse one tile per cycle. Figure 1.5 shows the links for three of these networks.

This type of tiled microarchitecture is *composable* at design time, permitting different numbers and topologies of tiles in new implementations with only moderate changes to the tile logic, and no changes to the software model. The particular arrangement of tiles in the prototype produces a core with 16-wide out-of-order issue, 80 KB of L1 instruction cache, 32 KB of L1 data cache, and four SMT threads. The microarchitecture supports up to eight TRIPS blocks in flight simultaneously, seven of them speculative if a single thread

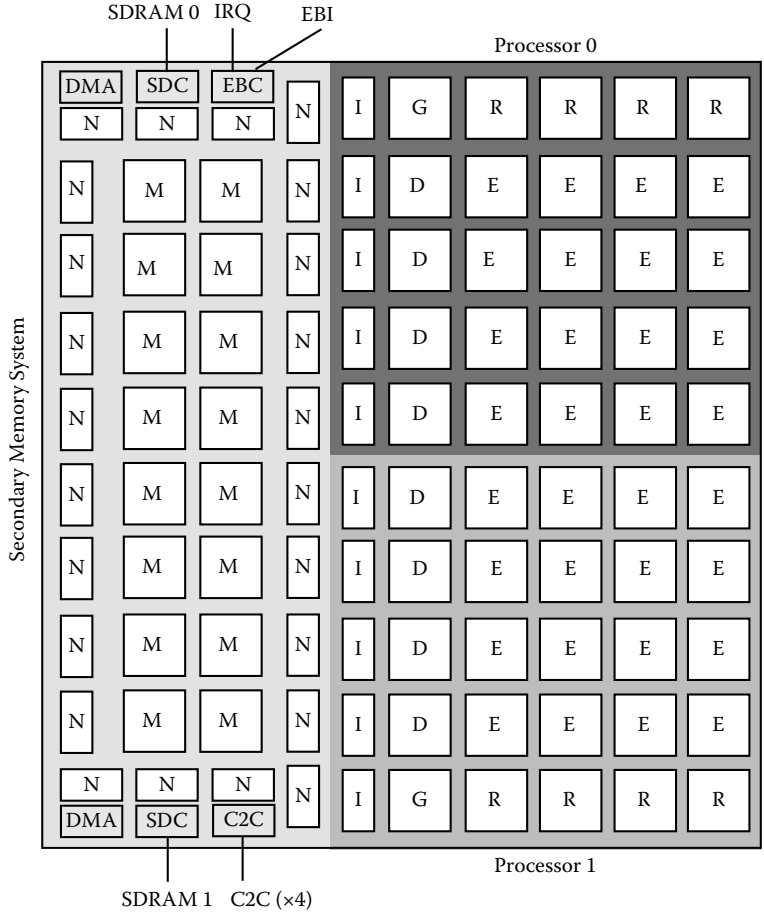


FIGURE 1.4
TRIPS prototype block diagram.

is running, or two blocks per thread if four threads are running. The eight 128-instruction blocks provide an in-flight window of 1,024 instructions.

The two processors can communicate through the secondary memory system, in which the On-Chip Network (OCN) is embedded. The OCN is a 4 × 10, wormhole-routed mesh network, with 16-byte data links and four virtual channels. This network is optimized for cache-line sized transfers, although other request sizes are supported for operations such as loads and stores to uncacheable pages. The OCN acts as the transport fabric for all interprocessor, L2 cache, DRAM, I/O, and DMA traffic.

1.3.1 Global Control Tile (GT)

Figure 1.6a shows the contents of the GT, which include the blocks' PCs, the instruction cache tag arrays, the I-TLB, and the next-block predictor. The GT

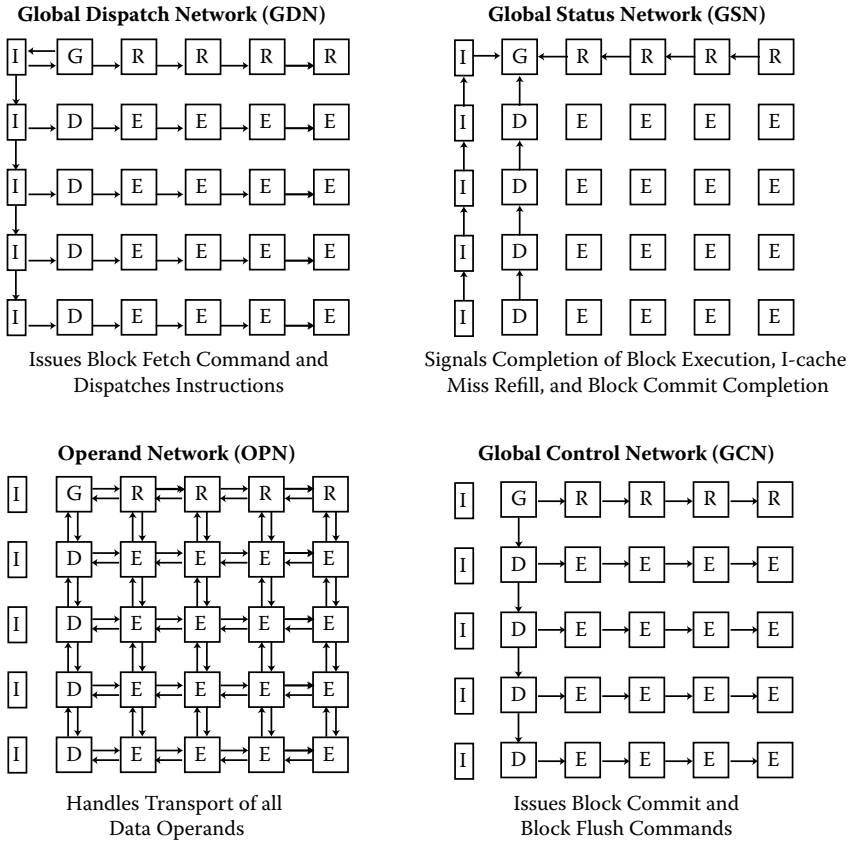


FIGURE 1.5
TRIPS micronetworks.

handles TRIPS block management, including prediction, fetch, dispatch, completion detection, flush (on mispredictions and interrupts), and commit. It also holds control registers that configure the processor into different speculation, execution, and threading modes. Thus the GT interacts with all of the control networks and the OPN, to provide access to the block PCs. The GT also maintains the current status of all eight in-flight blocks. When one of the block slots is free, the GT accesses the block predictor, which takes three cycles, and emits the predicted address of the next target block. Each block may emit only one "exit" branch, even though it may contain several predicated branches.

1.3.1.1 Fetch Unit

The fetch unit consists of a TLB (Translation-Lookaside Buffer) and a directory of the blocks that are resident in the I-cache. In addition, it contains the

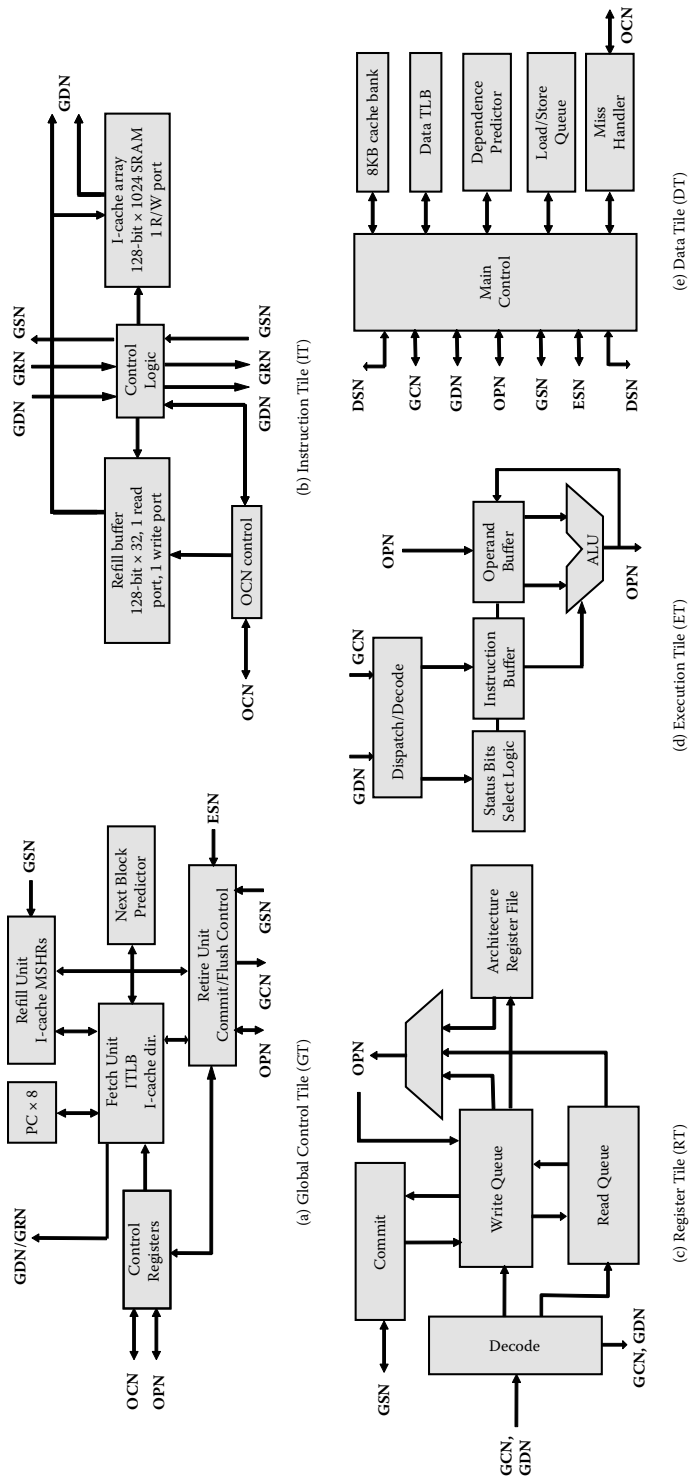


FIGURE 1.6
TRIPS tile-level diagrams.

program counters (PC) for each thread and control registers that are used to configure the execution of each block.

The instructions of a TRIPS block are striped across all of the ITs. For example, IT0 caches chunk 0 of a block and IT1 caches chunk 1 of the same block. The I-cache directory contains a listing of all blocks that are currently resident in the I-cache. The directory consists of 128 entries, organized in a two-way set-associative fashion, and each entry identifies a unique cached block. The directory is virtually indexed and entries are evicted and replaced in a LRU fashion.

The I-cache directory is similar to the tag array in conventional caches. In the TRIPS processor, the GT maintains a single array on behalf of all the ITs. This centralized directory provides a consistent view of the cached blocks and avoids scenarios where portions of a block are not present in the I-cache. The tag array in each IT can be eliminated, thus simplifying the implementation in both the GT and ITs. An alternate design could maintain the tag array as part of each IT. However, since a single block is striped across all ITs and each of them operates because in a distributed fashion, this approach would require special hardware to keep the tag arrays consistent.

The instruction TLB (ITLB) consists of a set of sixteen registers to provide the translations of virtual addresses of blocks to physical addresses. Each register defines the size and read/execute access attributes of up to sixteen memory segments. The minimum size of a memory segment is 64 KB and the maximum size is 1 TB. Instruction memory segments may be marked as uncacheable in the L1. A block in such a segment will never be filled into the I-cache. A miss in the ITLB or an access protection violation will generate an exception. Similar to the I-cache directory, implementing the ITLB inside the GT avoids redundant implementation in the ITs.

1.3.1.2 Refill Unit

The refill unit maintains the status of pending I-cache refills. The TRIPS processor supports up to four outstanding refills, but at most one per thread. To manage multiple outstanding refills, the GT tracks the I-cache set and the way being refilled and whether the refill has completed. The storage for tracking pending refills in the GT is similar to the I-cache MSHR (Miss Status Handling Register) state in conventional processors.

1.3.1.3 Retire Unit

The retire unit contains a retirement table, which tracks the execution state of all blocks in flight, and is responsible for initiating the flush, commit, and deallocation of the blocks in flight. Table 1.1 shows the details of the state maintained for each block. Some of the entries in this table are locally initialized by the GT, when it starts various block-level operations. Other state, which manages the completion and commit protocols, is updated as blocks produce their register, memory, and branch outputs. Neighboring

TABLE 1.1

State Tracked for Each Block in the Retirement Table

V	Valid block
O	Oldest block in thread
Y	Youngest block in thread
BADDR	Virtual address of the block
PADDR	Predicted address of the next block
RADDR	Actual resolved address of the next block
RC	Registers completed
SC	Stores completed
BC	Branch completed
RCOMM	Registers committed
SCOMM	Stores committed
E	Exception in block
F	Block already flushed

tiles deliver updates to this state to the GT using the TRIPS processor control networks.

The retirement table is similar to the reorder buffer (ROB) in conventional processors. However, this table does not track the status of individual instructions. It has only one entry for each block, thus containing far fewer entries than a conventional ROB.

1.3.1.4 Next Block Predictor

The next block predictor uses a branch instruction's three-bit exit field to construct exit histories instead of using taken/not-taken bits. The predictor has two major parts: an exit predictor and a target predictor. The predictor uses exit histories to predict one of eight possible block exits, employing a tournament local/gshare predictor similar to the Alpha 21264 [13] with 9 K, 16 K, and 12 K bits in the local, global, and choice exit predictors, respectively. The predicted exit number is combined with the current block address to access the target predictor for the next-block address. The target predictor contains four major structures: a branch target buffer (20 K bits), a call target buffer (6 K bits), a return address stack (7 K bits), and a branch type predictor (12 K bits). The BTB predicts targets for branches, the CTB for calls, and the RAS for returns. The branch type predictor selects among the different target predictions (call/return/branch/sequential branch). The distributed fetch protocol necessitates the type predictor; the predictor never sees the actual branch instructions, as they are sent directly from the ITs to the ETs.

The predictor performs three major operations: *predict*, *update*, and *repair*. Predict provides a prediction for the next block. Update modifies the predictor

tables with the information from a committing block. Repair corrects any predictor state modified by incorrect speculation. Each of the three operations consumes three processor cycles.

1.3.2 Instruction Tile (IT)

Figure 1.6b shows an IT containing a two-way, 16 KB bank of the total L1 I-cache. The ITs act as slaves to the GT, which holds the single tag array. Each of the five 16 KB IT banks can hold a 128-byte chunk (for a total of 640 bytes for a maximum-sized block) for each of 128 distinct blocks. An instruction cache tile also contains a 32-entry refill buffer with 128-bit entries. On an instruction cache miss, each IT will independently fetch its portion of the missed block into its own refill buffer. Later, when the block is dispatched, the contents of the refill buffer are delivered into the execution unit array and written into the instruction cache RAM. The refill buffer has room for four fetched blocks that are waiting to be dispatched.

1.3.3 Register Tile (RT)

To reduce power consumption and delay, the TRIPS microarchitecture partitions its many registers into banks, with one bank in each RT. The register tiles are clients on the OPN, allowing the compiler to place critical instructions that read and write from/to a given bank close to that bank. Because many def-use pairs of instructions are converted to intrablock temporaries by the compiler, they never access the register file, thus reducing total register bandwidth requirements by approximately 70%, on average, compared to a RISC or CISC processor. The four distributed banks can thus provide sufficient register bandwidth with a small number of ports; in the TRIPS prototype, each RT bank has two read ports and one write port. Each of the four RTs contains one 32-register bank for each of the four SMT threads that the core supports, for a total of 128 registers per RT and 128 registers per thread across the RTs.

In addition to the four per-thread architecture register file banks, each RT contains a read queue and a write queue, as shown in Figure 1.6c. These queues hold up to eight read and eight write instructions from the block header for each of the eight blocks in flight, and are used to forward register writes dynamically to subsequent blocks reading from those registers. The read and write queues perform a function equivalent to register renaming for a superscalar physical register file, but are less complex to implement due to the read and write instructions in the TRIPS ISA.

When a block dispatches, the read and write instructions are delivered to the RTs where they are captured in the read and write queues, respectively. Each read instruction queries the write queue, searching for write instructions from prior uncommitted blocks. If the read instruction finds that the most recent write queue entry to the sought after register has a valid value, it injects the value into the execution array. On the other hand, if it

finds that the most recent write instruction does not yet have a valid value, it stays in the read queue. When the prior block produces the register write, the pending register read wakes up and forwards the written value to its descendants. If the read instruction finds no matching write instruction in the write queue, it reads from the persistent register file. Register reads and forwarding of writes from prior blocks may be speculative and occur before the previous block commits. Speculation recovery involves invalidating the read and write queue entries of all affected blocks.

1.3.4 Execution Tile (ET)

As shown in Figure 1.6d, each of the 16 ETs consists of a fairly standard single-issue pipeline, a bank of 64 reservation stations, an integer unit, and a floating-point unit. All units are fully pipelined except for the integer divide unit, which takes 24 cycles. The 64 reservation stations hold eight instructions for each of the eight in-flight TRIPS blocks. Each reservation station has fields for two 64-bit data operands and a one-bit predicate. Arriving operands specify the exact reservation station to write into, eliminating the need for a CAM structure in the reservation stations; the TRIPS microarchitecture uses a lower power RAM structure instead.

Each ET is responsible for local wakeup and selection of instructions. According to the dataflow execution model, any instruction that has all of its operands present may be selected for execution. If more than one instruction is ready, the ET selects the oldest instruction, which corresponds to the earliest fetched instruction from the oldest block. When in the multithreaded configuration, the ET may be selecting instructions from up to four threads. In this case, selection priority rotates among the four threads, and within each thread the oldest instruction is given the highest priority.

When an instruction completes, the ET is responsible for forwarding the result to the target instructions. If the target resides on the same ET, the result can bypass directly to the target instruction for immediate execution; no bubbles between dependent instructions are required. If the target is remote, the ET injects the result into the operand network for delivery. If the instruction has multiple targets, the ET injects a separate message into the network for each target on successive cycles. When an operand arrives from a remote ET, the control packet arrives first and begins the wakeup process. If the ET selects the instruction receiving the operand, the instruction may rendezvous with the operand network data packet arriving on the next cycle and execute immediately. At a minimum, only one pipeline bubble is required between dependent instructions executing on adjacent ETs.

The ETs also implement the TRIPS predication and exception models. Unlike other predicated architectures, TRIPS only executes a predicated instruction if the arriving predicate polarity (true or false) matches the polarity of the predicated instruction. The ET matches arriving predicates to predicated instructions and keeps unmatched predicated instructions in their reservation stations until a matching predicate arrives. In TRIPS, an

instruction that causes an exception cannot immediately signal the problem because the instruction may be speculative. Instead, the ET generates a poison bit that propagates with the instruction through the datagraph execution. The processor will detect an exception only if the poison bit propagates to a block output (register write, store, or branch target) and the block is ready to commit. On a block flush or commit, the ET removes all of the flushed block's pending instructions from the reservation stations and any of the block's operand packets still propagating through the operand network.

1.3.5 Data Tile (DT)

Figure 1.6e shows a block diagram of a single DT. Each DT is a client on the OPN, and holds one 2-way, 8 KB L1 data cache bank, for a total of 32 KB across the four DTs. Virtual addresses are interleaved across the DTs at the granularity of a 64-byte cache-line. In addition to the L1 cache bank, each DT contains a copy of the load/store queue (LSQ), a dependence predictor, a one-entry back-side coalescing write buffer, a data TLB, and a MSHR that supports up to 16 requests for up to four outstanding cache lines.

1.3.5.1 Load Processing

The pipeline diagram in Figure 1.7 illustrates the different stages of load processing. Every incoming load accesses (a) the TLB to perform address translation and check the protection attributes, (b) the dependence predictor (DPR) to check for possible store dependences, (c) the LSQ to identify older matching uncommitted stores, and (d) the cache tags to check for cache hits. Based on the responses (hit/miss) from the four units, the control logic decides on the course of action for that load. Table 1.2 summarizes the possible load execution scenarios in the DT.

When the load hits in the cache, and only in the cache, the load reply can be generated in two cycles. This best-case latency is likely to be the common case for most loads. When a load hits both in the cache and the LSQ, the load return value is formed by composing the values obtained from the LSQ and cache. First, the load picks up any matching store's bytes from the LSQ and then reads the remaining bytes from the cache. This operation can take multiple cycles and is referred to as store forwarding.

A load may arrive at the DT before an earlier store on which it depends. Processing such a load right away will result in a dependence violation and a flush, leading to performance losses. To avoid this performance loss, the TRIPS processor employs a simple dependence predictor that predicts whether the load processing should be deferred. If the DPR predicts a likely dependence, the load waits in the LSQ until all prior stores have arrived. After the arrival of all older stores, the load is enabled from the LSQ, and allowed to access the cache and the LSQs to obtain the most recent updated store value.

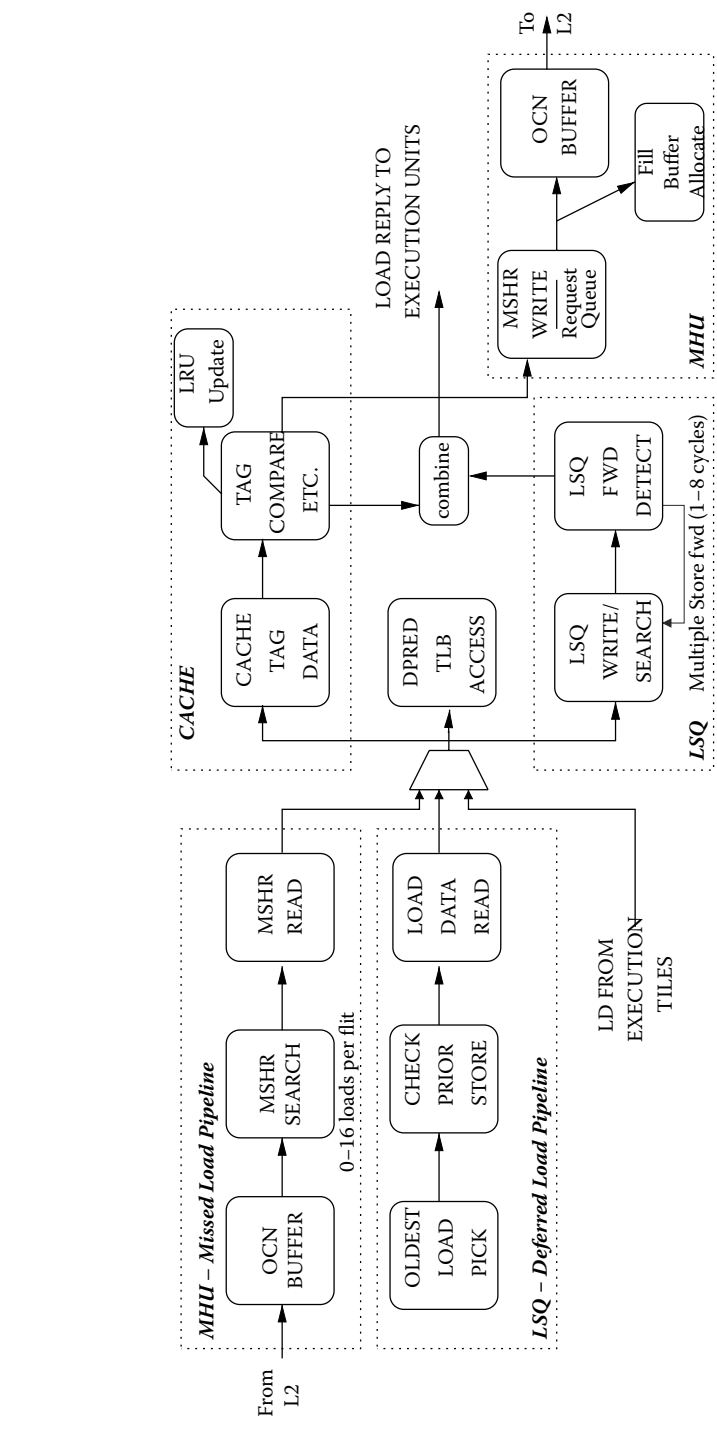


FIGURE 1.7
The DT load pipelines.

TABLE 1.2
Load Execution Scenarios

TLB	DPR	Cache	LSQ	Response
Miss	X	X	X	Report TLB Exception
Hit	Hit	X	X	Defer load until all prior stores are received
Hit	Miss	Hit	Miss	Forward data from cache
Hit	Miss	Miss	X	Forward data from L2 cache, issue cache fill request
Hit	Miss	Hit	Hit	Forward data from LSQ and cache

Note: X represents "don't care" state.

If the load misses in the cache, it is buffered in the MSHRs [16] and a read request is generated and sent to the L2. When the data is returned from the L2, the loads in the MSHRs are enabled and load processing resumes. As with deferred loads, missed loads also access the LSQ and cache to pick up the most recent store values.

1.3.5.2 Store Processing

Store processing occurs in two phases. During the first phase, each incoming store is buffered in the LSQ and the other DTs are notified about the store's arrival using the Data Status Network (DSN). During this phase each store checks for dependence violations; if any younger loads to the same address as the store are in the queue, then a violation is reported to the control unit, which initiates recovery. The dependence predictor is also trained to prevent such violations in the future.

When a block becomes nonspeculative, the second phase of store processing begins as illustrated in Figure 1.8. In this phase the oldest store is removed from the LSQ, checked in the TLB, and the store value is written out to the cache/memory system. If the store hits in the cache, the corresponding cache line is marked as dirty. If the store misses in the cache, the store miss request is sent to the L2. We chose a write-back, write-no allocate policy to minimize the number of commit stalls.

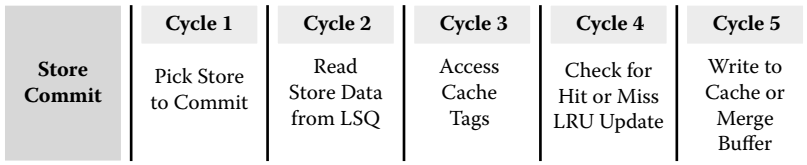


FIGURE 1.8
Store commit pipeline.

1.3.5.3 Store Tracking

In the TRIPS execution model, a block can commit only after all of its store outputs have been generated. When a store arrives at any DT, the store arrival information is broadcast to the other DTs through the DSN. Each DT then increments a local counter that counts the number of stores that have arrived at the memory system. When all of the stores in a block have been received, the DT that received the last store sends a message to the control tile indicating that all memory outputs have been generated.

1.3.5.4 Memory-Side Dependence Processing

Because the DTs are distributed in the network, we implemented a memory-side dependence predictor, closely coupled with each data cache bank [23]. Although loads issue from the ETs, a dependence prediction occurs (in parallel) with the cache access only when the load arrives at the DT. A naive extension of conventional dependence processing mechanisms [4] would hold back the load in the execution unit until the execution of the dependent store.

In TRIPS, the latency of dependent loads can be broken down into four parts: (1) the latency for the load to detect that the dependent store has executed, (2) the latency for the load to be delivered from the execution unit to the DT, (3) the latency to access the DT, and (4) the latency to deliver the value from the DT to the target of the load. With execution-side dependence processing, the system cannot overlap any of the latencies, because the loads are issued only after the dependent stores resolve and rest of the steps must be performed in order. However, memory-side dependence processing allows the overlap of steps (1) and (2).

The dependence predictor in each DT uses a 1024-entry bit vector. When an aggressively issued load causes a dependence misprediction (and subsequent pipeline flush), the dependence predictor sets a bit to which the load address hashes. Any load whose predictor entry contains a set bit is stalled until all prior stores have completed. Because there is no way to clear individual bit vector entries in this scheme, the hardware clears the dependence predictor after every 10,000 blocks of execution.

1.3.5.5 Load/Store Queues

The hardest challenge in designing a distributed data cache is the memory disambiguation hardware. Because the TRIPS ISA restricts each block to 32 maximum issued loads and stores and eight blocks can be in flight at once, up to 256 memory operations may be in flight. However, the mapping of memory operations to DTs is unknown until their effective addresses are computed. Two resultant problems are: (a) determining how to distribute the LSQ among the DTs, and (b) determining when all earlier stores have completed—across all DTs—so that a held-back load can issue.

Although neither centralizing the LSQ nor distributing the LSQ capacity across the four DTs were feasible options at the time, we solved the LSQ distribution problem largely by brute force. We replicated four copies of a 256-entry LSQ, one at each DT. This solution is wasteful and not scalable (because the maximum occupancy of all LSQs is 25%), but was the least complex alternative for the prototype. The LSQ can accept one load or store per cycle, forwarding data from earlier stores as necessary. Additional details on the DT and LSQ design can be found in [23].

1.3.6 Secondary Memory System

The TRIPS prototype supports a 1 MB static NUCA [14] array, organized into 16 memory tiles (MTs), each one of which holds a four-way, 64 KB bank. Each MT also includes an on-chip network (OCN) router and a single-entry MSHR. Each bank may be configured as an L2 cache bank or as a scratchpad memory, by sending a configuration command across the OCN to a given MT. By aligning the OCN with the DTs, each IT/DT pair has its own private port into the secondary memory system, supporting high bandwidth into the cores for streaming applications. Details of the OCN beyond the description below can be found in [10].

The OCN consists of 16 MTs, each containing an OCN router and a level-2 cache bank. Twenty-four network tiles (NT), each containing an OCN router and the system address translation tables, surround the 2×8 array of MTs. These together form a 4×10, 2D mesh, as shown in the left half of Figure 1.4. Connected to the OCN along the top and the bottom are the I/O tiles, including two DMA controllers, two SDRAM controllers (SDCs), the external bus controller (EBC), and the chip-to-chip network controller (C2C).

The OCN network is *Y-X* dimension-order, wormhole routed; flow control is credit-based, meaning that each node keeps track of the number of empty buffers in all of its neighbors' input FIFOs to determine when it is safe to send more data. Packets travel on one of four virtual channels, designated "Primary Request" (Q1), "Secondary Request" (Q2), "Secondary Reply" (P2), and "Primary Reply" (P1) in order of increasing priority. The packets range in size from 16 bytes to 80 bytes long, split into between one and five 16-byte flits.

OCN clients connect directly to NTs and include ten ports for instruction and data traffic to and from the two on-chip processors, two on-chip DMA controllers, two on-chip SDRAM controllers, one slow external bus controller, and one high-speed chip-to-chip controller. The C2C port is a direct extension of the OCN (albeit at one-eighth the bandwidth per channel), and enables TRIPS chips to be connected gluelessly to one another in a larger system. The OCN can be scaled by either increasing the mesh dimensions (more M and N tiles) or by utilizing the spare client connections on the east side.

1.3.6.1 OCN Router

Figure 1.9 shows an MT along with its embedded OCN router. The OCN router is typical of virtual channel router designs. Incoming packets are

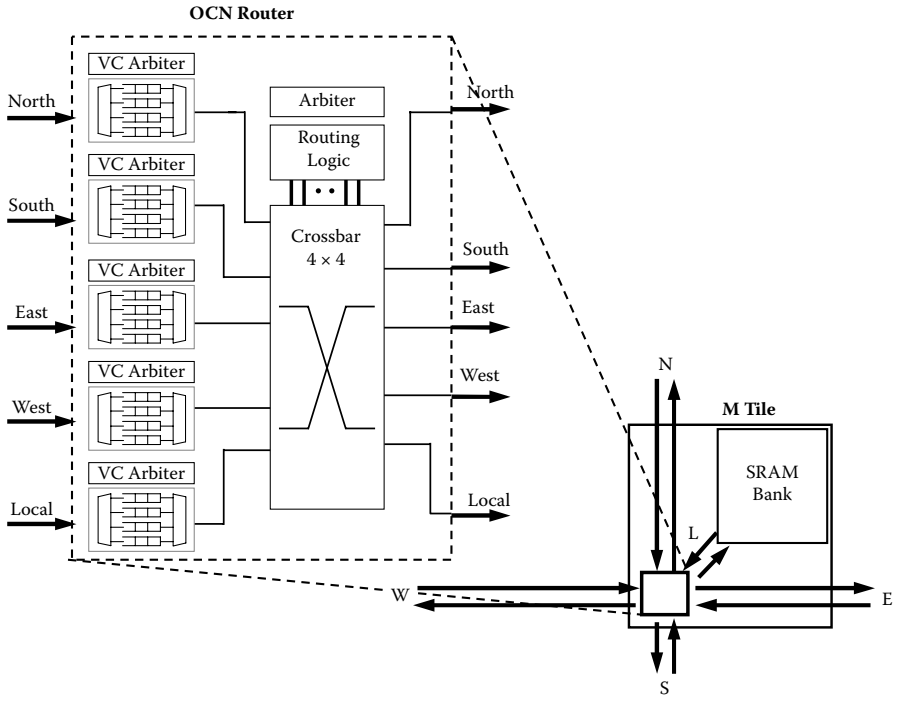


FIGURE 1.9
Memory tile block diagram highlighting OCN router in detail.

latched into one of the input FIFOs in one of five input directions, north, south, east, west, or local for the L2 bank itself. The router contains enough storage for two incoming flits of data per direction, per virtual channel. A 4×4 crossbar network connects each input to every other possible output; a 5×5 crossbar is unnecessary since a packet coming in from one direction cannot depart in that same direction. In cases of contention, the crossbar selects the higher priority channel. The router uses a round-robin arbitration scheme to resolve contention among requests at the same priority level. The direction of the last packet sent in each direction is stored and used on the next arbitration cycle to ensure routing fairness and livelock avoidance. A credit-based flow control scheme tracks the number of available buffers in neighboring receiver FIFOs. When a receiver removes a flit from an incoming FIFO, it sends a credit signal back to the sender to signify more FIFO buffer space is available for future flits.

1.3.6.2 Network Address Translation

An NT forms a gateway to the OCN for clients, such as the TRIPS processors and IO units, to inject packets. Each NT contains an OCN router similar to that discussed in the previous section. The main difference is that the

local interface is connected to an OCN client instead of an L2 cache bank. Virtual-to-system address translation is performed within processors using standard TLBs, but TRIPS supports an additional level of translation to enable reconfiguration of the memory system. An NT translates the system physical address to a network address using a simple table when the OCN client transmits a packet header to an NT. This table consists of 16 entries of eight bits each and is indexed using four bits from the system physical address. Each table entry contains the *X-Y* coordinates of the MT to which the address region is mapped. The table itself is memory mapped and can be modified on-the-fly by the runtime system. By adjusting the mapping functions within the TLBs and the network interface tiles (NTs), a programmer can configure the memory system in a variety of ways including as a single 1 MB shared level-2 cache, as two independent 512 KB level-2 caches (one per processor), as a 1 MB on-chip physical memory (no level-2 cache), or many combinations in between.

1.4 Distributed Microarchitectural Protocols

To enable concurrent, out-of-order execution on this distributed substrate, we implemented traditionally centralized microarchitectural functions, including fetch, execution, flush, and commit, with distributed protocols running across the control and data micronets.

1.4.1 Block Fetch Protocol

The fetch protocol retrieves a block of 128 TRIPS instructions from the ITs and distributes them into the array of ETs and RTs. Figure 1.10 shows the details of the GT's block fetch pipeline which takes a total of 13 cycles, including three cycles for prediction, 1 cycle for TLB and instruction cache tag access, and 1 cycle for hit/miss detection. On a cache hit, the GT sends eight pipelined indices out on the global dispatch network to the ITs. Prediction and instruction cache tag lookup for the next block is overlapped with the fetch commands of the current block. Running at peak, the machine can issue fetch commands every cycle with no bubbles, beginning a new block fetch every eight cycles.

When an IT receives a block dispatch command from the GT, it accesses its I-cache bank based on the index in the GDN message. In each of the next eight cycles the IT sends four instructions on its outgoing GDN paths to its associated row of ETs and RTs. These instructions are written into the read and write queues of the RTs and the reservation stations in the ETs when they arrive at their respective tiles, and are available to execute as soon as they arrive. Because the fetch commands and fetched instructions are delivered in a pipelined fashion across the ITs, ETs, and RTs, the furthest ET receives

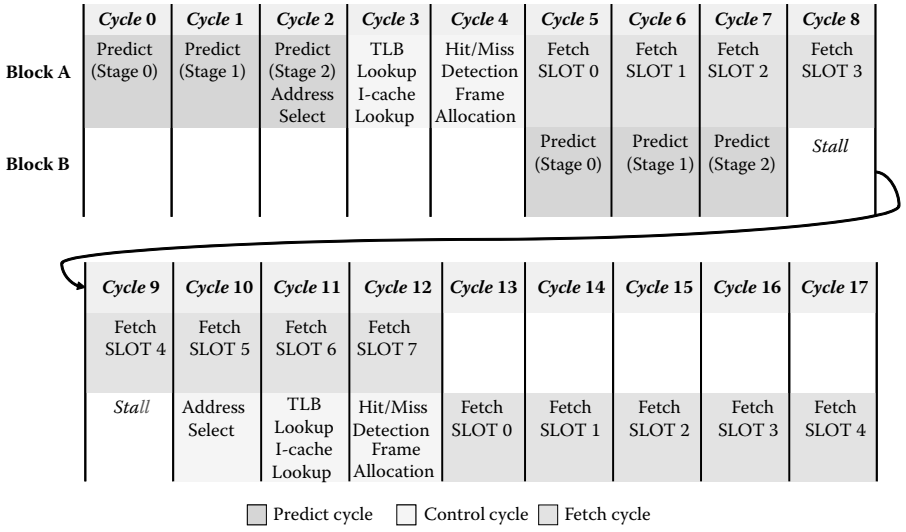


FIGURE 1.10
Fetch pipeline.

its first instruction packet 10 cycles and its last packet 17 cycles after the GT issues the first fetch command. Although the latency appears high, the pipelining enables a high-fetch bandwidth of 16 instructions per cycle in steady state, 1 instruction per ET per cycle.

On an I-cache miss, the GT instigates a distributed I-cache refill, using the global refill network to transmit the refill block's physical address to all of the ITs. Each IT processes the misses for its own chunk independently, and can simultaneously support one outstanding miss for each executing thread (up to four). When the two 64-byte cache lines for an IT's 128-byte block chunk return, and when the IT's south neighbor has finished its fill, the IT signals refill completion northward on the GSN. When the GT receives the refill completion signal from the top IT, it may issue a dispatch for that block to all ITs.

The distributed fetch protocol provides significantly higher fetch bandwidth compared to conventional processors. Directing the fetch from the GT obviates the need for reservation station management at every tile. Because a new set of reservation stations is required for executing every block, managing the free list of reservation stations in a distributed fashion and keeping them synchronized would require additional hardware mechanisms. Instead, by managing the free list in the GT and propagating the allocated identifier along with every fetch, the TRIPS implementation reduces the complexity in other tiles.

The implementation tightly couples the predictor operations and the fetch protocol operations in one single pipeline. In steady state, the three cycles for predict and three cycles for update can fully overlap with the eight cycles

of fetch required for one block. Thus there are no bubbles in the fetch pipeline, enabling a new block fetch every eight cycles. This offers a peak fetch rate of 16 instructions per cycle (128 instruction/8 cycles) matching the peak execution rate of the processor. Occasionally, the predictor update operation may delay the predict operation causing bubbles in the fetch pipeline. For example, in Figure 1.10, an update operation starting in cycle 5 could delay the predict operation for the second block until cycle 8. The fetch of block B will not start until cycle 14, introducing a bubble in the pipeline.

An alternate design could have completely decoupled the prediction pipeline from the fetch pipeline using a fetch target buffer [22]. That design offers two advantages. First, multiple refills can be initiated well ahead of a fetch, offering prefetching benefits. Second, stalls in the predict pipeline are less likely to affect the fetch pipeline. Implementing this design would require additional block management in the fetch unit. Our analysis indicated that the extra complexity of the fetch target buffer was not worth the potential benefits in the current implementation.

1.4.2 Distributed Execution

Dataflow execution of a block begins by the injection of block inputs from the RTs. An RT may begin to process an arriving read instruction even if the entire block has not yet been fetched. Each RT first searches the write queues of all older in-flight blocks. If no matching, in-flight write to that register is found, the RT simply reads that register from the architectural register file and forwards it to the consumers in the block via the OPN. If a matching write is found, the RT takes one of two actions: if the write instruction has received its value, the RT forwards that value to the read instruction's consumers. If the write instruction is still awaiting its value, the RT buffers the read instruction, which will be awakened by a tag broadcast when the pertinent write's value arrives.

Arriving OPN operands wake up instructions within the ET, which selects and executes enabled instructions. The ET uses the target fields of the selected instruction to determine where to send the resulting operand. Arithmetic operands traverse the OPN to other ETs, whereas load and store instructions' addresses and data are sent on the OPN to the DTs. Branch instructions deliver their next block addresses to the GT via the OPN.

An issuing instruction may target its own ET or a remote ET. If it targets its local ET, the dependent instruction can be awakened and executed in the next cycle using a local bypass path to permit back-to-back issue of dependent instructions. If the target is a remote ET, a control packet is formed the cycle before the operation will complete execution, and sent to wake up the dependent instruction early. The OPN is tightly integrated with the wakeup and select logic. When a control packet arrives from the OPN, the targeted instruction is accessed and may be speculatively awakened. The instruction may begin execution in the following cycle as the OPN router injects the arriving operand directly into the ALU. Thus, for each OPN hop between

dependent instructions, there will be one extra cycle before the consuming instruction executes.

Figure 1.11a shows an example of how a code sequence is executed on the RTs, ETs, and DTs. Block execution begins when the read instruction R[0] is issued to RT0, triggering delivery of R4 via the OPN to the left operand of two instructions, `teq` (N[1]) and `muli` (N[2]). When the test instruction receives the register value and the immediate "0" value from the `movi` instruction, it fires and produces a predicate that is routed to the predicate field of N[2]. Because N[2] is predicated on false, if the routed operand has a value of 0, the `muli` will fire, multiply the arriving left operand by four, and send the result to the address field of the `lw` (load word). If the load fires, it sends a request to the pertinent DT, which responds by routing the loaded data to N[33]. The DT uses the load/store IDs (0 for the load and 1 for the store, in this example) to ensure that they execute in the proper program order if they share the same address. The result of the load is fanned out by the `mov` instruction to the address and data fields of the store.

If the predicate's value is 1, N[2] will not inject a result into the OPN, thus suppressing execution of the dependent load. Instead, the `null` instruction fires, targeting the address and data fields of the `sw` (store word). Note that

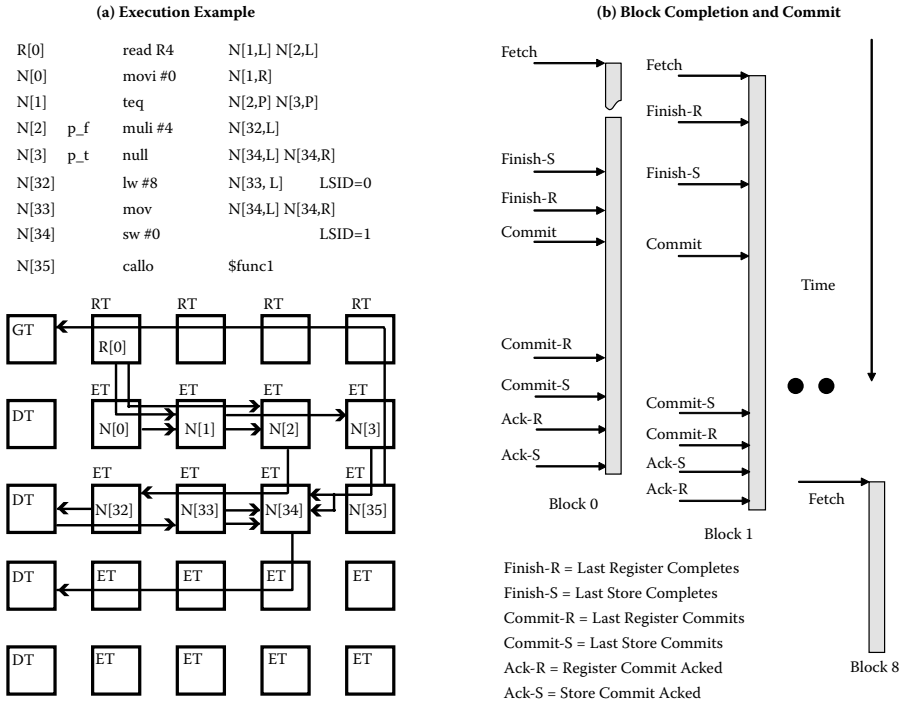


FIGURE 1.11
TRIPS operational protocols.

although two instructions are targeting each operand of the store, only one will fire, due to the predicate. When the store is sent to the pertinent DT and the block-ending call instruction is routed to the GT, the block has produced all of its outputs and is ready to commit. Note that if the store is nullified, it does not affect memory, but simply signals the DT that the store has issued. Nullified register writes and stores are used to ensure that the block always produces the same number of outputs for completion detection.

1.4.3 Block/Pipeline Flush Protocol

Because TRIPS executes blocks speculatively, a branch misprediction or a load/store ordering violation could cause periodic pipeline flushes. These flushes are implemented using a distributed protocol. The GT is first notified when a misspeculation occurs, either by detecting a branch misprediction itself or via a GSN message from a DT indicating a memory-ordering violation. The GT then initiates a flush wave on the GCN which propagates to all of the ETs, DTs, and RTs. The GCN includes a block identifier mask indicating which block or blocks must be flushed. The processor must support multiblock flushing because all speculative blocks after the one that caused the misspeculation must also be flushed. This wave propagates at one hop per cycle across the array. As soon as it issues the flush command on the GCN, the GT may issue a new dispatch command to start a new block. Because both the GCN and GDN have predictable latencies, the instruction fetch/dispatch command can never catch up with or pass the flush command.

1.4.4 Block Commit Protocol

Block commit is the most complex of the microarchitectural protocols in TRIPS, because it involves the three phases illustrated in Figure 1.11b: block completion, block commit, and commit acknowledgment. In phase one, a block is complete when it has produced all of its outputs, the number of which is determined at compile-time and consists of up to 32 register writes, up to 32 stores, and exactly one branch. After the RTs and DTs receive all of the register writes or stores for a given block, they inform the GT using the global status network. When an RT detects that all block writes have arrived, it informs its west neighbor. The RT completion message is daisy-chained westward across the RTs, until it reaches the GT indicating that all of the register writes for that block have been received.

Detecting store completion is more difficult because each DT cannot know a priori how many stores will be sent to it. To enable the DTs to detect store completion, we implemented a DT-specific network called the data status network. Each block header contains a 32-bit *store mask*, which indicates the memory operations (encoded as an LSID bit mask) in the block that are stores. This store mask is sent to all DTs upon block dispatch. When an executed store arrives at a DT, its 5-bit LSID and block ID are sent to the other DTs on the DSN. Each DT then marks that store as received even though it

does not know the store's address or data. Thus, a load at a DT learns when all previous stores have been received across all of the DTs. The nearest DT notifies the GT when all of the expected stores of a block have arrived. When the GT receives the GSN signal from the closest RT and DT, and has received one branch for the block from the OPN, the block is complete. Speculative execution may still be occurring within the block, down paths that will eventually be nullified by predicates, but such execution will not affect any block outputs.

During the second phase (block commit), the GT broadcasts a commit command on the global control network and updates the block predictor. The commit command informs all RTs and DTs that they should commit their register writes and stores to architectural state. To prevent this distributed commit from becoming a bottleneck, we designed the logic to support pipelined commit commands. The GT can legally send a commit command on the GCN for a block when a commit command has been sent for all older in-flight blocks, even if the commit commands for the older blocks are still in flight. The pipelined commits are safe because each tile is guaranteed to receive and process them in order. The commit command on the GCN also flushes any speculative in-flight state in the ETs and DTs for that block.

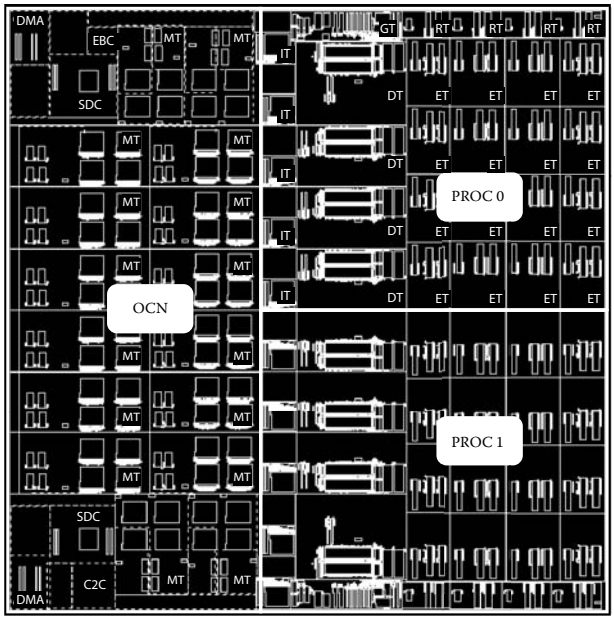
The third phase acknowledges the completion of commit. When an RT or DT has finished committing its architectural state for a given block and has received a commit completion signal from its neighbor on the GSN (similar to block completion detection), it signals commit completion on the GSN. When the GT has received commit completion signals from both the RTs and DTs, it knows that the block is safe to deallocate, because all of the block's outputs have been written to architectural state. When the oldest block has acknowledged commit, the GT initiates a block fetch and dispatch sequence for that block slot.

1.5 Physical Design/Performance Overheads

The physical design and implementation of the TRIPS chip were driven by the principles of partitioning and replication. The chip floorplan directly corresponds to the logical hierarchy of TRIPS tiles connected only by point-to-point, nearest-neighbor networks. The only exceptions to nearest-neighbor communication are the global reset and interrupt signals, which are latency tolerant and pipelined in multiple stages across the chip.

1.5.1 Chip Specifications

The TRIPS chip is implemented in the IBM CU-11 ASIC process, which has a drawn feature size of 130 nm and seven layers of metal. The chip itself includes more than 170 million transistors in a chip area of 18.30 mm by



Overall Chip Area	
29%	Processor 0
29%	Processor 1
21%	Level-2 Cache
14%	On-chip Network
7%	Other (controllers, etc.)

Processor Area	
30%	Functional Units (ALUs)
4%	Register Files and Queues
10%	Level-1 Caches (I and D)
13%	Instruction Queues
13%	Load/Store Queues
12%	Operand Network
2%	Next Block Predictor
16%	Other

FIGURE 1.12
TRIPS physical floorplan and area breakdown.

18.37 mm, which is placed in a 47.5 mm square ball-grid array package. Figure 1.12 shows an annotated floorplan diagram of the TRIPS chip taken directly from the design database, as well as a coarse area breakdown by function. The diagram shows the boundaries of the TRIPS tiles, as well as the placement of register and SRAM arrays within each tile. We did not label the network tiles that surround the OCN because they are so small. Also, for ease of viewing, we have omitted the individual logic cells from this plot. On the right of the diagram is a summary of the fraction of the chip occupied by the major microarchitectural structures.

In addition to the core tiles, TRIPS also includes six controllers that are attached to the rest of the system via the on-chip network (OCN). The two 133/266 MHz DDR SDRAM controllers (SDC) each connect to an individual 1 GB SDRAM DIMM. The chip-to-chip controller extends the on-chip network to a four-port mesh router that gluelessly connects to other TRIPS chips. These links nominally run at one-half the core processor clock and up to 266 MHz. The two direct memory access (DMA) controllers can be programmed to transfer data to and from any two regions of the physical address space including addresses mapped to other TRIPS processors. Finally, the external bus controller (EBC) is the interface to a board-level PowerPC control processor. To reduce design complexity, we chose to offload much of the operating system and runtime control to this PowerPC processor.

TRIPS relies on the trends toward hierarchical design styles with replicated components, but differs from SOCs and CMPs in that the individual

TABLE 1.3

TRIPS Tile Specifications

Tile	Function	Cell Count	Array Bits	Size (mm^2)	Tile Count	% Chip Area
GT	Processor control	52K	93K	3.1	2	1.8
RT	Register file	26K	14K	1.2	8	2.9
IT	Instruction cache	5K	135K	1.0	10	2.9
DT	L1 Data cache	119K	89K	8.8	8	21.0
ET	Instruction execution	84K	13K	2.9	32	28.0
MT	L2 Data cache	60K	542K	6.5	16	30.7
NT	OCN NW interface and routing	23K	—	1.0	24	7.1
SDC	DDR SDRAM controller	64K	6K	5.8	2	3.4
DMA	DMA controller	30K	4K	1.3	2	0.8
EBC	External bus controller	29K	—	1.0	1	0.3
C2C	Chip-to-chip communication controller	48K	—	2.2	1	0.7
Chip Total		5.8M	11.5M	334	106	100.0

tiles are designed to have diverse functions but to cooperate to implement a more powerful and scalable uniprocessor. The entire TRIPS design is composed of only 11 different types of tiles, greatly simplifying both design and verification. Table 1.3 shows additional details of the design of each TRIPS tile. The *Cell Count* column shows the number of placeable instances in each tile, which provides a relative estimate of the complexity of the tile. *Array Bits* indicates the total number of bits found in dense register and SRAM arrays on a per-tile basis, and *Size* shows the area of each type of tile. *Tile Count* shows the total number of tile copies across the entire chip, and % *Chip Area* indicates the fraction of the total chip area occupied by that type of tile.

1.5.2 Chip Verification

The partitioned nature of the TRIPS chip facilitated a highly hierarchical verification strategy. Each of the 11 tile design teams created a sophisticated self-checking testbench for their tile that employed both directed and random tests to exercise as many of the corner cases as possible. The three additional verification models above the tile level were the component level (one each for processor and OCN) and the chip level. The processor verification model executes real TRIPS programs which included simple hand-generated programs, simple compiled programs, automatically generated integer and floating-point instruction test programs, and randomly generated programs. The random program generator obeyed TRIPS block constraints, and also varying instruction distributions, dependence chain length distributions, branch behavior, and predicate chain lengths. The OCN testbench was driven

by regular and random access patterns, typically at rates that far exceed what the OCN would experience in situ. For the full-chip Verilog verification simulations, which only simulate 25 processor cycles per second, we focused primarily on diagnostic tests that read and write all on-chip states from the external bus controller, and tiny single- and multithreaded programs. This hierarchical approach enabled us to eradicate most of the bugs at the tile level; we discovered very few bugs at the component or full-chip level. Those we did discover typically stemmed from incorrect implementation of the micronet interfaces, which were relatively easy to track down and fix.

1.5.3 TRIPS System

A TRIPS system is a multiprocessor constructed from multiple TRIPS chips. Each TRIPS chip is mounted on a daughtercard with two DRAM DIMMs of 1 GB each and voltage regulator circuits. Four daughtercards are connected to a motherboard via high-density connectors. The four TRIPS chips can communicate directly via the chip-to-chip network via the traces on the motherboard. The motherboard also includes a PowerPC 440GP embedded processor chip, a Xilinx FPGA chip, a flash EEPROM chip, and voltage regulators.

The PowerPC chip runs an embedded Linux operating system and serves as both an OS offload processor for the TRIPS prototype chip and as a conduit between the TRIPS chips on a motherboard and a monitor running on a host computer. The monitor connects to the PowerPC via a 100 Mbit ethernet and can access all of the memory and the registers on each of the TRIPS chips for the purposes of program execution and debugging. The FPGA chip connects the C2C ports of two TRIPS chips to connectors at the edge of the board to enable high-speed I/O devices to be connected to the motherboard. Multiple TRIPS boards can be assembled in a single system by connecting the C2C links of adjacent motherboards to each other. The maximum-sized TRIPS system includes 8 boards, 32 TRIPS chips, 64 TRIPS processors, and 64 GB of DRAM. Assuming a processor clock rate of 500 MHz, this system has a peak performance of 545 GFlops.

1.5.4 Area Overheads of Distributed Design

The principal area overheads of the distributed design stem from the wires and logic needed to implement the tile-interconnection control and data networks listed in Table 1.4. The most expensive in terms of area are the two data networks: the operand network (OPN) and the on-chip network. In addition to the 141 physical wires per link, the OPN includes routers and buffering at 25 of the 30 processor tiles. The four-port routers and the eight links per tile consume significant chip area and account for approximately 12% of the total processor area. Strictly speaking, this area is not entirely overhead as it takes the place of the bypass network, which would be much more expensive than the routed OPN for a 16-issue conventional processor.

TABLE 1.4
TRIPS Control and Data Networks

Network	Use	Bits
Global Dispatch (GDN)	I-fetch	205
Global Status (GSN)	Block status	6
Global Control (GCN)	Commit/flush	13
Global Refill (GRN)	I-cache refill	36
Data Status (DSN)	Store completion	72
External Store (ESN)	L1 misses	10
Operand Network (OPN)	Operand routing	141 (×8)
On-chip Network (OCN)	Memory traffic	138 (×8)

The OCN carries a larger area burden with buffering for four virtual channels at each of the four-ported routers. It consumes a total of 14% of the total chip area, which is larger than a bus architecture for a smaller scale memory system, but necessary for the TRIPS NUCA cache. In general, the processor control networks themselves do not have a large area impact beyond the cost of the wires interconnecting the tiles. However, we found that full-chip routing was easily accomplished, even with the large number of wires.

Another large source of area overhead due to partitioning comes from the oversized load/store queues in the DT, accounting for 13% of the processor core area. The LSQ cell count and area are skewed somewhat by the LSQ CAM arrays which had to be implemented from discrete latches, because no suitable dense array structure was available in the ASIC design library.

Across the entire chip, the area overhead associated with the distributed design stem largely from the on-chip data networks. The control protocol overheads are insignificant, with the exception of the load/store queue.

1.5.5 Timing Overheads

The most difficult timing paths we found during logic-level timing optimization were: (1) the local bypass paths from the multicycle floating-point instructions within the ET, (2) control paths for the cache access state machine in the MT, and (3) remote bypass paths across the operand network within the processor core. The operand network paths are the most problematic, because increasing the latency in cycles would have a significant effect on instruction throughput. In retrospect, we underestimated the latency required for the multiple levels of muxing required to implement the operand router, but believe that a customized design could reduce routing latency. These results indicate a need for further research in ultra-low-latency micronetwork routers.

1.5.6 Performance Overheads

We examine the performance overheads of the distributed protocols via a simulation-based study using a cycle-level simulator, called *tsim-proc*, that models the hardware at a much more detailed level than higher-level simulators such as SimpleScalar. A performance validation effort showed that performance results from *tsim-proc* were on average within 4% of those obtained from the RTL-level simulator on our test suite and within 10% on randomly generated test programs. We use the methodology of Fields et al. [9] to attribute percentages of the critical path of the program to different microarchitectural activities and partitioning overheads.

The benchmark suite in this study includes a set of microbenchmarks (dct8x8, sha, matrix, vadd), a set of kernels from a signal processing library (cfar, conv, ct, genalg, pm, qr, svd), a subset of the EEMBC suite (a2time01, bezier02, basefp01, rspeed01, tblook01), and a handful of SPEC benchmarks (mcf, parser, bzip2, twolf, and mgrid). In general, these are small programs or program fragments (no more than a few tens of millions of instructions) because we are limited by the speed of *tsim-proc*. The SPEC benchmarks use the reference input set, and we employ subsets of the program as recommended in [24]. These benchmarks reflect what can be run through our simulation environment, rather than benchmarks selected to leave an unrealistically rosy impression of performance. The TRIPS compiler toolchain takes C or FORTRAN77 code and produces complete TRIPS binaries that will run on the hardware. Although the TRIPS compiler is able to compile major benchmark suites correctly (i.e., EEMBC and SPEC2000) [25], there are many TRIPS-specific optimizations that are pending completion. Until then, performance of compiled code will be lacking because TRIPS blocks will be too small.

Although we report the results of the compiled code, we also employed some hand optimization on the microbenchmarks, kernels, and EEMBC programs. We optimized compiler-generated TRIPS high-level assembly code by hand, feeding the result back into the compiler to assign instructions to ALUs and produce an optimized binary. Where possible, we report the results of the TRIPS compiler and the hand-optimized code. We have not optimized any of the SPEC programs by hand and are working to improve compiler code quality to approach that of hand-optimized.

1.5.6.1 Distributed Protocol Overheads

To measure the contributions of the different microarchitectural protocols, we computed the critical path of the program and attributed each cycle to one of a number of categories. These categories include instruction distribution delays, operand network latency (including both hops and contention), execution overhead of instructions to fan operands out to multiple target instructions, ALU contention, time spent waiting for the global control tile (GT) to be notified that all of the block outputs (branches, registers, stores) have been produced, and the latency for the block commit protocol to complete. Table 1.5 shows the overheads as a percentage of the critical path of the

TABLE 1.5

Network Overheads and Preliminary Performance of Prototype

	Distributed Network Overheads as a Percentage of Program Critical Path							Preliminary Performance				
								Speedup		IPC	IPC	IPC
Benchmark	IFetch	OPN Hops	OPN Cont.	Fanout Ops	Block Complete	Block Commit	Other	TCC	Hand	Alpha	TCC	Hand
dct8x8	5.39	30.57	10.04	3.76	3.24	2.11	44.89	2.25	2.73	1.69	5.13	4.78
matrix	7.99	20.25	17.24	4.89	4.10	3.17	42.36	1.07	3.36	1.68	2.85	4.12
sha	0.57	17.91	6.29	11.73	0.10	0.66	62.74	0.40	0.91	2.28	1.16	2.10
vadd	7.41	17.66	13.79	5.61	5.99	7.48	42.06	1.46	1.93	3.03	4.62	6.51
cfar	3.75	32.06	9.39	9.78	2.44	0.99	41.59	0.66	0.81	1.53	1.35	1.98
conv	4.10	34.29	16.16	2.71	2.49	2.48	37.77	1.48	2.48	2.08	4.27	5.94
ct	6.23	18.81	16.25	6.04	3.65	3.79	45.23	1.29	3.84	2.31	4.22	5.25
genalg	3.85	18.60	5.76	8.82	2.21	0.62	60.14	0.51	1.46	1.05	1.10	1.65
pm	2.89	25.86	6.21	3.86	1.86	1.03	58.29	0.57	0.99	1.19	1.41	1.96
qr	4.53	22.25	8.85	11.97	2.72	2.23	47.45	0.47	0.98	1.30	1.94	2.36
svd	5.13	15.77	3.84	4.59	3.15	1.46	66.06	0.29	0.68	1.02	1.25	1.11
a2time01	4.94	13.57	6.47	9.52	2.05	4.02	59.43	1.21	4.38	0.95	1.50	4.11
bezier02	2.59	16.92	5.22	12.54	0.21	2.63	59.89	1.61	3.30	1.05	1.91	3.20
basefp01	3.36	13.63	5.44	6.34	2.74	2.90	65.59	1.35	8.02	0.78	1.03	3.55
rspeed01	0.76	28.67	10.61	11.77	0.39	0.14	47.66	1.26	4.18	1.03	1.82	3.38
tblook01	2.88	28.83	9.38	5.68	1.73	0.69	50.81	0.18	0.61	1.44	0.77	1.46
181.mcf	1.64	28.52	6.10	0.00	0.08	0.18	63.48	0.51	—	0.54	0.78	—
197.parser	2.96	30.76	3.99	0.84	0.30	0.66	60.49	0.60	—	1.18	1.10	—
256.bzip2	1.97	33.87	15.17	0.18	0.01	0.18	48.62	0.36	—	1.40	1.27	—
300.twolf	3.01	18.05	3.08	0.75	0.25	0.84	74.02	0.65	—	1.00	1.24	—
172.mgrid	5.06	18.61	25.46	4.03	3.00	2.78	41.06	1.49	—	1.33	4.89	—

program, and the column labeled "Other" includes components of the critical path also found in conventional monolithic cores including ALU execution time, and instruction and data cache misses.

The largest overhead contributor to the critical path is the operand routing, with hop latencies accounting for up to 34% and contention accounting for up to 25%. These overheads are a necessary evil for architectures with distributed execution units, although they can be mitigated through better scheduling to minimize the distance between producers and consumers along the critical path and by increasing the bandwidth of the operand network. For some of the benchmarks, the overheads of replicating and fanning out operand values can be as much as 12%. Most of the rest of the distributed protocol overheads are small, typically summing to less than 10% of the critical path. These results suggest that the overheads of the control networks are largely overlapped with useful instruction execution, but that the data networks could benefit from further optimization.

1.5.6.2 Total Performance

To understand the impact of the distributed protocols on overall performance, we compared execution time on *tsim-proc* to that of a more conventional, albeit clustered, uniprocessor. Our baseline comparison point was a 467 MHz Alpha 21264 processor, with all programs compiled using the native Gem compiler with the "-O4 -arch ev6" flags set. We chose the Alpha because it has an aggressive ILP core that still supports low FO4 clock periods, an ISA that lends itself to efficient execution, and a truly amazing compiler that generates extraordinarily high-quality code. We use Sim-Alpha, a simulator validated against the Alpha hardware to take the baseline measurements so that we could normalize the level-2 cache and memory system and allow better comparison of the processor and primary caches between TRIPS and Alpha.

Table 1.5 shows the performance of the TRIPS processor compared to the Alpha. Because our focus is on the disparity between the processor cores, we simulated a perfect level-2 cache with both processors, to eliminate differences in performance due to the secondary memory system. The first column shows the speedup of TRIPS compiled code (TCC) over the Alpha. We computed speedup by comparing the number of cycles needed to run each program. The second column shows the speedup of the hand-generated TRIPS code over that of Alpha. Columns 3–5 show the instruction throughput (instructions per clock or IPC) of the three configurations. The ratios of these IPCs do not correlate directly to performance, because the instruction sets differ, but they give an approximate depiction of how much concurrency the machine is exploiting. Our results show that on the hand-optimized codes, TRIPS executes between 0.6 and 1.8 times as many instructions as Alpha, largely due to fanout instructions and single-to-double conversions required by TRIPS for codes that use 32-bit `floats`. The code bloat is currently larger for compiled code, up to four times as many instructions in the worst case.

Although these results are far from the best we expect to obtain, they do provide insight into the capabilities of TRIPS. The results show that for the hand-optimized programs, the TRIPS distributed microarchitecture is able to sustain reasonable instruction-level concurrency, ranging from 1.1 to 6.5. The speedups over the Alpha core range from 0.6 to just over 8. **sha** sees a slowdown on TRIPS because it is an almost entirely serial benchmark. What little concurrency there is is already mined out by the Alpha core, so the TRIPS processor sees a slight degradation because of the block overheads, such as interblock register forwarding. Convolution (**conv**) and **vadd** have speedups close to two because the TRIPS core has exactly double the L1 memory bandwidth as Alpha (four ports as opposed to two), resulting in an upper-bound speedup of two. Compiled TRIPS code does not fare as well, but does exceed the performance of Alpha on about half of the benchmarks. The maturation time of a compiler for a new processor is not short, but we anticipate significant improvements as our hyperblock generation and optimization algorithms come online.

We conclude from this analysis that the TRIPS microarchitecture can sustain good instruction-level concurrency—despite all of the distributed overheads—given kernels with sufficient concurrency and aggressive hand-coding. Whether the core can exploit ILP on full benchmarks, or whether the compiler can generate sufficiently optimized code, remain open questions that are subjects of our current work.

1.6 Related Work

Much of the TRIPS architecture is inspired by important prior work across many computer architecture domains, including tiled architectures, dataflow architectures, superscalar processors, and VLIW architectures.

1.6.1 Tiled Architectures

With transistor counts approaching one billion, tiled architectures are emerging as an approach to manage design complexity. The RAW architecture [30] pioneered research into many of the issues facing tiled architectures, including scalar operand networks, a subset of the class of micronetworks designed for operand transport [29]. Another more recent tiled architecture that, as does RAW, uses homogeneous tiles is Smart Memories [19]. Emerging fine-grained CMP architectures, such as Sun's Niagara [15] or IBM's Cell [21], can also be viewed as tiled architectures. All of these architectures implement one or more complete processors per tile. In general, these other tiled architectures are interconnected at the memory interfaces, although RAW allows register-based interprocessor communication. TRIPS differs in three ways: (1) tiles are heterogeneous, (2) different types of tiles are composed to create a uniprocessor, and (3) TRIPS uses distributed control network protocols to

implement functions that would otherwise be centralized in a conventional architecture.

1.6.2 Dataflow Architectures

The work most similar to TRIPS are the two recent dataflow-like architectures that support imperative programming languages such as C. These architectures, developed concurrently with TRIPS, are WaveScalar [28] and ASH [2]. WaveScalar breaks programs into blocks (or "waves") similar to TRIPS, but differs in the execution model because all control paths are mapped and executed, instead of the one speculated control path of TRIPS. Other major differences include dynamic, rather than static placement of instructions, no load speculation, and more hierarchy in the networks, because WaveScalar provides many more execution units than TRIPS. ASH uses a similar predication model and dataflow concepts, but targets application-specific hardware for small programs, as opposed to compiling large programs into a sequence of configurations on a programmable substrate such as TRIPS. The behavior inside a single TRIPS block builds on the rich history of dataflow architectures including work by Dennis [8], Arvind [1], and hybrid dataflow architectures such as the work of Culler [7] and Iannucci [12].

1.6.3 Superscalar Architectures

The TRIPS microarchitecture incorporates many of the high-ILP techniques developed for aggressive superscalar architectures, such as two-level branch prediction and dependence prediction. The TRIPS block atomic execution model is descended from the Block-Structured ISA proposed by Patt et al. to increase the fetch rate for wide issue machines [11]. Other current research efforts also aim to exploit large-window parallelism by means of checkpointing and speculation [6,27].

1.6.4 VLIW Architectures

TRIPS shares some similarities to VLIW architectures in that the TRIPS compiler decides where (but not when) instructions execute. Although the TRIPS compiler does not have to decide instruction timing—unlike VLIW architectures—the VLIW compilation algorithms for forming large scheduling regions, such as predicated hyperblocks [18], are also effective techniques for creating large TRIPS blocks.

1.7 Conclusions

When the first TRIPS paper appeared in 2001 [20], the high-level results seemed promising, but it was unclear (even to us) whether this technology was implementable in practice, or whether it would deliver the performance

indicated by the high-level study. The microarchitecture described in this chapter is an existence proof that the design challenges unanswered in 2001 were solvable; the distributed protocols we designed to implement the basic microarchitecture functions of instruction fetch, operand delivery, and commit are feasible and do not incur prohibitive overheads. The distributed control overheads are largely overlapped with instruction execution, the logic required to implement the protocols is not significant, and the pipelined protocols are not on critical timing paths.

The data networks, however, carry a larger area and performance burden because they are on the critical paths between data-dependent instructions. In the prototype, we are working to reduce these overheads through better scheduling to reduce hop-counts; architectural extensions to TRIPS may include more operand network bandwidth. The original work assumed an ideal, centralized load/store queue, assuming that it could be partitioned in the final design. Because partitioning turned out to be unworkable, we elected to put multiple full-sized copies in every DT, which combined with an area-hungry standard-cell CAM implementation, caused our LSQs to occupy 40% of the DTs. Solving the problem of area-efficiently partitioning LSQs has been a focus of our research for the past year.

These distributed protocols have enabled us to construct a 16-wide, 1024-instruction window, out-of-order processor, which works quite well on a small set of regular, hand-optimized kernels. We have not yet demonstrated that code can be compiled efficiently for this architecture, or that the processor will be competitive even with high-quality code on real applications. Despite having completed the prototype, much work remains in the areas of performance tuning and compilation before we will understand where the microarchitectural, ISA, and compiler bottlenecks are in the design. Once systems are up and running in the fall of 2006, we will commence a detailed evaluation of the capabilities of the TRIPS design to understand the strengths and weaknesses of the system and the technology.

Looking forward, partitioned processors composed of interconnected tiles provide the opportunity to dynamically adjust their granularity. For example, one could subdivide the tiles of a processor to create multiple smaller processors, should the balance between instruction-level and thread-level parallelism change. We expect that such substrates of heterogeneous or homogeneous tiles will provide flexible computing platforms that can be tailored at runtime to match the concurrency needs of different applications.

Acknowledgments

We thank our design partners at IBM Microelectronics, and our colleagues at Synopsys for their generous university program. This research was supported financially by the Defense Advanced Research Projects Agency under contracts F33615-01-C-1892 and NBCH30390004, NSF instrumentation grant

EIA-9985991, NSF CAREER grants CCR-9985109 and CCR-9984336, IBM University Partnership awards, and grants from the Alfred P. Sloan Foundation and the Intel Research Council.

References

[1] Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[2] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 14–26, October 2004.

[3] D. Burger, S. Keckler, K. McKinley, M. Dahlin, L. John, C. Lin, C. Moore, J. Burrill, R. McDonald, and W. Yoder. Scaling to the end of silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[4] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the International Symposium on Computer Architecture*, pp. 142–153, June, 1998.

[5] K. Coons, X. Chen, S. Kushwaha, K. McKinley, and D. Burger. A spatial path scheduling algorithm for edge architectures. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 129–140, October 2006.

[6] A. Cristal, O. J. Santana, M. Valero, and J. F. Martinez. Toward kilo-instruction processors. *ACM Transactions on Architecture and Code Optimization*, 1(4):389–417, December 2004.

[7] D. E. Culler, A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 164–175, April 1991.

[8] J. Dennis and D. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the International Symposium on Computer Architecture*, pp. 126–132, January 1975.

[9] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the International Symposium on Computer Architecture*, pp. 74–85, July 2001.

[10] P. Gratz, C. Kim, R. McDonald, S. W. Keckler, and D. Burger. Implementation and evaluation of on-chip network architectures. In *Proceedings of the International Conference on Computer Design*, October 2006.

[11] E. Hao, P. Chang, M. Evers, and Y. Patt. Increasing the instruction fetch rate via block-structured instruction set architectures. In *Proceedings of the International Symposium on Microarchitecture*, pp. 191–200, December 1996.

[12] R. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the International Symposium on Computer Architecture*, pp. 131–140, May 1988.

[13] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[14] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 211–222, October 2002.

[15] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[16] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings International Symposium on Computer Architecture*, pp. 81–87, Los Alamitos, CA, 1981. IEEE Computer Society Press.

[17] B. A. Maher, A. Smith, D. Burger, and K. S. McKinley. Head and tail duplication for convergent hyperblock formation. In *Proceedings of the International Symposium on Microarchitecture*, December 2006.

[18] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the International Symposium on Microarchitecture*, pp. 45–54, June 1992.

[19] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceedings of the International Symposium on Computer Architecture*, pp. 161–171, June 2000.

[20] R. Nagarajan, K. Sankaralingam, D. Burger, and S. W. Keckler. A design space evaluation of grid processor architectures. In *Proceedings of the International Symposium on Microarchitecture*, pp. 40–51, December 2001.

[21] D. Pham, S. Asano, M. Bolliger, M. Day, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor. In *Proceedings of the International Solid-State Circuits Conference*, pp. 184–185, February 2005.

[22] G. Reinman, T. M. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *Proceedings of the International Symposium on Computer Architecture*, pp. 234–245, May 1999.

[23] S. Sethumadhavan, R. McDonald, R. Desikan, D. Burger, and S. W. Keckler. Design and implementation of the TRIPS primary memory system. In *Proceedings of the International Conference on Computer Design*, October 2006.

[24] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Technique*, pp. 3–14, September 2001.

[25] A. Smith, J. Burrill, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, and K. S. McKinley. Compiling for EDGE architectures. In *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 185–195, March 2006.

[26] A. Smith, R. Nagarajan, K. Sankaralingam, R. McDonald, D. Burger, S. W. Keckler, and K. S. McKinley. Efficient dataflow predication. In *Proceedings of the International Symposium on Microarchitecture*, December 2006.

[27] S. Srinivasan, R. Rajwar, H. Akkary, A. Ghandi, and M. Upson. Continual flow pipelines. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 107–119, October 2004.

[28] S. Swanson, K. Michaelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proceedings of the International Symposium on Microarchitecture*, pp. 291–302, December 2003.

[29] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pp. 341–353, February 2003.

[30] E. Waingold, M. Taylor, D. 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*, 30(9):86–93, September 1997.

2

High-Performance Data Security in an x86 Processor

G. Glenn Henry, Terry Parks, and Tom Crispin

Centaur Technology Inc.

CONTENTS

2.1 Introduction

2.1.1 Background

Data security is a major concern in personal computing (PC) systems due to the ubiquitous connectivity of PCs to the Internet and the many well-publicized cases of viruses, data theft, and so forth. The increasing use of wireless connectivity further increases security risks. As designers (Centaur Technology Inc.) of the VIA Technologies Inc. (VIA) family of Intel®-compatible (x86) processors, we decided in 2001 to integrate the fundamental building blocks of data security into all of our future x86 processors. By integrating these functions, we could exploit the inherently high performance of the processor, as well as ensure that add-on hardware adapters, additional chips, and the like were not needed to provide data security: fast data security features would always be available to software. Because no other x86 processor manufacturer (Intel and AMD®) provided these security functions in their processors, we felt that we should set the example by providing such functionality in all of our processors, for free. Our challenge was thus twofold: to design secure and high-performance security features, and to do it with little development effort or die-size impact. This chapter summarizes the security features we created, and how we did it.

The integrated hardware functions we provide are fundamental building blocks of good data security: hardware random number generation,

symmetric-key encryption, public-key encryption performance assistance (modular multiplication), and secure hash generation. Our hardware implementation provides significant performance improvements over software-only implementation, as well as increasing the integrity of the security functions. Unique hardware features allow multiple applications to use these security functions directly, without any intervening operating system software. In this chapter, the hardware random number generator and the symmetric key encryption are covered in more detail than the other two functions. This reflects the design complexity as well as the importance from a security aspect.

The functions described here started to ship in VIA processors in 2003. The implementation described is the VIA C7™ processor, a low-cost, low-power, x86 processor manufactured in 90-nm silicon-on-insulator (SOI) technology, and shipping since mid-2005. Note that several patents cover the design and implementation described here. More information about the VIA security initiative can be found at [VIA06].

The primary Centaur Technology personnel that developed these security features were the authors and Tim Elliott, Jim Lundberg, and Brian Snider. Thanks also go to Phil Zimmermann for his useful suggestions during the formative stages of this project.

2.1.2 Key Design Precepts

The following basic objectives or precepts guided our implementation of security functions within the processor.

Provide very robust security. Whatever features we provide must meet very high security standards. Where applicable, we implemented the U.S. government standard because these standards are subject to extensive public review and analysis. (And, of course, they are used a lot.) An example of the impact of this objective is the extensive design and validation effort made to ensure that our random number generator produced cryptographically good random bits. Another example is the encryption unit, which was designed to be secure against timing and power attacks. In addition, the basic objective of requiring no operating system software (discussed subsequently) was primarily driven by concerns for the integrity of the security features.

Provide significant performance improvements over software implementation. Our goal here was to be faster by orders of magnitude, not merely by a few percentage points. This level of performance required major hardware components to be added to the chip; a few additional gates in existing components could not provide the desired performance.

Require no software outside of the application. The reason for this unobvious requirement is part philosophical and part practical.

The philosophical reason is that eliminating OS software and device drivers from the critical security paths improves security. The OS and its drivers are subject to viruses and other corrupting agents; hardware is not. Also, an application doesn't know what is really being done in OS software: Is there a back door? Is the data being recorded? Can the data leak out somehow? Thus, we felt that highly secure applications would prefer a direct interface to the hardware, assuming, of course, that the hardware interface was fast, secure, and easy to use. The practical reason is the difficulty in getting OS support from Microsoft for new instructions such as these security enhancements. (It turns out, however, that the open software community aggressively supported our security functions in the form of kernel support as well as library support.)

The implication of the requirement is that applications must be able to use the hardware security functions using nonprivileged x86 instructions. Thus, we had to add new x86 instructions to the existing instruction set. A further implication is that the processor must provide multitasking support for these features; we assumed no OS support in saving new instruction states, and the like. Thus, in addition to the security hardware itself, the instruction set design and associated microcode are important components.

Provide for free, on all of our processors. Because there was no quantified market opportunity for hardware security, any die cost increase associated with the functions needed to be insignificant; no one would pay extra for these features, nor did we want anyone to pay extra. The impact on other development activities also needed to be insignificant because we are a very small development team and have little "extra" manpower to apply to such projects.

The die cost impact of the security features was minimized by our normal custom design approach and by the fact that there is usually some unused space on a custom design such as ours. The development cost impact was reduced by extensive use of existing design elements and by volunteer work: that is, using designer's time that wouldn't normally be spent on Centaur activities.

2.1.3 Security Features Provided

2.1.3.1 Hardware Random Number Generator (RNG)

Random numbers are used in many computer applications such as simulation and statistical analysis. Random numbers, however, are essential in all forms of data security including generating keys for symmetric-key and public-key encryption, digital signatures, secure data hashes, seeds for secure passwords, challenges in authentication protocols, and so forth. Ultimately, using modern security algorithms, security is only as good as the randomness of the underlying "random" numbers. Unfortunately, random numbers

good enough for security applications are very difficult to generate using deterministic mechanisms such as computer programs. Accordingly, there is a large body of research, analysis, and literature (which we do not cover here) on the basic topics of, "What is randomness, how can we generate it, and how can we test or verify it?"

There are three classes of random number generators based on the characteristics of the values they produce. *Truly random* numbers must (1) have unbiased statistical properties, (2) be unpredictable, and (3) be unrepeatable [Sch96]. The last criterion makes it technically impossible to generate truly random numbers using only software on a computer. *Pseudorandom numbers* (numbers that statistically appear to be random, but don't meet the second and third criteria), however, can be generated using software algorithms. The commonly available generators—such as those available in high-level language libraries—can have good statistical properties, but are very predictable as well as repeatable. *Cryptographically secure pseudorandom numbers* are, for practical purposes, unpredictable, a critical requirement for random numbers used in security applications. The unpredictability is usually obtained by combining data obtained from real-world actions such as keystroke timing with some other pseudorandom generator. The real-world data keeps the result from being predictable. These generators are usually fairly slow, however. An example of a cryptographically secure pseudorandom generator—along with some good discussion of basic concepts—is found in [Kel00]. These types of generators are also called deterministic random bit generators (DRBG) in National Institute of Standards and Technology (NIST) literature. [Bar06] contains the NIST recommendations for DRBGs.

Even cryptographically secure pseudorandom number generation algorithms, however, do not generate truly random numbers because they are repeatable. Thus, only hardware mechanisms can generate truly random numbers. Although a few add-on hardware RNG devices exist for PCs, these add-ons are slow, very expensive, and generally unavailable. An example of such a hardware add-on is [Qua06]. The price for the PCI card version is over $2000 (in 2006). Before our implementation there was no cost-effective or easy-to-use source for truly random numbers on modern PCs.

Given these factors, we decided to implement a robust hardware-based RNG with extensive focus on the "quality" (or randomness) of the generated sequences. Our approach utilized a unique design approach for a hardware RNG that is small, fast, and scales with improving technology. Part of the design effort (in fact, the largest part) was extensive mathematical analysis to characterize the randomness of the generated bits.

2.1.3.2 Advanced Encryption Standard (AES) Encryption Hardware

After a good RNG, the next most important data-security function is *symmetric-key* (or block) encryption. This type of encryption can provide both high security and high performance. It is used for data encryption where the data is more than a few hundred bytes, or the computational

capabilities available for encryption are low. AES is the new U.S. government symmetric-key encryption standard, becoming an official NIST standard in early 2002 [NI01a]. AES is more secure than the old Data Encryption Standard (DES), which it replaced. In addition, AES was adopted after an open competition with worldwide participation as opposed to the oft-maligned secret selection of DES. The winning cipher was Rijndael, invented by Belgian cryptographers [Dae02]. AES has received worldwide acceptance and is rapidly becoming the dominant block cipher in use.

The AES standard operates on a 128-bit block of data (with key sizes of 128, 192, and 256 bits) to produce a corresponding 128-bit block. This one-to-one encryption approach, or *operating mode*, is called Electronic Code Book (ECB). In practice, however, real-world encryption rarely uses ECB; there are other more secure operating modes that all have the property of mixing something with the input or output data such that the same data input block never produces the same output block. These more secure operating modes are defined and recommended by another NIST publication [Dwo01].

VIA processors implement the entire AES standard in hardware. The performance improvement of this hardware approach is one to two orders of magnitude faster than the same operation done in x86 software depending on cachability. In addition to the basic ECB mode, Centaur processors directly support in hardware four additional common operating modes. The modes are cipher block chaining (CBC), cipher feedback (CFB), output feedback (OFB), and counter mode (CTR).

2.1.3.3 Secure Hash Algorithm (SHA) Hardware

A secure hash is a short digest, or summary, of a longer message such that "It is computationally infeasible, (1) to find a message that corresponds to a given message digest, or (2) to find two different messages that produce the same message digest." [NI02]. The intent of the hash is to guarantee a message's integrity. If the message is changed somehow, then the hash of the new message will not match the original hash. This integrity guarantee is essential to many cryptographic protocols including digital signatures.

The Federal Information Processing Standards (FIPS) standard defines four variants of the same basic approach, which is to irreversibly mix incoming data blocks in a complex fashion with the accumulating hash. The standard variants are called SHA-1, SHA-256, SHA-384, and SHA-512 [NI02]. The variants differ in the size of the data block to be hashed and the size of the resultant hash. The basic algorithmic approach is the same for all SHA varieties although there are some detailed differences.

VIA processors implement the SHA-1 and SHA-256 algorithms in hardware. The performance improvement over software is about one order of magnitude.

2.1.3.4 Montgomery Multiplier Hardware

AES is a symmetric-key algorithm. The other basic type of encryption is *public-key* encryption. This has significant advantages over symmetric-key

encryption in that no secret keys need to be distributed. Public-key algorithms are, however, computationally very expensive: several hundred (or thousands) of times slower than a good symmetric-key algorithm. For that reason, they are not used for large blocks of data. Public-key encryption is heavily used, however, for short messages such as distributing secure keys and digital signatures over the Internet. The dominant public-key algorithm today, especially for Internet applications, is called RSA (named after its inventors: Ron Rivest, Adi Shamir, and Leonard Adelman) [Riv78]. RSA is also defined as a part of the FIPS Digital Signature Standard [NI00].

The fundamental computational element of RSA is modular multiplication:

$$C = M^e \bmod n$$

where the M, e, and n values are thousands of bits long. This calculation (done by repeated calculations of $M \times M \bmod n$) is very slow, even on modern processors. A well-known performance improvement that can speed up this calculation is the Montgomery Multiply algorithm [Mon85]. In this approach, all numbers are transformed into a different mathematical field such that the modular multiplication in that field does not require a division operation. Using software, this special modular multiplication operation is slightly slower than a normal multiplication, but significantly faster than multiplying and doing the modulus operation (division). At the end of all of the Montgomery multiplies, the result must be transformed back into our normal number field to yield the true result. The field transformations are slow, but they only have to be done twice for each RSA calculation.

We have directly implemented the Montgomery Multiply function as a special hardware multiply instruction. This hardware implementation runs several times faster than software using the normal x86 multiply instructions. Unlike AES where doing the entire algorithm in hardware can speed up performance by orders of magnitude, doing the total RSA algorithm in hardware would offer no significant performance improvement over just implementing the Montgomery Multiply function.

2.1.4 x86 Instruction Set

2.1.4.1 Instruction Structure

As described above, our security functions are designed for direct use by an application. Two x86 primary opcodes provide access to all VIA security functions. These opcodes are unused and cause invalid opcode exceptions in other x86 processors (Intel and AMD). On VIA processors, these opcodes can be switched back to being invalid by the operating system or BIOS using a machine specific register (MSR). An extension to the standard x86 CPUID instruction function reports information about the existence of the security functions. Each primary opcode (called a "block" opcode in x86 terminology) provides eight more specific "subopcodes," which correspond directly

TABLE 2.1

New x86 Opcodes for security

Opcode	Function
0x0F 0xA7 0xC0	Store available random bytes
0xF3 0x0F 0xA7 0xC0	REP store random bytes
0xF3 0x0F 0xA7 0xC8	REP AES ECB encrypt/decrypt
0xF3 0x0F 0xA7 0xD0	REP AES CBC encrypt/decrypt
0xF3 0x0F 0xA7 0xD8	REP AES CTR encrypt/decrypt
0xF3 0x0F 0xA7 0xE0	REP AES CFB encrypt/decrypt
0xF3 0x0F 0xA7 0xE8	REP AES OFB encrypt/decrypt
0xF3 0x0F 0xA6 0xC0	REP Montgomery Multiply
0xF3 0x0F 0xA6 0xC8	REP SHA-1
0xF3 0x0F 0xA6 0xD0	REP SHA-256

to major functions such as read random bits and encrypt a block. Table 2.1 lists the currently implemented opcodes and their functions.

Because the instructions are optimized for multiblock processing (see next section), the instruction operands are mostly pointers to memory locations. For example, for AES-CBC encryption, the operands are (using standard x86 notation):

- ES:[EAX] = pointer to initialization vector
- ES:[EDX] = pointer to control word
- ES:[EBX] = pointer to encryption key
- ES:[ESI] = pointer to plaintext input
- ES:[EDI] = pointer to ciphertext output
- ECX = block count

2.1.4.2 Instruction Functions

The x86 instructions do not directly access the security hardware; a layer of microcode resides between the instruction and the hardware in order to provide the following important instruction functions.

Optimizing Multiblock Operations. The typical use of the security functions (other than the RNG) is to perform the same operation (such as encryption) sequentially on many sequential blocks (e.g., encrypting an entire file). The VIA security instructions handle multiblock sequences in a manner consistent with other x86 string operations that use the x86 REP prefix. This prefix effectively directs the processor to perform the same instruction operation repetitively for a given count. Implementing this REP function in microcode provides

a significant performance advantage over having the program issue a non-REP version multiple times in a loop. Microcode can overlap loads, stores, addressing updates, and string-completion branches with the actual hardware operation. Microcode can also interrupt the REP operation while ensuring that the original REP instruction can be restarted and continue properly after the interrupt (this requirement is part of the standard x86 architecture).

Transparent AES Multitasking. All functions other than AES encryption and decryption use no additional state other than that in memory and the standard x86 registers. Thus, no unique operating system support is required to share the security functions among multiple software tasks. As described subsequently in more detail in the AES details section, AES is different. Optimal performance requires a large amount of "hidden" state (the expanded AES key) not present in normal x86 architecture. Such a state would normally require the operating system to save and reload the state across context switches. Instead, the AES microcode is able to re-create this hidden state automatically when needed after a software context switch.

Further details of the VIA x86 security instructions are found in [VIA05].

2.1.5 Performance Considerations

Because increased performance was one of the major objectives of our hardware implementation approach, we describe performance in the more detailed sections of this chapter. For AES, SHA, and Montgomery Multiply, we present two views of performance. The "datasheet number" view is that typically used in the processor industry to describe the performance of instructions; that is, we quote the number of clocks it takes to do some basic processor function (such as encrypt a block) assuming all the data in the level-one cache, nothing else is happening in the processor, no I/O is running, and so on.

The second type of performance we report uses a real system with real application software operating on real data files: it uses real memory and cache characteristics. In addition, this approach can easily provide comparative numbers to equivalent software implementations on our and other processors. To obtain this comparative data, we use well-known publicly available source code libraries that perform the function in software. These libraries are either modified directly, or an equivalent version is made that uses the VIA security instructions.

2.1.6 Physical Design Methodology

The subsequent description of our implementation assumes some familiarity with the general physical design of our processor. Figure 2.1 is a picture of the referenced VIA C7 processor. The die size is approximately 31 mm^2.

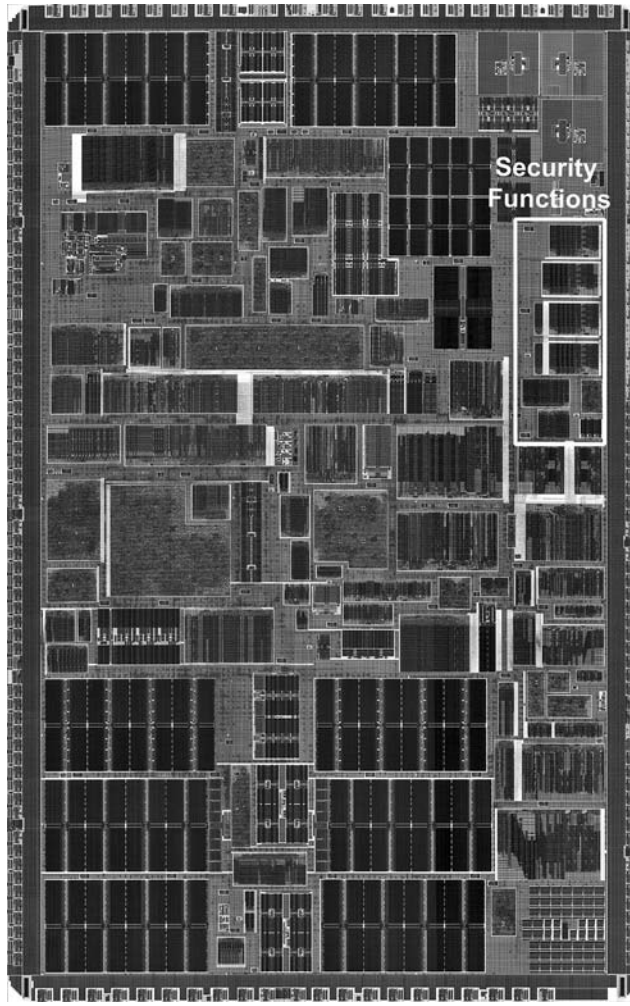


FIGURE 2.1
VIA C7 processor.

The security components discussed in the chapter are outlined near the upper right edge. The size of the components themselves totals about 0.5 mm^2. Figure 2.2 shows a close-up of the security components. In these and other pictures of chip components in this chapter, almost all metal routing has been removed.

The total chip contains about 400 discrete "top-level" components that are each built independently as a physical block. To form the chip, these top-level components are connected together with global signal routing as well as power distribution, global clock distribution, RC repeaters, bulk capacitor fill, and others. The three basic types of top-level components are as follows.

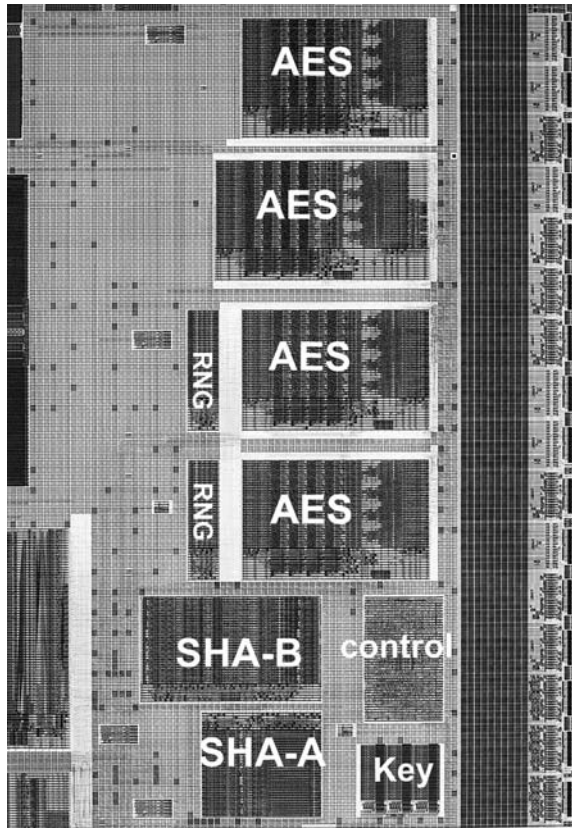


FIGURE 2.2
VIA C7 security components.

Full Custom. The entire component is designed at the schematic level and laid out by hand. There are about 74 custom block designs instantiated 319 times. In Figure 2.2, the AES key-RAM component is a full-custom element.

Datapath Stack. The component comprises several *n*-bit wide (typically 32 or 64 bits) library elements that "snap" together using common power rails and signal channels. The definition of the stack (the specification of the elements, their physical positioning, and the interconnects) is controlled through a special language. A Centaur tool creates the physical stack from this definition language by placing the stack elements and routing the signals. The library elements are designed using a full-custom approach: schematics to hand layout. There are 63 datapath stacks on this chip constructed out of a library of a few thousand elements. Many library elements do the same basic function except for differences in bit width,

timing, power, size, and drive strength. For example, the library contains 25 different two-input adders (which are instantiated 65 total times across the chip). In Figure 2.2, the blocks labeled AES, SHA, and RNG are 32-bit wide datapath stacks.

Synthesized Logic. There are 20 blocks on the chip of synthesized control logic built using the conventional methodology of compiling Verilog into a gate library. These are sometimes called auto-place and route (APR) or random logic blocks. The gate library used is a Centaur-designed custom library (of a few thousand elements) containing many specialized gates. In Figure 2.2, the block labeled "control" is an APR containing the control logic for all of the security functions.

2.2 RNG Design

2.2.1 Design Goals

The major objectives for our hardware RNG design were the following:

Truly Random at Large Sample Sizes. There is no test for a random number, per se. All that can be determined is the statistically calculated probability that a particular sample of bits came from a truly random source. Thus, determining if a generator produces random bits requires defining acceptable statistical criteria and statistically evaluating many samples. There are many test suites and statistical tests (and acceptable probability criteria) used in assessing randomness of generators. An old version of FIPS 140-2 defined some tests and criteria to be applied to a small sample of 20,000 bits; this "standard" has now been withdrawn due to problems with the randomness tests [NI01b], but the tests are still used in some environments. The most commonly used battery of tests is the Diehard suite, developed by George Marsaglia [Mar06]. Diehard tests typically use a 10-megabyte sample size (one test in one variation of this suite uses a 250-MB sample size). The Diehard suite of tests is primarily aimed at software random number generators, which have unique statistical characteristics. Other lesser-used collections of tests exist, but these suites are also primarily aimed at issues related to software generators and such relatively small sample sizes.

Our experience has shown that 10 MB is too small a sample size. It is relatively easy to find a large sample that is clearly not random (using standard statistical tests and criteria) but whose 10-MB subsets do pass the popular randomness test suites (such as Diehard). Thus, our design efforts were focused on randomness of large sample sizes. In particular, our goal was to generate bits that (1) were themselves statistically random up to one-gigabit sample sizes, and (2) had entropy sufficient to seed standard cryptographic mixing generators yielding statistically random samples up to one terabyte.

Interestingly, the largest single development effort on the VIA security functions was the verification of randomness of actual chips. An extensive

set of software had to be developed (and statistics had to be relearned) along with the collection and analysis of terabytes of samples from hundreds of parts. In addition, Cryptography Research Inc. has independently analyzed the VIA RNG mechanism and issued a detailed report on its characteristics and performance [CRI03].

Fast Bit Generation Speed. A fast bit generation rate is desired by some applications. More important, in order to meet our standards for randomness, we needed to analyze hundreds of gigabit samples taken at each interesting operating "corner" (voltage, temperature, etc.) of the processor. This required very high speed just to be able to collect the terabytes of data we needed to evaluate hardware.

This requirement effectively determined the technical approach used. Most conventional hardware RNGs use thermal noise on the chip as the fundamentally unstable component. Thermal noise, however, does not have a high inherent bit rate. Also, thermal noise does not scale with improving technologies: it is no "faster" at 65-nm technology than at 90 nm, for example. Accordingly, our design solution uses a new technique that is both fast and scales with technology.

Flexible Analysis and Tuning Options. Because it is impossible to verify the randomness of a hardware RNG during the design process (the randomness comes from unpredictable circuit variations), it is important to include features for experimentation and adjustment on the silicon version.

2.2.2 RNG Hardware

Figure 2.3 illustrates the VIA RNG hardware components.

2.2.2.1 Random-Bit Generator

The randomness of our bit generator derives from the interactions of four free-running oscillators. These oscillators are inherently unstable. Their frequency constantly changes due to their inherent instability, as well as manufacturing variations and voltage, temperature, and electrical environment changes. The outputs of two of the fast oscillators (running at a nominal frequency of about 1 GHz) are each divided by eight and then XORed. The resultant waveform is passed through an RC load (poly resistor and large gate capacitance) and used as the bias voltage adjustment for a third free-running oscillator, whose frequency varies from about 100 MHz to 30 MHz based on the bias input. The output of this oscillator is used as the clock for a one-bit register whose data is the output of another 1-GHz free-running oscillator. The output of this register is the raw generated bit.

These raw bits are not themselves truly random, but they do exhibit adequate entropy to be used as a good entropy source for mixing generators such as SHA or AES. This mixing of high-entropy bits is often used as a software random number generation approach. Because both SHA and AES are performed in hardware on our chips, this two-stage approach (raw bits into

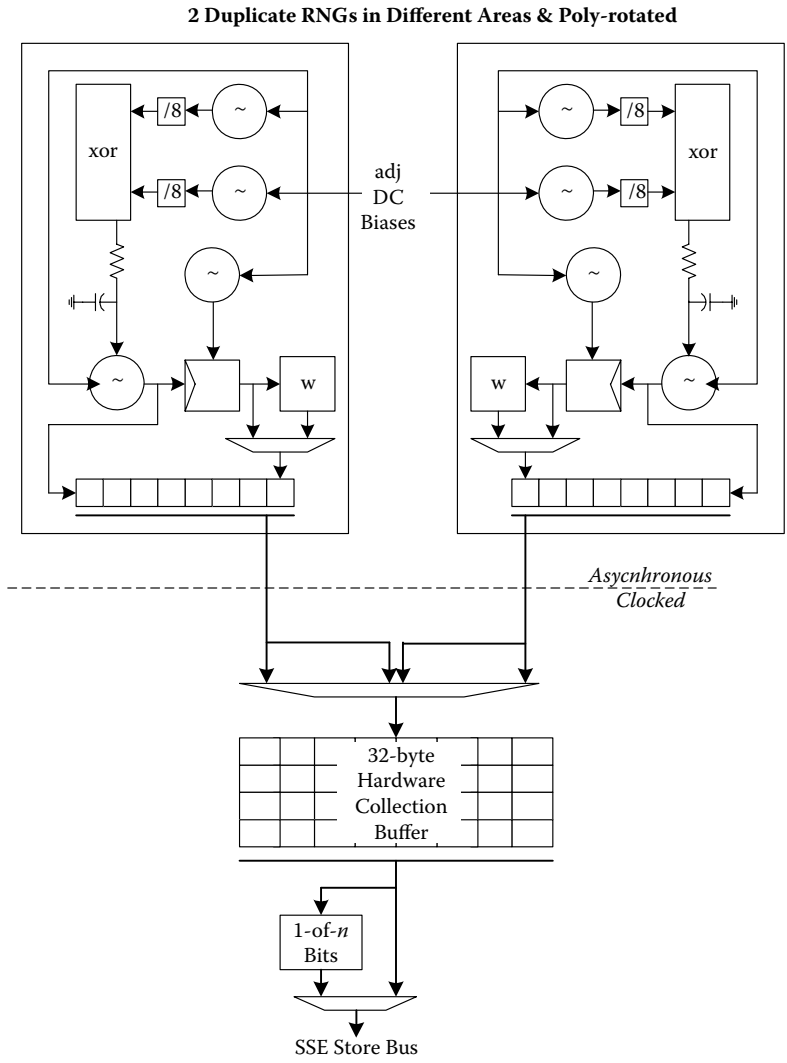


FIGURE 2.3
VIA random number generator design.

mixing algorithm) gives the highest random bit rate and the best statistical results for huge bit lengths (tested up to five-terabyte sample size).

Optionally, however, the raw bits can be fed through a Von Neumann "whitener" circuit to produce a direct high-quality random bit string. This circuit outputs a bit whenever a polarity of incoming bits changes. When used with our subsequent "1-of-*n*" bit selector, the whitened bit stream is sufficiently random to pass the popular Diehard criteria (10 megabyte samples).

The random bits (raw or whitened) are collected into an eight-bit buffer as the bits arrive. This buffer uses the same randomly generated clock as the

earlier described flip-flop. When this buffer is full, the associated processor interface logic is then signaled to read the byte. Note that this random byte generation mechanism runs asynchronously and completely overlaps with other processor activity.

VIA processors contain two of these hardware random bit generators. They are identical except one is rotated 90° on the silicon. This rotation introduces different poly-width variations between the two instantiations. The original intent of having two generators was in case one was more random than the other. In practice, they behave so similarly that both are typically used together to increase the data rate.

Each generator has an adjustable DC bias. Fuses set the default for this bias during manufacturing, but software can override that setting via the RNG control register. This bias adjustment was added for our internal experimentation purposes, but our general philosophy is to allow software to use all experimentation features. In practice, the default value works well and there is no reason for changing the bias.

To conserve power, the random bit generator is shut off when the processor is in a "sleep" mode. When exiting this low-power mode, both the current contents of the output buffer and the next complete generated byte are discarded. This reduces the risk of a glitch in the entropy of the generated bits due to startup behavior.

2.2.2.2 System Interface Logic

When signaled that eight new random bits are available, the system interface logic collects the random byte from the generator and places it into a 32-byte first-in first-out buffer (FIFO). Either generator, or both, can be selected to input bytes to the FIFO based on the control register setting. This operation is completely asynchronous and overlapped with other processor activity. The interface between the random generator and the system logic is simplified by design: after the generator buffer fills, the next processor clock boundary is guaranteed to occur before the arrival of the next random bit. This clock arrival resets the "byte available" signal to the system logic; thus, the system logic never reads the same byte twice.

Due to implementation specifics, a maximum of eight bytes of FIFO data may be stored by a non-REP x86 instruction. No data is stored, however, unless there are at least eight bytes in the FIFO. Any data read by an x86 instruction is removed from the FIFO. This ensures that random bits are never reused across multiple tasks.

A 1-of-*n* bit selector can further filter the random data bits read from the FIFO. The choices for eight FIFO bits are to return eight bits, four bits (every other bit), two (every fourth bit) bits, or one bit (every eighth bit). The control of the selector is specified in an x86 GPR when a "store RNG" instruction is executed. The intent of this selector is to improve randomness of whitened bits. To have good statistical characteristics at typical ten-megabyte sample sizes, the selector should be set to the one-of-eight rate. Smaller sample sizes

may be able to use more bits. Thus, based on the selector specified, and the number of bytes in the FIFO, the number of random data bytes returned on each x86 "store RNG" instruction request can be zero, one, two, four, or eight. The final random data output is multiplexed into the internal SSE result bus for delivery. This bus choice is transparent to software; the SSE bus was chosen based on processor floor plan considerations.

2.2.2.3 Implementation

Figure 2.2 shows the layout of a random bit generator. This tiny component (138 × 50 microns in 90-nm technology) is implemented as a full custom block. The system interface logic, including the FIFO, is implemented as two small 32-bit wide datapath stacks (shown in Figure 2.2 labeled as "RNG"), each with a size of 227 × 63 microns.

2.2.3 RNG Software Interface

The software interface to the hardware RNG mechanisms is a new "store random" (XSTORE) instruction. This instruction stores from zero to eight bytes of random data using a memory address in a general-purpose register (GPR). There are several unique features of this instruction beyond the simple store function:

Control Word Reporting. The RNG has selectable features that affect the delivered bits but the normal application probably shouldn't change (DC bias, raw vs. whitened bits). In this case, these features are controlled via a machine-specific register (MSR). This register can only be changed by a privileged program, typically, the BIOS or operating system kernel. A security-sensitive application, however, may care about these settings. For example, the application may be expecting whitened bits; if somehow the control word has changed to generate raw bits, the use of these bits may be incorrect within the application. Thus, the XSTORE instruction also returns the control register settings in a GPR on each "store RNG" bits instruction.

Bit Selector Control. The application can control the one-of-*n* bit selector by a control value placed in another GPR. As described above, this affects the statistical randomness of the stored bits.

Reporting Number of Bytes Returned. Because the number of random bytes returned for an XSTORE instruction is variable (zero through eight), the application must have a way of determining what was stored. Although the number of bytes stored can be obtained from the destination pointer (next topic), for convenience this number is also returned in another GPR.

REP String Capability. The REP XSTORE instruction operates similarly to an x86 store string (STOSB) instruction in that the destination

pointer in ES(EDI) is advanced by the number of bytes stored. Just as for STOS, the ECX register defines how many bytes need to be stored before the instruction completes normally. Interrupts and exceptions terminate the REP store in such a way that ECX and EDI correctly identify how many bytes have been stored. (One difference from x86 string instructions is that EDI always increments.) When the REP store random instruction completes, there is a contiguous stream of random bytes in memory.

There is a theoretical problem with using the REP XSTORE instruction because (1) the instruction does not return control until all requested bytes are stored, and (2) the generation bit rate is variable. The REP version of XSTORE prevents easy detection of a situation where the RNG hardware is not returning any bytes. In practice this not a real risk, since this situation cannot occur without a catastrophic hardware failure. We recommend that applications address this possibility by doing some non-REP XSTORES to verify that random bytes are being returned.

2.2.4 Performance and Randomness

The bit generation speed of a RNG is unlike other computer performance specifications in that the randomness "quality" of the generated data must also be specified. The topic of what do we mean by the randomness of a sample or a generator, and how to verify the randomness, is beyond the scope of this chapter; we provide here a simple summary of an extensive amount of verification and analysis.

The VIA RNG has two major options to control generation speed and thus the associated randomness of the generated data. In addition, because the hardware generation mechanism is variable based on "random" factors, no single speed reflects all parts or environments. Given these disclaimers, here is an attempt to summarize our RNG performance. "Random" here means that it passes standard statistical tests and criteria (the output always meets the other two random criteria: unpredictable and unrepeatable).

- For directly produced whitened bits, the average performance is approximately
  - 13.6 Mb/s using a divider of 1:1 (random for small sample sizes).
  - 3.4 Mb/s using a divider of 4:1 (random for FIPS-140 tests, sizes up to at least 20,000 bits).
  - 1.7 Mbs using a divider of 8:1 (random for up to 10 MB sample sizes).
- For raw bits (no whitening), the range of typical performance is approximately
  - 28–280 Mb/s (not random, but usable as an entropy source).

- Feeding raw bits (as entropy source) into AES yields a pseudorandom bit rate of
  - 20–200 Mb/s using a 1:1 ratio of entropy bits to output bits. This is statistically random up to terabit sample size, but a cryptographic purist will worry about its underlying randomness because the entropy of the input bits is less than one.
  - 10–100 Mb/s using 2:1 ratio of entropy bits to output bits. This is statistically random up to terabit sample size with no entropy concerns.
  - We have run up to 900 Mb/s using fewer entropy bits than output bits. This can be statistically random up to terabit sample size, but a purist will not accept this approach because the inherent entropy is low and thus it may be theoretically predictable.

2.3 AES Design

2.3.1 AES Standard Background

AES encrypts or decrypts only 16-byte data blocks. Key sizes are 128, 192, or 256 bits. Encryption or decryption of a block consists of multiple "rounds" of data transformations starting with the input data, and with the output of each round inputting to the next round. The functions performed are essentially the same for each round except that the original key is expanded such that a unique 128-bit "round key" is available for use in each round. For 128-bit keys, the number of rounds required is 10, for 192-bit keys, the number of rounds is 12, and for 256 bits, the number is 14.

For the purposes of understanding the algorithm as defined in FIPS publication 197, the 16 bytes of data processed each round should be thought of as being organized into a 4×4 matrix of bytes such that the first four bytes in the 16-byte data vector represent the first column, the second four the second column, and so on. The arithmetic (addition and multiplication) used in AES uses a particular finite field: GF(2^8) (a Galois field of order 2^8). Addition in this field is merely a bitwise XOR. Multiplication is more complicated: it is multiplication modulo of a certain irreducible polynomial in the field.

Each AES data transformation round comprises four steps in the following order.

1. Row-Shift. Various bytes within the 4×4 matrix rows are rotated within each row.
2. Byte substitution transform ("S-box"). This is a nonlinear transform of each of the 16 bytes from the row-shift operation. Each byte from the row-shift is transformed according to the same rules as

all other bytes, and each byte transformation is independent of any other bytes. This type of transformation is common among block ciphers and is usually called an "S-box." The transform values cannot be calculated quickly and some type of table lookup is usually used in software implementations.

3. Column-Mix. The four bytes in each column (resulting from the first two steps) are multiplied by a 4×4 constant matrix to produce the result column. Again, this multiplication is done in the particular Galois field used by AES.
4. Add round key. The final step for each round is to add (XOR) the unique round key with the result data.

After 10 rounds (for key size of 128 bits), the result from the last step is the 16-byte encrypted (or decrypted) value for the original 16-byte input data block.

The differences between encryption and decryption are:

1. The row shift rules are different: which bytes go where is different (one is the inverse of the other mode).
2. The S-box transform is different between the two modes (one is the mathematical inverse of the other mode). Thus, two S-box tables must be implemented.
3. The column-mix multiplication matrix is different (one is the mathematical inverse of the other mode). Thus, two multiplication matrixes must be implemented.
4. The key expansion algorithm is different. That is, even though the master key is the same for encryption and decryption, the round keys are different.

2.3.2 AES Design Topics

2.3.2.1 General Approach

Figure 2.4 is a logical view of the VIA AES hardware unit. Our goals and general priorities for this implementation were:

- Be as fast as possible given our normal processor implementation methodology and using the "rest-of-processor" clock speed. That is, we accepted the clock speed as a given, and our focus was to fit the AES design into that clock speed. In the case of the implementation described here, the clock speed is 2 GHz.
- Optimize file performance as well as individual block performance. We assumed that most encryption or decryption uses would involve multiple blocks using the same key (e.g., a file). Thus, we wanted to make the total multiblock operation as fast as possible. This had an

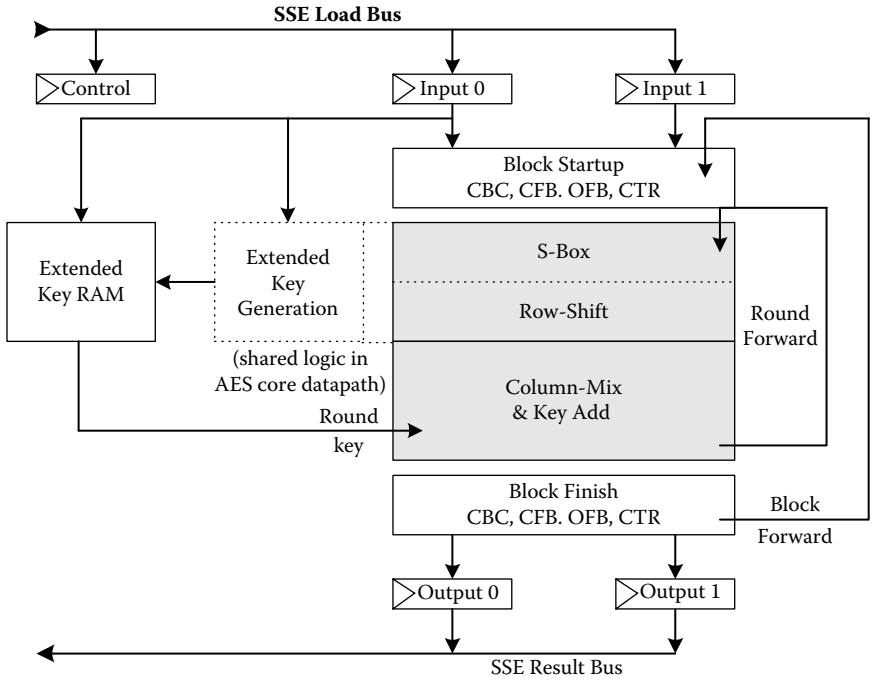


FIGURE 2.4
VIA AES unit design.

impact on the design of the x86 instruction interface as well as on the solution to the key expansion issues (see below).

- Provide complete standard functionality. We did not want to guess what people might want to use, so we implemented the entire standard in hardware: encryption, decryption, and all three optional key sizes. In addition, we implemented hardware support for generating the expanded key. This particular generation is questionable because (1) key expansion may not be important to performance (expansion is done only once per use of the key whereas the key may be used multiple times across many blocks of data), and (2) several different alternatives exist as to how to support key expansion; we chose a particular one. This complicated topic is discussed further in the next section.
- Make it easy to verify. This implied that we needed to support an "intermediate mode" that provides the result of each step. This is used to compare more closely with reference models as opposed to just looking at the final result.
- Be as easy to design as possible. This implied that we use our standard development methodology and library elements wherever possible.

2.3.2.2 Key Expansion

The expanded key required by the AES standard comprises a unique 128-bit key for each round of the block operation (10, 12, or 14 rounds). The expanded key is derived from the primary key by an algorithm using the same S-box transform as the main encryption operation as well as rotations and XORing with magic constants.

The first design choice we had to make was where does the expanded key reside during an encryption or decryption operation: generated on-the-fly, in an on-chip storage area, or in memory? The first option of generating the round key in parallel with the round logic was rejected by us because it required fairly large duplicate hardware (in particular, a duplicate S-box). In retrospect, we would now choose to provide this duplicate hardware because it would improve performance and security, but actually would have no affect on the total die size.

Having rejected calculating the expanded keys on-the-fly, the choice was now between getting the expanded keys from memory as we execute or preloading them into on-chip memory area. Although getting the expanded keys from memory as we encrypt is simple, this would significantly affect the speed of encryption. This approach also is a major security hole (the key can change during execution, etc.). Thus, we decided to maintain the entire expanded key in an internal on-chip RAM area. This "key RAM" can provide the specific 128-bit round key to the round logic each clock.

The next obvious question is from where does the expanded key come? We ended up supporting two options for getting the complete expanded key into the key RAM:

- Software can generate the entire expanded key in memory. An x86 AES instruction option causes microcode to load the complete expanded key from memory into the key RAM before encryption starts. This gives the user full control over the expanded key and avoids key-sharing complications with multitasking.
- A faster and more convenient option causes the hardware to generate the expanded key from the provided master key. The generation algorithm is much faster in hardware than in software. In addition, this hardware feature greatly simplifies software: the complicated key expansion algorithm does not need to exist in software. Note that this option is not the on-the-fly alternative discussed above; the entire expanded key is generated and stored in the key RAM before any block encryption starts. This approach allowed us to reuse the round logic (especially the S-box) for performing the key expansion: only a few extra muxes and XORs were added into the normal round logic. This hardware option, however, is not available for 192-and 256-bit key sizes due to practical reasons: the larger key sizes would have required significantly more extra hardware and introduced potential timing problems within the round logic.

2.3.2.3 Multitasking Support

Keeping the expanded key completely within the hardware introduces a major problem. Because there is no practical restriction on the size of a multiblock encryption or decryption operation invoked by one instruction, we must allow interrupts to be taken at various times during execution. (There is a limit on how long a PC can live with allowing interrupts to happen.) Assume task 1 is using the AES unit and is interrupted and some form of context switch occurs to task 2. The new task 2 now wants to use AES; this causes the key RAM of task 1 to be replaced. But when we later return to continue executing the AES instruction in task 1, the wrong expanded key (that of task 2) is in the hardware.

If we could assume that the OS knows about the AES unit, the solution is simple: just have the OS save and restore the key RAM along with the other context data. But we do not assume any OS knowledge. Our solution is to use a previously undefined bit (bit 30) in the x86 flags register (EFLAGS). When the key RAM is loaded with an expanded key, bit 30 is set to a 1. All of the x86 instructions or operations that save EFLAGS automatically clear this bit (including exceptions and task switch instructions).

Following the example above:

1. Task 1 starts an AES encryption or decryption. Because bit 30 is zero, the expanded key is calculated and placed in the internal key RAM (all AES instructions point to the normal key). Bit 30 is now set to a 1.
2. Task 1 continues to issue AES instructions. Because bit 30 is one, the existing expanded key in the key RAM is reused for each of these operations.
3. An interrupt occurs and the context is switched. EFLAGS is saved by the hardware on this transition and thus bit 30 is cleared.
4. Task 2 starts an AES encryption or decryption. Because bit 30 is zero, the expanded key is calculated and placed in the internal key RAM. Bit 30 is now set to a 1 so that task 2 can continue to use the AES hardware without any additional expanded key generation.
5. An IRET instruction ultimately returns us to task 1. Bit 30 is cleared by this instruction. Thus, the first AES use by task 1 causes the expanded key generation and the reload of the key RAM to occur.

Note that this entire mechanism of regenerating and reloading expanded keys is transparent (other than performance) to software; the hardware (really the microcode) does it when needed. The only affect on software is that whenever a key is used for the first time, bit 30 must be set to zero before the AES instruction is issued. We recommend using a PUSHF instruction (followed by a POPF to keep the stack consistent) before the use of AES instruction (remember that all AES instructions are REPs and thus operate typically on many blocks).

2.3.3 AES Hardware Design

2.3.3.1 Round Logic

To accommodate the speed goals and to fit into our existing implementation methodology, the standard definition of a round was modified slightly:

- The data was organized into four 4-byte words from consecutive bytes of the input. That is, our 4-byte words were the "columns" as defined in the standard. The reason is that each of our words became a 32-bit datapath stack (note the four AES stacks in Figure 2.2). The "row-shift" operation thus was a mux moving bytes across the four 32-bit stacks, but the complex "column-mix" operation is completely contained within one 32-bit stack.
- The order of the S-box substitution and the row-shift steps was interchanged; we do S-box followed by row shift. Because the S-box transform does not have any inter-byte affect, this change is logically transparent to the order in the standard. The reason for this change is performance. As described further subsequently, the S-box is implemented as a dynamic ROM, and thus needs to start the ROM translation on a clock boundary. The row shift can be done merely as a static wired mux. Because we needed to start a round on a clock edge, we had to put the ROM first.
- The column-mix operation (multiplication) is performed using a "sea" of multiway XORs (this is described further subsequently). Although mathematically equivalent to the standard definition, the equivalence is not particularly clear. Also, there is no discrete key-add step: this is performed by merely another XOR input into the multiplication logic.

It was obvious early in the design that this round approach could not be done in one processor clock, but likely could be done in two clocks. Thus, the design target, and the final implementation, was a two-clock round. In ECB mode, where each block is encrypted independently, the logic is pipelined so that the throughput is only one clock per round. For the other operating modes (CBC, etc.), where the encryption of each block uses data from the previous block's encryption, the throughput is two clocks per round.

A round thus comprises the following datapath logic.

1. Input register. The initial input data is loaded into this register and the forward data from step 6 comes into this register.
2. S-box lookup. This is done using a dynamic ROM. Thus, the input register is really part of the ROM implementation; basically the incoming bytes are the addresses into the ROM.
3. Row shift. This consists of intrastack wires feeding output of ROM into muxes in each 32-bit stack.

4. Clocked register. This is merely an internal pipeline register passing the data to the next clock.
5. Column mix and key add. This is a large fanout of the 32-bit data into a sea of cascading multibit XORs. The incoming round key data is merely one XOR input.
6. Forwarding results of step 5 back to the input register. On the last round, this data is passed to the end-of-block logic.

The control logic for this round path is simple: a counter controlling (1) the input to the round input register, (2) the selection of the round key from its storage RAM, and (3) the signaling when an encryption or decryption is complete. See Figure 2.5.

2.3.3.2 Interface Logic

Surrounding the round logic is the interface to microcode and logic to implement the supported encryption operating modes (EBC, CBC, CFB, OFB, CTR). The hardware interface comprises the following:

- Two 128-bit input registers and one small control register attached to the internal SSE source bus. The data registers feed the round logic with either two data blocks (for pipelined ECB) or one data block and one initialization vector (for the other operating modes). One of the data registers is also used to feed the extended key generation logic.
- Two output registers attached to the internal SSE result bus.

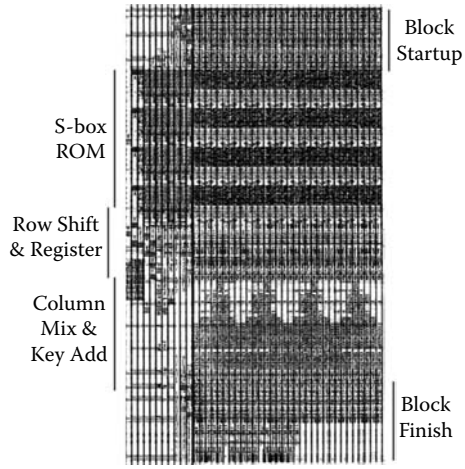


FIGURE 2.5
VIA AES unit stack implementation.

In addition to the round logic and timing (two clocks per round), there is an additional clock stage of logic at the start of the block and at the end (after all rounds are completed). This stage consists of the internal registers and muxes required to forward data to and from previous blocks to implement the operating modes other than ECB.

2.3.3.3 Expanded Key Logic

The logic to produce the expanded key is merely some extra muxes and XORs embedded within the round logic. This allows the key generation to use the S-box ROM used in round calculations.

2.3.4 AES Implementation Specifics

2.3.4.1 S-Box ROM

The AES S-box is implemented as a fast lookup table using a custom ROM built to fit into a datapath stack. Each stack has a four-byte ROM where the input address for each byte is nine bits (the eight bits of data and a bit saying encryption or decryption). The output is the eight data bits.

Due to the extensive wire load on the following step (row-shift), the ROM needed to be as fast as possible. It is implemented using dynamic (domino) logic where the bit lines are precharged high in the second phase of a clock, and held high or discharged low in the subsequent first phase. The address decodes that drive the "word select" lines are also precharged dynamic logic. To further improve speed, the address decode has two stages. The stage first selects word lines that select four-bit lines. The second stage generates the mux select among the four selected word lines. Bits are "programmed" in the ROM by diffusion. That is, a zero bit has no transistor connection between the word line and the bit lines. At our nominal timing corner (1.0 V, 100 C) the data to out time is about 210 ps. Its size is approximately 0.037 nm^2. The ROM timing could have been made considerably faster, except that additional speed was not needed given the target clock speed.

2.3.4.2 Column-Mix Logic

For the four bytes processed in the second clock of the round logic, each output byte is the sum of all four input bytes each multiplied by constants. Multiplication is done modulo a certain defined polynomial. As the standard document points out, this means that a multiplication by two can be done with a left shift of one bit followed by conditionally subtracting 0x1 B (subtracted if the high bit of the result byte is one). Thus, the multiplying by two ("two_x" below) can be implemented by wires (shift of one) and an XOR of a constant formed by bit 6 of the incoming byte (which will become bit 7 of the shifted result):

```
wire[7:0] two_x_mask = {0,0,0,in[6], in[6], 0,in[6], in[6]}; // 0 or 0x1B
wire[7:0] two_x = {in[6:0], 0} ^ two_x_mask.
```

As an example, consider the equation for encryption of byte 0:

$$B0_{out} = 2 \bullet B0_{in} + 3 \bullet B1_{in} + B2_{in} + B3_{in}.$$

Expanding each term and adding an XOR for the round key add yields:

```
wire[7:0] out_0 = two_x_b0_in ^ two_x_b1_in ^ b1_in ^ b2_in ^ b3_in ^
rnd_key_b0.
```

Because each two_x values contains an XOR, the full expanded equation consists of seven XORs. This can be expanded to 32 bits by writing out all the equations and collecting common terms such that the actual logic used for encryption column-mix of 32 bits becomes a two-to-one XOR (producing all four two_x bytes) followed by a six-to-one XOR (and a lot of wires).

For performance reasons, we actually use custom-designed multi-input XORs that are much faster than a sequence of two-to-one XORs. For example, the six-to-one XOR we use has a nominal propagation time of 86 ps, whereas the equivalent two-to-one XOR has a propagation time of 36 ps. This advantage of using a wide XOR circuit versus cascaded two-to-ones becomes even more apparent in the decryption logic, which ends with an eight-to-one XOR.

2.3.4.3 Expanded Key RAM

Each round of AES encryption or decryption requires a different 16-byte key, which is delivered from the key RAM. The key RAM component is a small single-ported register file. This component contains 16 entries of 16 bytes each. The reason for 16 entries is that our AES implementation allows a user-defined (nonstandard) number of rounds up to a maximum of 16. The clock-to-out timing of the custom-designed key RAM is about 245 ps and the size is 0.016 mm^2.

2.3.5 AES Performance

The VIA C5J performance (at 2 GHz) for instruction set performance (again assuming everything is in the cache) is approximately

- Single block, ECB mode, keys already loaded:
  - 1 block: 17 clocks
  - Large block count: 11.8 clocks/block average
- Single block, CBC etc modes, keys already loaded:
  - 1 block: 37 clocks
  - Large block count: 22.8 clocks/block average

- Expanded key generation/load
  - Hardware generated expanded keys: 38 clocks
  - Expanded keys loaded from memory: 53 clocks

Note that the time to load the expanded keys from memory does not include the time for software to generate them; this is a much longer time. All of these times include the microcode overhead.

To obtain comparative performance numbers for AES, we use the well-known Gladman library of cryptography functions [Gla06a]. This source code runs on any modern PC and is used by many applications. This software is highly optimized and is the fastest software implementation we know of with freely available source code. Because the code is distributed in source form, it is easy to modify it to support the VIA AES instructions. This allows us to measure very accurately the performance benefit an application will get on a real system.

Some sample performance using the Gladman library is shown in Table 2.2. Both the Intel P4 and the C5J system had the same memory size (512 MB) and speed, the processors had the same bus speed, the same hard drive was used, and so on. (The reason for the relatively obsolete P4 and the small memory sizes is that this information was created more than two years before this chapter was written). The caches were preloaded (by running an iteration of the code) and the numbers are the average of several consecutive runs after this point. The wall-clock time has been translated into gigabits per second (a common metric for security functions).

In addition to our runs, Dr. Gladman performed the same experiments and reported them here [Gla06b]. His maximum throughput results on an old 1.2-MHz VIA processor are about 15 Gb/s. This corresponds to our 21.5-Gb/s number on a 2-GHz processor.

Several conclusions are obvious. On data sizes that generally fit in the caches, the VIA hardware performance is about 40 times that of software (on a faster processor). The P4 is limited by processing speed, not by cache misses. When the data is large enough to substantially overflow the caches, the VIA solution becomes memory bound, but is still ten times the P4 software speed.

TABLE 2.2

Real System AES Performance

Data Size	2.53 GHz Intel P4 (Gb/s)	2.0 GHz C5J (Gb/s)
8 KB	≈0.56	≈21.5
64 KB	≈0.56	≈19.5
1 MB	≈0.56	≈5.45
10 MB	≈0.56	≈5.23
Etc.	≈0.56	≈5.23

2.3.6 Technology License Issues

Implementations of a symmetric-key encryption algorithm with key lengths greater than 64 bits are subject to U.S. export control restrictions. The VIA products containing AES have been reviewed by the appropriate U.S. government agencies and are classified as 5A0002 devices ("Systems, equipment, application specific electronic assemblies, modules and integrated circuits for information security") that require U.S. export licenses. One license is required for export of the design ("tapeout") to a foreign country and one required for sales of the final product in foreign countries. All VIA products containing AES have the appropriate U.S. government export licenses.

We mention this nontechnical topic as a warning to other engineers: obtaining these licenses was complicated and took longer than it took to design and implement the AES unit. A legal firm that specializes in export controls is strongly recommended.

2.4 SHA Design

2.4.1 SHA Standard Background

The secure hash standard as defined in FIPS 180-2 takes two inputs—a previously calculated message digest (hash) and a new message data block—and produces a new digest. For SHA-1 and SHA-256, the data block size is 512 bits and the digest size is 160 bits (SHA-1) or 256 bits (SHA-256). The algorithm is designed to be used on a large file of data with the digest from each 512-bit block being fed into the calculation of the digest of the next block.

The basic SHA-1 algorithm for producing a block digest is:

1. The 16 32-bit words (512 bits) of the message data are expanded into 80 32-bit words of data. This expanded data is called a message schedule. The additional words are formed by rotations and XORs of the input words.
2. Five 32-bit values—called a, b, c, d, and e—are initialized with the five 32-bit words of the incoming digest data.
3. Eighty iterative calculations are then performed. Each iteration uses a new expanded message word and the five values (a, b, c, d, e) produced from the prior iteration. A complex calculation then produces new values for a, b, c, d, and e. Four of the values are directly produced from the five input values by rotation and mixing. The fifth value is calculated by a five-to-one add (arithmetic) of functions of the five input values, a constant specific to the

iteration number, and the expanded message word. The critical terms are

$$T = \text{rotation(a)} + f_1(\text{b,c,d}) + \text{e} + K_\text{i} + \text{expanded\_key}_\text{i}.$$

$$a = T1.$$

(The $f(x)$ notation indicated some function derived from the values of x. The exact function is not critical to understanding the hardware except to note that the function is performed by special logic in the first clock of the iteration.)

4. After all 80 iterations, the resulting five intermediate values are added to the five parts of the original incoming message digest to produce the output message digest.

The SHA-256 algorithm is similar except that

- There are eight words of digest and intermediate values.
- The algorithm for calculating expanded message words is different, and there are only 64 iterations instead of the 80 for SHA-1.
- The calculation of the eight intermediate values is more complex: the critical components are the adds:

$$T1 = h + f_2(\text{e}) + f_3(\text{e,f,g}) + K_i + \text{expanded\_key}_\text{i}$$

$$T2 = f_4(a) + f_5(\text{a,b,c})$$

$$E = \text{d} + T1$$

$$A = T1 + T2.$$

2.4.2 SHA-1 Hardware Design

Figure 2.6 illustrates the SHA-1 subset of the VIA SHA hardware implementation. It closely follows the standard definition with several implementation optimizations. The datapath we use is generally (but not always) 32 bits wide. That allows us to easily perform one of the 80 iterations every two clocks. (We could have done the entire SHA-1 iteration in one clock, but that would have required unique SHA-1 and SHA-256 logic and we didn't want to expend the effort.)

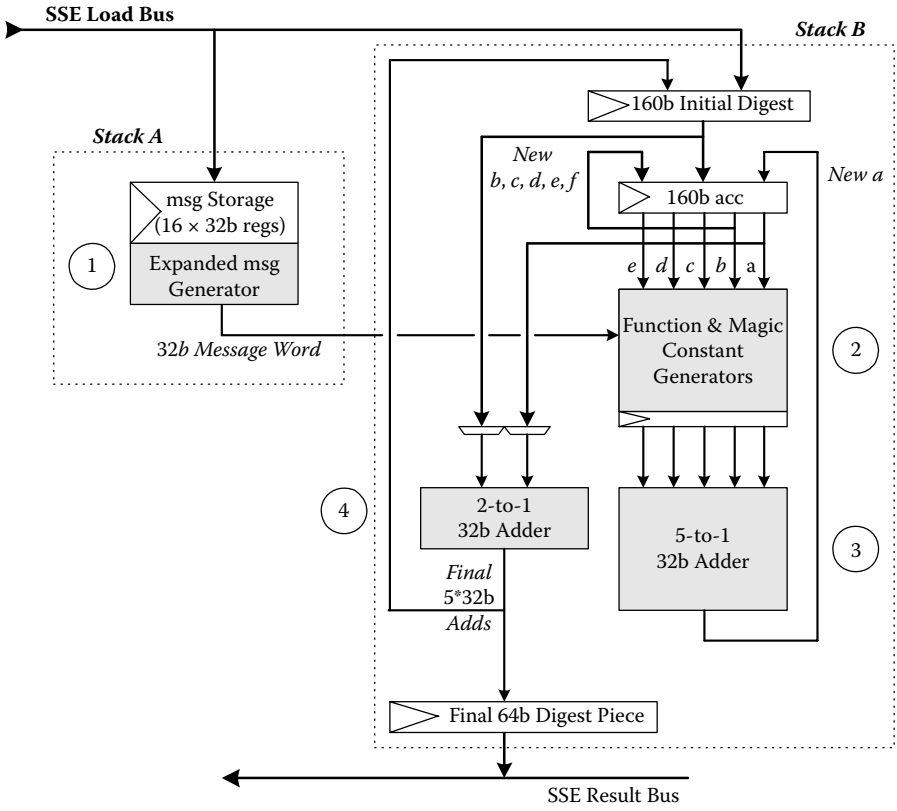


FIGURE 2.6
VIA SHA unit design.

Referring to Figure 2.6, the major features are:

1. Only the original 16 32-bit word data block is saved. The remaining 64 expanded words are produced on the fly from this saved data as needed.
2. Each of the 80 iterations takes two clocks. In the first clock, four of the new 32-bit outputs (b, c, d, e) are produced by wired muxes. In the same clock, four of the inputs are converted to three of the terms for the arithmetic add. An iteration-specific constant is generated with the logic and is the fourth add operand. The last add operand is the iteration-specific expanded message word.
3. In the second clock of an iteration, the five operands are added and the result is forwarded back to the input of the next iteration.

4. After all iterations are complete, five extra clocks at the end of the block operation are taken to add together the five 32-bit outputs of the iterations and the five portions of the original 160-bit digest. Note that the loads of the next block of message data can be overlapped with this final add step. Note that using a separate adder for this step is not necessary for SHA-1; the inputs could have been routed through the five-way adder. This separate adder is needed, however, for the SHA-256 calculation so it is also used for SHA-1.
5. Because the hardware is designed to execute multiple block digests, the results from step 4 are internally forwarded to the inputs for the next block calculations.

Figure 2.6 illustrates that the SHA components are contained within two separate datapath stacks as shown in Figure 2.2. The actual stack sizes (that include both SHA-1 and SHA-256 logic) are 0.046 and 0.069 mm^2.

2.4.3 SHA-256 Hardware Design

SHA-256 uses the same basic datapath as shown in Figure 2.6. The timing is three clocks per iteration rather than two for SHA-1. (Again, the clock timing is a function of our wanting to share as much logic as possible between SHA-1 and SHA-256 and being lazy). The main differences in the implementation over what is shown in Figure 2.6 are:

- There are eight 32-bit words of digest and intermediate values.
- The pipeline forwarding paths are changed for SHA-256. Four separate adds (three 2-way and one 5-way) are needed for each iteration. This is performed by circulating the data back through the second stage of the iteration logic, and thus the three clock timing. The first pass through this stage does the five-input add and a two-input add. The second pass does 2 two-input adds (one reuses the five-input adder).

2.4.4 SHA Performance

The SHA-1 timing for one block is approximately 251 clocks. The SHA-256 time is 262 clocks. These numbers include substantial overhead associated with multiblock optimization. For large blocks, and assuming data is in the cache, the performance approaches 190 and 225 clocks, respectively.

Some sample real-system performance is shown in Table 2.3. The data sizes are SHA "blocks," which in this case are 64 bytes. Note that at 1,000,000 blocks, the functions are memory limited.

TABLE 2.3

Real System SHA Performance in Gb/s

Data Size (Blks)	2.53 GHz Intel P4		2.0 GHz C5J	
	SHA-1 (Gb/s)	SHA-256 (Gb/s)	SHA-1 (Gb/s)	SHA-256 (Gb/s)
10	≈0.07	≈0.04	≈0.38	≈0.35
100	≈0.43	≈0.24	≈2.41	≈2.24
1,000	≈0.59	≈0.33	≈3.81	≈3.60
1,000,000	≈0.62	≈0.34	≈2.97	≈2.97

2.5 Montgomery Multiplier Design

2.5.1 Hardware Design

Unlike AES and SHA, where the entire standard algorithm was implemented in hardware, the Montgomery Multiplier implements only a 32-bit modular multiply step in hardware ($A \times B$ mod M). This step function is designed to be used by microcode to execute modular multiplies of indefinite length values. This scalability is very important because the recommended size of a RSA key has continued to increase, and a longer key always means improved security. The microcode component is quite complex in that it uses the 32-bit hardware multiply step to provide the full $A \times B$ mod M calculation for values up to 32,768 bits long. We know of no other hardware-based modular multiply implementation that can handle values this big.

Unlike AES, SHA, or RNG, the Montgomery Multiplier hardware is not a separate or discrete component; it comprises several components added into an existing 32-bit multiplier stack. The major development effort for the Montgomery Multiplier was not adding the unique hardware datapath components; instead it was (1) figuring out what the proper hardware datapath should be to perform the mathematical function, and (2) writing the complicated microcode to perform the entire n-bit modular multiply. We do not cover the mathematical derivation here, but rather show the resultant design.

Figure 2.7 illustrates the additions to the normal multiplier circuitry. The shaded components represent the components added to the existing two 32×32 hardware multipliers. Obviously, there are hidden muxes selecting the Montgomery paths or the normal multiplier paths. Conceptually, the hardware function is used (by microcode) in a double loop multiplying each 32-bit word of the multiplicand (A) by all 32-bit words of the multiplier (B) modulo the 32-bit word of the modulus (M) (using the Montgomery modular mathematics), adding in the previous partial products,

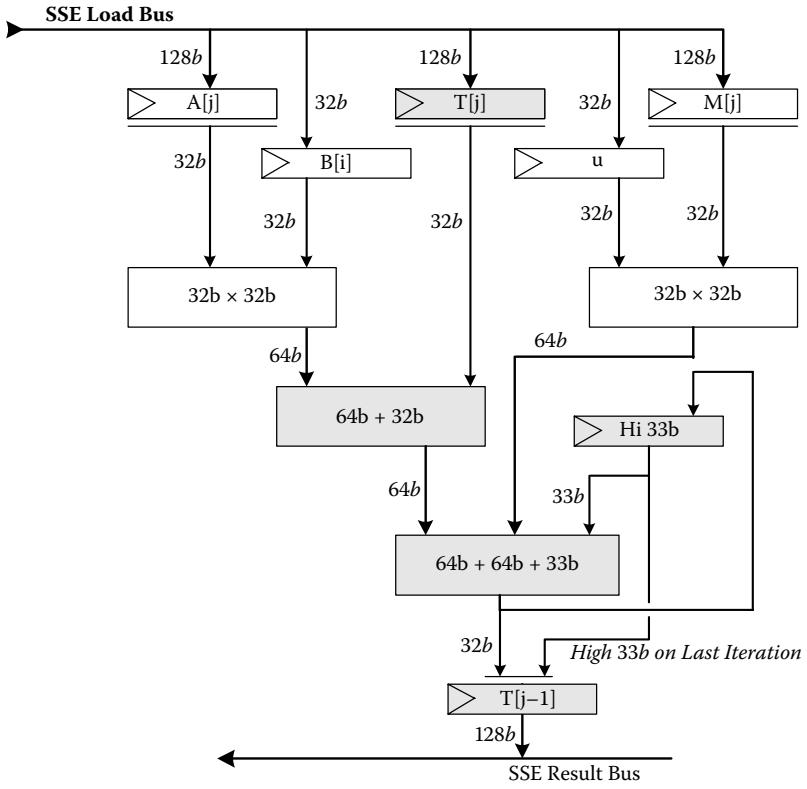


FIGURE 2.7
VIA Montgomery Multiplier design.

and accumulating the new partial products (*T*). A simplified view of this double loop is

```
for (j = 0; j < num_words_a; j++)
  for (i = 0; i < num_words_b; i++)
    T[j-1] = Montgomery_hdw(A[j], B[i],
      M[j], T[j], U);
```

U is a constant related to the transformation into the Montgomery number field. The [j – 1] indicates that the first low-order multiply result is thrown away.

2.5.2 Microcode Design

A major complication (and performance improvement) not totally obvious in Figure 2.7 is that four words of *A*, *M*, and *T* are loaded and stored at a time. This significantly improves performance because the Montgomery Multiplier is attached to a 128-bit load/store bus. To take advantage of this, the hardware can sequence the appropriate word out of the 128-bit input

registers and into the 128-bit output register. Thus, the microcode loads up the 128-bit values, issues the hardware Montgomery Multiplier instruction four times, and then stores the 128-bit partial product results.

The microcode overlaps loads, stores, addressing calculations, loop branches, and so on with the hardware multiply instructions such that (for data in the cache), the execution time approximates the hardware multiply time contribution.

2.5.3 Montgomery Multiply Performance

The hardware latency for a single 32-bit Montgomery Multiply step is four processor clocks, with a pipelined time of two processor clocks. Thus, assuming no microcode overhead, the total time for an n word by m word multiply can approach 2-nm clocks.

In practice, of course, cache misses may occur for real-world operands so the meaningful performance needs to be measured on a real system. Also, the end objective is a modular exponentiation (the RSA algorithm) not merely a big number multiply. Thus, a meaningful measure of the value of our implementation needs to be done on the total time to do a $c = m^e \bmod n$ calculation for RSA-sized keys. This total calculation includes many functions that also have to be done in software on VIA processors, thus diluting the performance gain of the Montgomery Multiply hardware.

The comparison software we used was the well-known big number library GMP [GM06]. Their claim is that they are the fastest library of big number functions and they specifically target cryptography applications. GMP includes a big-number modular exponentiation function that uses the Montgomery Multiply algorithm in software. For comparison purposes, we wrote a simple driver to perform the modular exponentiation function using our modular multiply instruction.

Following are some sample times for typical RSA-length operands. The actual values were chosen randomly.

- Modulus size = 1024 bits
  - 2.53 GHz P4 = 340 exponentiations/s avg.
  - 2.00 GHz VIA C7 = 1800 exponentiations/s avg.
- Modulus size = 2048 bits
  - 2.53 GHz P4 = 7.1 exponentiations/s avg.
  - 2.00 GHz VIA C7 = 35 exponentiations/s avg.

Note that there are many variables that affect the full RSA timing. Use the Chinese Remainder Theorem (CRT) or not? Use a sign or verify exponent? and so on. The above numbers used the CRT and the verify-sized exponent.

2.6 Summary Observations

Some after-the-fact observations about our integrated security implementation are:

- The architecture and high-level design for some of the functions was not obvious at the start and it took about one-half man-year to figure out how we were going to do things such as the RNG and Montgomery Multiply.
- The actual implementation effort (logic, circuit, layout, microcode) was relatively easy: maybe two man-years total. The largest component in terms of effort was microcode.
- Verification was easy except for the RNG. The non-RNG functions are algorithmic and are easy to model in software, and easy to test because they have simple pass/fail criteria. The total verification effort for non-RNG components was less than one-half man-year.
- The RNG could only be tested in silicon and there is no simple pass/fail test. Rather, extensive statistical testing at many silicon "corners" had to be performed. This took a lot of effort: maybe one man-year total.
- An unexpected (to us) complication was obtaining U.S. export licenses. But, having now learned (and having hired a good lawyer), the process is easy.
- AES and SHA functions could have been made smaller, maybe by 20%. We didn't do this because there was no need relative to the chip's floor plan.
- The AES two-clock round could not be made faster given our basic design methodology and technology. However, the two-clock SHA-1 iteration could be made to work in one clock and we probably could get the three-clock SHA-256 iteration down to two clocks with a new custom seven-input adder, but it wasn't worth the effort to us.
- Working with the open source software community is very easy. All of the major open source operating systems (Linux, Open BSD, and Free BSD) use our instructions in the kernel. In addition, major software libraries such as OpenSSL directly use our instructions.

In retrospect, now having a lot of real application usage, we realize that some things should have been done differently. Some of these changes are implemented in our next processor design (not yet shipping).

- We would implement an on-the-fly expanded key calculation for all key sizes. This is not trivial, but it can be done. This approach would

eliminate the complication of having to protect the expanded-key RAM from task switches. It also removes any need for software to have to do anything except invoke the AES instructions (today, 192- and 256-bit keys require software to generate the expanded keys).

- We would have supported unaligned (relative to a 16-byte boundary) data blocks for AES and SHA in our first implementation. We originally assumed, however, that software would always have the input blocks aligned. We discovered situations where this was not true and had to change the implementation in later versions. Our currently shipping implementation of AES does support unaligned data.
- We would implement a "progressive" version of the SHA instruction that does not do the padding required by the SHA standard. Our current implementation adds the padding automatically. This is convenient for software that has all the data for a hash available when the SHA instruction is executed. But this is inconvenient for software that obtains the data for the hash a piece at a time.

References

[Bar06] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," NIST Special Publication 800-90, June 2006.

[CRI03] Cryptographic Research Inc., *Evaluation of VIA C3 Nehemiah Random Number Generator,* February 2003, available at http://www.cryptography.com/resources/whitepapers/VIA_rng.pdf.

[Dae02] J. Daemon and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*, Springer, New York, 2002.

[Dwo01] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators," NIST Special Publication 800-38A, December 2001.

[Gla06a] B. Gladman Web site at http://fp.gladman.plus.com/AES/index.htm.

[Gla06b] B. Gladman Web site at http://fp.gladman.plus.com/ACE/.

[Kel00] J. Kelsey, B. Schneier, and N. Ferguson. Yarrow-160: Notes on the design and analysis of the yarrow cryptographic pseudorandom number generator, In H. Heys and C. Adams, editors, *Selected Areas in Cryptography, SAC'99*, LNCS1758, Springer, New York, 2000.

[GM06] GMP library information and code found at http://www.swox.com/gmp/.

[Mon85] P. L. Montgomery. Modular multiplication without trial division, *Mathematics of Computation,* 44(170):519–521, April 1985.

[NI00] National Institute of Standards and Technology, *Digital Signature Standard (DSS)*, January 2000. FIPS PUB 186-2 available from http://csrc.nist.gov/publications/fips/fips186-2/fips186-2-change1.pdf.

[NI01a] National Institute of Standards and Technology, *Advanced Encryption Standard (AES)*, November 2001. FIPS PUB 197 available from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[NI01b] National Institute of Standards and Technology, *Security Requirements for Cryptographic Modules*, May 2001. FIPS PUB 140-2 available from http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf.

[NI02] National Institute of Standards and Technology, *Secure Hash Standard*, August 2002. FIPS PUB 180-2 available from http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf.

[Mar06] Marsaglia Random Number CDROM, available from http://www.stat.fsu.edu/pub/diehard/.

[Sch96] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, John Wiley & Sons, New York, 1996.

[Qua06] Quantis Random Number Generators, http://www.idquantique.com/products/quantis.htm.

[VIA05] *VIA Padlock Programmers Guide,* found at http://www.via.com.tw/en/downloads/whitepapers/initiatives/padlock/programming_guide.pdf.

[VIA06] Material about VIA security features found at http://www.via.com.tw/en/initiatives/padlock/software.jsp.

3

ARM Cortex-A8: A High-Performance Processor for Low-Power Applications

David Williamson
ARM Inc.

CONTENTS

3.1 Cortex-A8

3.1.1 Overview

The ARM Cortex-A8 is a microprocessor targeted at systems that require high performance for both general-purpose and media applications while maintaining a low, sub 1 Watt, power profile and a small silicon footprint. This processor targets a significantly higher performance point than any previous ARM processor. The increased level of performance for general-purpose applications is realized through an energy-efficient balance of both increased operating frequency and improvements in machine efficiency as measured by instructions per cycle (IPC). The increase in frequency is achieved using a deeper pipeline with less logic depth per stage when compared to previous ARM cores. The increase in IPC comes mainly from superscalar execution of instructions, but the improved branch prediction, efficient memory system, and other features contribute as well to the machine performance. Performance for media and graphics applications is increased even further than what is achieved for general purpose applications with a 64-bit SIMD integer and floating-point engine (NEON).

3.1.2 Instruction Set Architecture

The ARM architecture is a load/store architecture with an instruction set that is largely consistent with other RISC processors. Some special attributes worth mentioning include instructions capable of both shift and ALU operations in the same instruction, the ability to use the program counter as a general-purpose register, support for variable 16-bit and 32-bit instruction opcodes, and a fully conditional instruction set. The ARM integer register file includes 16 32-bit registers; 13 of these registers are general purpose. The remaining three special-purpose registers are the stack pointer, a link register, and the program counter. Although these registers have special uses, they can also be used by most data processing and load/store instructions. The classic floating-point and new NEON media instructions both use a second register file that contains 32 64-bit registers. When used for integer SIMD operations, each register can contain a single 64-bit value, two 32-bit values, four 16-bit values, or eight 8-bit integer values. When used for floating-point operations, each register can contain a single 64-bit double precision value or two 32-bit single precision values.

3.1.3 Basic Pipeline Description

Cortex-A8 is an in-order, dual-issue superscalar processor with in-order instruction issue, execution, and retire. The processor has a 13-stage main pipeline that is used for all instructions. This main pipeline can be broken into three decoupled parts: fetch, decode, and execute. Individual pipeline stages within each part are simply numbered F1, F2, D0, D1, and so on. The two fetch stages at the front of the pipeline are responsible for predicting the instruction stream, fetching instructions from memory, and placing the fetched instructions into a buffer for consumption by the decode pipeline. The five decode stages take care of decoding, scheduling, and issuing instructions. They also deal with sequencing complex instructions and replaying instruction sequences when a memory stall occurs. The six execute stages consist of two symmetric ALU pipelines, a load-store pipeline, and a multiply pipeline. In addition to the main pipeline, there is a 10-stage pipeline for the NEON SIMD execution engine, an eight-stage pipeline for the level-2 memory system, and a 13-stage pipeline for the debug trace generation. The 10-stage NEON pipeline includes four stages of instruction decode and issue and six stages for instruction execution. NEON instruction decode stages are numbered M0, M1, M2, …, and NEON execute stages are numbered N1, N2, …. Level-2 memory system pipeline stages are numbered L1, L2, …. A diagram including all stages in the full pipeline can be seen in Figure 3.1. Each main section of the pipeline is discussed in more detail in the subsequent sections.

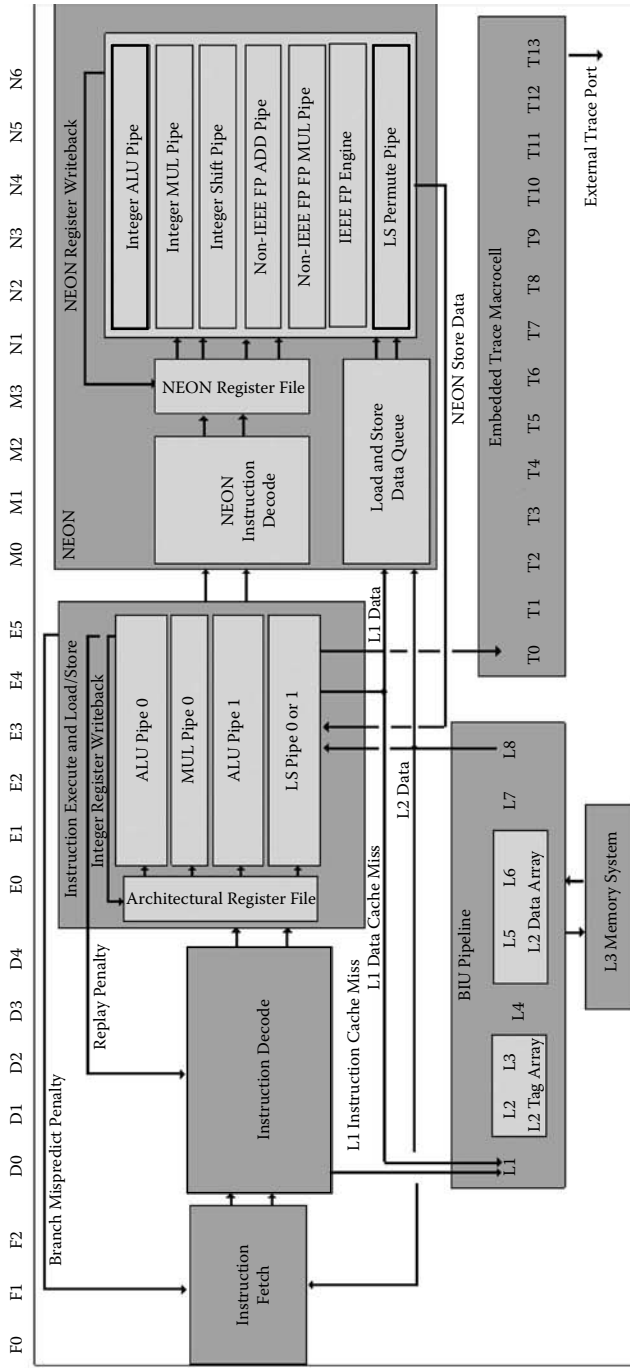


FIGURE 3.1
Cortex-A8 full pipeline.

3.2 Instruction Fetch

3.2.1 Instruction Fetch Pipeline Overview

The instruction fetch unit (I-fetch unit) includes the entire level-1 instruction-side memory system as well as dynamic branch prediction, and instruction queuing hardware. The instruction fetch pipeline runs decoupled from the rest of the processor, speculatively fetching up to four instructions per cycle along the predicted execution stream and placing them in the instruction queue to be consumed by the decode unit.

The fetch pipeline begins with the F0 stage where a new virtual address is generated. This address can either be a branch target address provided by a branch prediction for a previous instruction, or if there is no prediction made this cycle, the next address will be calculated sequentially from the fetch address used in the previous cycle. Note that the F0 fetch stage is not counted as an official stage in the 13-stage main integer pipeline. This is because ARM processor pipelines have always counted stages beginning with the instruction cache access as the first stage.

Once an address has been calculated, it is used to access the instruction cache arrays to obtain data for the next set of instructions in the F1 stage. In parallel, the fetch address is also used in the F1 stage to access the branch prediction arrays to determine if a branch prediction should be made for the next fetch address.

In the final fetch pipeline stage, the F2 stage, instruction data is returned from the instruction cache (assuming a hit occurs) and placed into the instruction queue (or the queue bypass registers) for future consumption by the decode unit. Also in the F2 stage, if this instruction results in a branch prediction, the new target address is sent to the address generation unit to be used as the next fetch address.

When a branch prediction for a taken branch is made, the instruction currently in the F2 stage changes the fetch address that is calculated in the F0 stage. Therefore, the instruction fetch currently in the F1 stage will need to be thrown away. This means there is a one-cycle bubble in the fetch pipeline whenever a branch prediction is made for a taken branch. Typically, this bubble is not exposed because the fetch engine runs ahead of the rest of the machine, but it can be exposed in branch-heavy code sequences or any taken branch that closely follows a branch misprediction.

3.2.2 Instruction Cache

The instruction cache is the largest component of the instruction fetch unit. It is a physically addressed, four-way set associative cache capable of returning 64 bits of data per access, and it is configurable to be either 16 KB or 32 KB in size. The cache line length is 64 bytes and line replacement is done using a random policy. The instruction cache also includes a 32-entry, fully associative

TLB. TLB misses are serviced by a hardware table walk mechanism that is part of the level-2 memory system.

To minimize design effort, the instruction and data caches are essentially identical, making use of the same array structures with only the minimum differences in the control logic that are needed to support each. In most ways, the level-1 cache design is a traditional cache design and it includes the typical data array, tag array, and TLB structures found in most processor caches. However, there is one additional structure in the Cortex-A8 that is not present as part of a traditional cache design: the hashed virtual address buffer (HVAB) array.

A traditional set-associative cache fires all ways of the data and tag RAMs in parallel at the same time that the physical address is being read from the TLB. This physical address is then compared with the values read from the tag array to determine which of the data RAM ways contains the required data and should be selected (or if a cache miss occurred). To avoid firing all these arrays in parallel, Cortex-A8 implements a way indication scheme based on a 6-bit hash of the virtual address and process ID of the access. This hash is used to index into the HVAB and determines the cache way that is likely to contain the data. This lookup is done quickly and is available in time to prevent firing of all the ways in both the data and tag arrays. A TLB translation and tag compare is still required in order to validate the hit. If the hit proves to be incorrect then the access is flushed, the HVAB and cache data is updated, and the access is repeated. Even though the TLB translation and tag read of the way containing the data are still required, they are removed from the critical path of the cache access, which is another benefit of the HVAB array.

Because the HVAB only brings a benefit in power savings when its hits are correct and actually costs performance and energy when its predictions are wrong, the key to its success is a good hash function that has a low likelihood of generating false matches. Another key attribute of a hash function in this application is that it can be evaluated relatively quickly. The hash used in Cortex-A8's caches is a two-level XOR reduction that mixes the address bits and process ID in a carefully selected order. Modeling this function across a large range of applications has shown a negligible increase in additional cache misses from the use of the hash function.

3.2.3 Instruction Queue

Fetched instructions from the instruction cache are placed into the instruction queue (IQ), or registered for forwarding on to the D0 stage if the IQ is empty. The purpose of the IQ is to absorb instruction delivery and consumption discontinuities between the instruction cache and the decode unit. The decoupling afforded by the queue allows the I-fetch unit to prefetch ahead of the rest of the integer unit and build up a backlog of instructions that are ready to be decoded. This backlog often hides the latency involved in predicting a change in the instruction stream and starting to fetch instructions

from a new location. The queue also prevents stalls from the decode unit from propagating back into the prefetch unit in the same cycle in which a stall condition is detected.

The IQ is organized physically as four parallel FIFOs, each six entries deep for a total of 24 entries. Each entry is 20-bits wide, holding 16 bits of instruction data and 4 bits of control state. Depending on the size of the instruction, an instruction may be contained within just one or two entries of the queue. Packing instructions in the queue in this manner can be complex to implement but it has efficiency advantages for the intermixed 16- and 32-bit long instructions that are common in the Thumb instruction set.

3.2.4 Branch Prediction

The branch predictor includes two arrays: the 512-entry branch target buffer (BTB) and the 4096-entry global history buffer (GHB). The BTB indicates whether the current fetch address will return a branch instruction and, if so, gives the branch target address. The GHB contains two-bit saturating counters that give the indication of whether conditional branches should be predicted taken or not taken.

The branch prediction arrays are both accessed in parallel with the instruction cache access in the F1 stage. The GHB entry is selected using a 10-bit global branch history and four low-order bits of the PC. Branch history is created from the taken/not taken status of the ten most recent branches. This information is saved in the global history register (GHR). Using branch history to determine prediction works well because heuristically instruction traces tend to take similar paths through a program creating different histories that predict the outcome on the next subsequent branch. The only flaw to a global history type of prediction is that it is possible to alias on two similar histories that differ in the nth branch where n is the number of history bits + 1. To help prevent this type of aliasing, low-order instruction address bits are also used to index the GHB. The GHB has 4096 entries, but is organized as a 256 entry by 32-bit array. So, only the upper eight bits of history are used to access the array and the final indexing based on remaining history and low-order PC bits is done after the array is accessed. Each access to the GHB reads out 16 two-bit prediction values, each of which indicates whether the next branch should be predicted taken or not taken. The 16 values are multiplexed down to a single prediction using an XOR combination of the remaining history bits and the low-order PC bits. GHB accesses always return a valid value and therefore there is no concept of a GHB miss. Instead, the GHB prediction is qualified by a hit from the BTB. To save power, the GHB is only accessed when the global history has changed. Because branch history is only updated on the prediction of a branch, the GHB array is not accessed any more often than necessary.

The BTB is indexed by the fetch address and contains branch target addresses and information about the branch type. The BTB stores predicted target addresses for both direct and indirect branches. On a BTB hit, if the

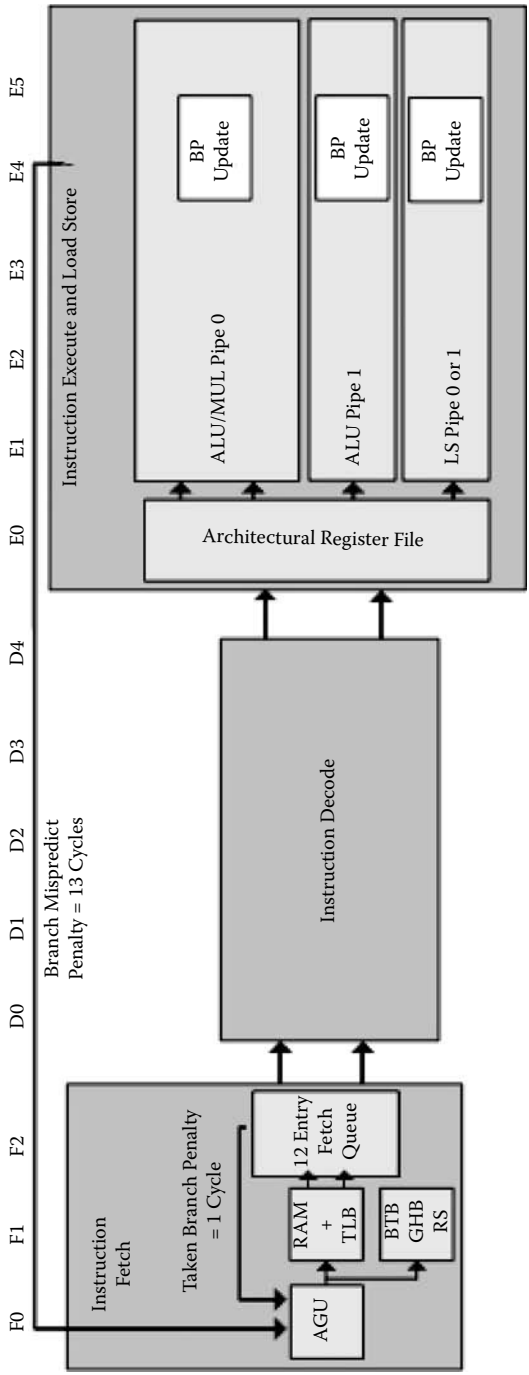


FIGURE 3.2
Instruction fetch pipeline and branch update.

entry is marked unconditional, the branch target address is used to fetch the next instruction. If the branch is conditional, then the value returned from the GHB access indicates whether the target address should be used. If the GHB lookup indicates that the branch was not taken, then the instruction cache continues fetching sequentially. On a BTB hit, the global history used to access the GHB is updated by shifting the taken/not taken status bit into the lowest-order bit of the global history. This update to the GHB history is done in the D2 stage (two cycles after the prediction).

All branch predictions are resolved in the E4 stage at the end of the integer pipeline by comparing the predicted and calculated PCs. If the branch mispredicts, or a taken branch is not predicted, the pipeline is flushed and the BTB and GHB arrays are updated accordingly. The global history register, used to index the GHB, is updated as well to a nonspeculative version of the global history to keep the predictor more accurate. In the case of a correct prediction, the GHB saturating counter and the nonspeculative global history register are both updated. Figure 3.2 shows the instruction fetch pipeline and the branch mispredict update path.

3.2.5 Return Stack

Cortex-A8 also makes use of a return stack for subroutine prediction. The subroutine return stack depth is eight entries. Return addresses are pushed onto the stack when the BTB lookup indicates that the branch is a subroutine call. When the BTB lookup indicates that the instruction is a subroutine return, the branch target address is popped from the return stack instead of being read from the BTB entry. Subroutines are often short and therefore it is important to support multiple push and pop operations in flight in the pipeline at a time. However, speculative updates to the return stack can be destructive because an update from the incorrect path can result in getting the return stack out of sync, generating multiple mispredictions. To achieve the performance benefits of speculative updates to the return stack without the performance cost, the instruction fetch unit maintains both a speculative and nonspeculative return stack. The speculative return stack is read and updated immediately based on the information returned from the BTB access and the nonspeculative stack is not updated until the branch is known to be nonspeculative at the end of the pipeline. If a branch is mispredicted, then the state of the speculative stack is overwritten by the state in the nonspeculative stack.

3.3 Instruction Decode

3.3.1 Instruction Decode Pipeline Overview

The instruction decode unit is responsible for decoding, sequencing, and issuing instructions. It also handles sequencing of exceptions and other

unusual events. The main blocks within the decode unit include the instruction decoders, the instruction sequencer, the pending and replay queue, and the instruction issue logic.

The logic of the decode unit occupies the D0–D4 stages of the pipeline. Early instruction decoding is done in pipeline stages D0 and D1. In these stages the instruction type, the source and destination operands, and resource requirements for the instruction are all determined. Multicycle instructions are broken down by the sequencer into multiple single-cycle operations in the D1 stage. These early stages generate all the decode information that is needed by the issue logic that lives in the D3 stage of the pipeline. The D2 stage is used to write instructions into and to read from the pending/replay queue structure. The instruction scheduling logic operates in D3. The scoreboard is read in this stage for all the operands of the next two instructions that could issue. These instructions are read from the pending queue or directly from the D2 stage if the pending queue is empty. The two instructions are also cross-checked against each other to check for any other dependency hazards that would not be detected by the scoreboard. The cross-checks and scoreboard results are combined to determine whether 0, 1, or 2 instructions will be issued. Once this issue decision is made and the next set of instructions is issued across the D3/D4 boundary, these instructions cannot be stalled. After this point, instructions advance one pipeline stage per cycle and the replay mechanism will be used to handle any unpredictable hazards from the memory system (cache miss, store buffer full, etc.). The D4 stage performs the final decode for all the control signals required by the I-execute and load-store units. These are then registered and sent to the I-execute and load-store units in E0. Figure 3.3 outlines the structure and the pipeline of the instruction decode unit.

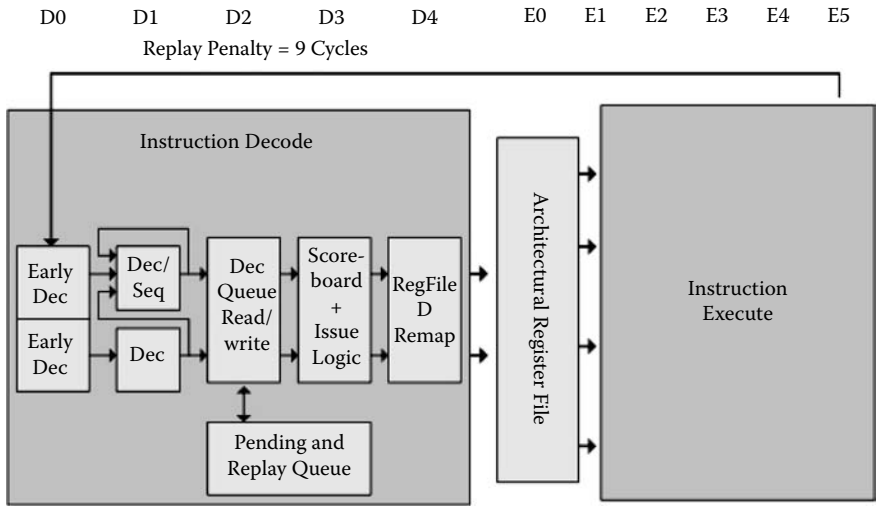


FIGURE 3.3
Instruction decode pipeline.

3.3.2 Static Scheduling Scoreboard

The scoreboard in Cortex-A8 is a predicting scoreboard that statically predicts when operands should become available. This is different and more complex than a traditional scoreboard that uses a single bit to indicate whether a source operand is ready for use. Rather than returning a single bit, the value read from the Cortex-A8 scoreboard indicates the number of cycles until a valid result will be available for forwarding to a following instruction. This information is used in combination with the source operand will be needed to determine if a dependency hazard should prevent the instruction from issuing. For example, we have an instruction being scheduled that requires the register R1 to be available as a source operand in the E2 stage. Because the scoreboard is accessed in the D3 stage, the E2 stage is four cycles away in the pipeline (D4, E0, E1, and E2). Thus, if the scoreboard indicates that the value for R1 will be available in four cycles or less, then no hazard exists on this source operand. However, if the scoreboard indicates that the value will not be available for five cycles or more, then the instruction could not issue due to the dependency. In order for this scoreboard to work properly, in addition to being updated whenever a new register is written, each entry in the scoreboard is also self-updating on a cycle-by-cycle basis. The value in each scoreboard entry will count down each cycle by one until updated by a new register write or the value reaches zero, indicating that the value is now available directly from the register file. In addition to indicating when an operand will be available, each scoreboard entry also contains information to track which execution pipeline is producing the register result and in what stage in the pipeline the producing instruction currently is. This additional information is used to generate the forwarding multiplexer control signals that are passed along with the consuming instruction when it is issued.

As mentioned above, the static scoreboard implemented in Cortex-A8 is more complex than a traditional scoreboard. However, there are several advantages to using this method. First, when used in combination with the replay queue, it allows the implementation of a fire-and-forget pipeline with no stalls in the execution pipeline. This is important for removing fundamental speedpaths from the design that would otherwise prevent high-frequency operation of the processor. Second, it is also a good technique to use in a low-power design because knowing early in the pipeline which execution units will be required each cycle allows aggressive clock gating without the creation of speedpaths in the design.

3.3.3 Instruction Scheduling

Cortex-A8 is a dual-issue processor with two main integer pipelines called "pipe0" and "pipe1." When two instructions are issued, pipe0 will always contain the older instruction in program order and pipe1 will always contain the younger instruction. This means that if the older instruction in pipe0 cannot issue, then the instruction in pipe1 will not issue, even if no hazard or resource conflicts exist for that instruction. When only one instruction is

issued, it is always issued in pipe0. Furthermore, all issued instructions will progress in order down the execution pipeline and retire with results written back into the register file in the E5 stage. This in-order nature of instruction issue and retire completely prevents WAR hazards and keeps tracking of WAW hazards and recovery from flush conditions straightforward.

If no RAW hazard is indicated from the scoreboard, the instruction in pipe0 should be free to issue. However, there are other constraints to consider besides just the scoreboard indicators for dual-issuing a pair of instructions. The first thing to consider is the instruction types for the two instructions and whether that combination is supported. In a given cycle, the following combinations of dual issue are supported.

- Any two data processing instructions
- One load–store instruction and one data processing instruction, in any order
- Older multiply instruction with a younger load–store or data processing instruction

In addition to the constraints listed above, only one of the two instructions issued can change the program counter. Instructions that change the PC include traditional branch instructions as well as any data processing or load instruction with the PC as the destination register.

In addition to the resource check, the two instructions are also cross-checked against each other to determine whether there are any RAW or WAW hazards between the two instructions that would prevent dual issue. If both instructions are writing to the same destination register (a WAW hazard), or if the younger instruction needs a destination register before it is produced from the older instruction (a RAW hazard), then dual issue is prevented. It's worth noting that when checking for a RAW hazard, it is not enough to simply check if the second instruction requires a source operand that is a destination register from the first instruction. In a similar fashion to what is done when accessing the scoreboard, a comparison is done between when the data will be produced by the older instruction and when it is needed by the younger instruction. If the data isn't needed until one cycle or more after it is produced, then the dependent pair can still be dual issued. Some examples of cases where this occurs include:

- Compare or subtract instruction that sets the flags followed by a flag-dependent conditional branch instruction
- Any ALU instruction followed by a dependent store of the ALU result to memory
- A move or shift instruction followed by a dependent ALU instruction

This dual issue of dependent instruction pairs is a key feature in the design that provides a significant performance increase inasmuch as the pairs of dependent instructions mentioned above are quite common in typical code sequences.

3.3.4 Replay and Pending Queue

The D2 stage of the decode pipeline is where instructions are inserted into and extracted from the pending/replay queue structure. This is a single structure with multiple read and write pointers used to hold instructions for two purposes. The pending portion of the queue holds new instructions that have not yet been issued whereas the replay entries in the queue hold instructions that have recently been issued and are still in flight in the execution pipeline.

The pending queue serves two purposes. First, it prevents the stall signal generated from the issue logic from rippling any farther up the pipeline. The determination in D3 of how many instructions will be issued occurs very late in the cycle. Therefore, it is important to limit the fanout of this signal to maintain the high-frequency operation of the design. The second purpose of the pending queue is to pack the pending instructions as closely as possible so there are always two instructions available to consider for dual issue. In the case where only one instruction is issued, it is important that the next two instructions can be considered together, even though they were originally sent from the fetch unit in different cycles. The packing done by the pending queue makes this possible and maximizes the opportunities for dual issue of instructions.

When instructions are issued, they move naturally from the pending queue to the replay queue simply by manipulating the queue pointers. The replay queue is the other key component, along with the static scheduling scoreboard, in the Cortex-A8 fire-and-forget pipeline. As mentioned in the scheduling section, instructions are statically scheduled in the D3 stage based on a prediction of when the source operand will be available. However, there are some cases, due to stalls from the memory system, when the data will not be available as expected. To handle these cases a recovery mechanism is used to flush all subsequent instructions in the execution pipelines and reissue (replay) them. The replay queue records every instruction in flight in the integer execution pipeline. Instructions are placed in the queue before they are issued and removed as they write back their results and retire. The replay queue tracks the information necessary to restart instruction execution cleanly from the issue point.

The most common memory system stall is a level-1 data cache miss. Therefore, the time taken to replay is balanced to be equal to the minimum level-2 cache hit latency of eight cycles; that is, if the instruction were stalled, rather than replayed, it would still spend eight cycles waiting for the data to be returned from the L2 cache. Most other causes of replay have latencies longer than eight cycles before they are resolved. To prevent multiple replays from occurring on a longer stall condition, an indicator from the memory system is used to hold the first replayed instruction until the appropriate time for it to issue.

The position of the replay queue in D2/D3 is a trade-off between performance and area/power costs. The earlier the queue in the pipe, the deeper but narrower it is and the greater the cycle cost of replay. At one extreme

replay could be implemented by refetching instructions from the I-cache. However, that would result in an unacceptably long replay penalty. At the other extreme, replay could be implemented using a queue of all the control signals crossing the decode–execute issue point, but that would result in an unacceptably large structure required for the replay queue and would also not allow it to be combined as easily with the pending queue. The best trade-off is to position the queue in D2. This allows for the combined structure with the pending queue and also matches well to the minimum L2 cache hit latency.

3.3.5 Multicycle Instructions

The ARM instruction set is considered a RISC instruction set and therefore the large majority of commonly executed instructions have a single opcode and make one pass through the execution pipeline. However, there are a few less commonly executed complex instructions that must be broken down into multiple instruction opcodes and make multiple passes through the execution pipeline. These instructions are called multicycle instructions. Multicycle instructions are unrolled into micro-ops in the sequencer which operates in the D1 stage. The micro-ops then move on to D2 to be queued in the pending and replay queue. Each micro-op of a multicycle operation is effectively treated as if it were an independent instruction. To help in the expanding of multicycle instructions into multiple micro-ops, a temporary register (Rtmp) is used to pass data from one micro-op to a subsequent one. Rtmp is scheduled using the scoreboard and issue logic just as with any other register, but it is only used to pass intermediate results between micro-ops that are part of a single multicycle instruction. When micro-ops from a multicycle instruction are issued, they will issue at the rate of one micro-op per cycle. However, it is possible to pair the last micro-op with a following independent instruction.

3.3.6 NEON SIMD Instructions

Although the bulk of advanced SIMD instruction decode takes part in the NEON unit itself, there is still some amount of work for I-decode to perform on these instructions. First, valid NEON and floating-point instructions are recognized and tagged for routing to the NEON engine and any undefined NEON instructions are trapped. Second, all NEON loads, stores, and register transfers are decoded and scheduled because the main pipeline is responsible for providing data to the NEON unit for these instructions. Third, before issuing NEON instructions, the decode unit must first check the number of NEON instructions currently in flight that have not yet been consumed by the NEON unit. If the NEON queue could overflow from too many instructions, then issue must stall until a slot in the NEON instruction queue is free. All decoding and scheduling of NEON instructions in addition to these cases is handled within the NEON unit.

3.4 Integer Execute

3.4.1 Integer Execute Pipeline Overview

The integer execution unit is pipelined across the E0 to E5 stages. It is responsible for doing the full execution of all ARM data processing, multiply, and traditional branch instructions. It is also responsible for maintaining the program counter, resolving condition codes for conditional instructions, generating addresses for load/store instructions, and prioritizing all potential pipeline flushes due to exceptions, branch misprediction, or a memory replay. NEON data processing instructions pass straight through the execution pipeline and are passed into the NEON instruction queue after the E5 stage.

The register bank is accessed in E0. Up to six registers can be read from the register file for two instructions (a maximum of four sources per instruction). After the register file is accessed, instructions are sent to one of two symmetrical ALU pipelines named Pipe-0 and Pipe-1. All instructions have an instruction component that is sent down either Pipe-0 or Pipe-1, even load/store, NEON, and multiply instructions that will not make use of the shifter or ALU. This component of the instruction is used for maintaining program order for instruction flushing, PC tracking, and data forwarding. The multiply–accumulate unit is bound to ALU Pipe-0 and therefore multiplies will always be the older instruction if dual issued. The load–store pipeline can be combined and used with either Pipe-0 or Pipe-1 on an instruction-by-instruction basis. This allows loads and stores to issue as either the older or the younger instruction, increasing the dual-issue opportunities.

In the two symmetric ALU pipelines, the E1 stage contains the shifter and the E2 stage includes the ALU. Most data processing operations are completed using either the shifter logic in E1 or the ALU present in E2. Some ARM instructions require use of both the shifter and the ALU which is why these blocks are pipelined instead of being implemented in parallel in a single execute stage. The E3 stage is used to complete saturation arithmetic used by some ARM data processing instructions. In E4, any change in control flow including branch mispredictions, exceptions, and memory system replays are prioritized and processed. The multiply pipeline does the actual multiply operation in the E1, E2, and E3 stages. Multiply accumulate instructions do the accumulate operation in the E4 stage. All results of ARM instructions are written back into the register file in the E5 stage. Even when a result for an instruction is available early, it is always written to the register file in order in the E5 stage and made available early using forwarding logic. This prevents the creation of any WAW hazards. See Figure 3.4.

3.4.2 Processing Flags and Conditional Instructions

As in most processor architectures, the ARM instruction set defines arithmetic flags such as carry, overflow, sign, and the like that are used to resolve branches and other conditional instructions. There are four of these flags

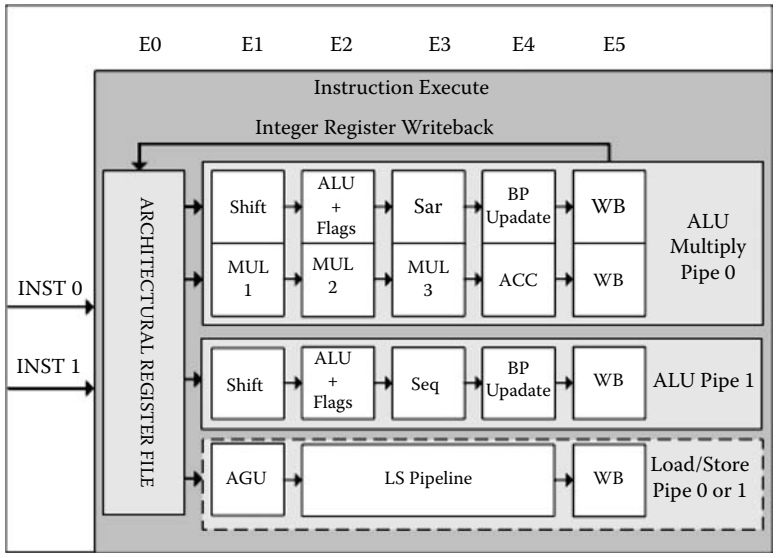


FIGURE 3.4
Execution pipeline.

and they are part of a larger processor status register called the CPSR. These flag bits are difficult to scoreboard in the traditional fashion because some instructions set just a subset of the flags. To scoreboard them properly, each individual bit would have to be tracked in the scoreboard as a separate register. This is too much overhead and complexity. Therefore, flags are not placed in the scoreboard as with other registers. Instead, maintenance of the latest values of the flags is handled within the execute unit in the E2 stage. To keep this maintenance as straightforward as possible, all instructions that update the flags or read the flags must do so in one of the two ALU pipelines in the E2 stage. This way, the latest copy of the flags can live physically in the E2 stage, generated each cycle by a merging of the flag outputs from the Pipe-0 ALU and the Pipe-1 ALU. Because Pipe-0 is guaranteed to contain the older instruction, it is always possible to create the correct final set of flags for use in the next cycle by merging the output from both instructions. Allowing two instructions to update the flags in parallel is particularly relevant to the Thumb instruction set where many instructions are only available in a flag-setting variant.

As mentioned in the architecture discussion, one of the unique features of the ARM instruction set is that almost all ARM instructions can be made conditional. Generally speaking, the ARM compiler makes fairly heavy use of conditional instructions to reduce the number of short branches. Therefore, it is important to handle the execution of conditional instructions in an efficient manner. From an architectural point of view, when an instruction fails its condition codes and is not executed, it becomes a NOP

(No OPeration). However, late conversion of an instruction to a NOP does not work well in a statically scheduled machine. Therefore, conditional instructions are implemented on Cortex-A8 such that they always generate a result regardless of whether they pass or fail their condition codes. This is done by reading the old value of the destination register as a source operand for the instruction. If a conditional instruction fails its condition code check, the old destination value will be passed on as the result of the instruction that failed its condition codes. As with all uses of the flags, condition code resolution is done in the E2 stage of the pipeline local to the two ALUs. Inasmuch as flags are forwarded between the ALU pipelines in the same cycle that they are generated, back-to-back condition code setting and use in a conditional instruction is fully supported.

On occasion, the entire Processor Status Register (CPSR) will need to be read by an instruction, including the latest value of the flags that are stored in the E2 stage of the execution pipeline. When a read of the full CPSR is required, the execution pipeline must first be allowed to drain of all outstanding operations so that the latest value of the flags can be read and merged with the rest of the CPSR register. This means that an instruction that reads the CPSR register will take a few additional cycles to execute. However, these instructions are not common, so this is a good trade-off to make in order to keep updates to the flags and condition code resolution as efficient as possible.

3.4.3 Forwarding Paths

One of the most important features in Cortex-A8 for efficient execution of code is the extensive support of key forwarding paths. Result data is forwarded from the outputs of both shifters, both ALUs, the multiplier, and the data cache. Data from each of these sources is made available for any consumer instruction that requires it as soon as it is produced. Also, once data is available for forwarding, it will continue to be available while the instruction is in flight until its result is written into the register file in the E5 stage. Supporting all of these forwarding cases requires a significant amount of data multiplexing. However, high-frequency operation is still achievable because the controls to steer the data to the correct pipeline are available early. Also, any noncritical sources of data (data that was produced in a previous cycle) can be combined early to reduce the number of critical inputs into the forwarding multiplexer to a maximum of six in all cases.

3.4.4 Exceptions and Branches

All control flow changes for mispredicted branches and exceptions are handled and prioritized at one time in the E4 stage. Resolving all branches and exceptions together allows for a relatively simple prioritization scheme and a single path for communicating all changes in the instruction stream to the prefetch and decode units. This simplifies the branch resolution logic

and the interface to the prefetch unit, and reduces the number of potential sources of the next fetch address. The downside of processing all branches in one stage is that you have to pick a point that is late enough in the pipeline for all branches to have their outcome resolved. This means a consistent, but high, branch mispredict penalty for all types of mispredicted branches. However, in most cases, a conditional branch is immediately preceded by the operation that will set the condition codes. Therefore, the number of times the branch could be resolved early is fairly small. Also, there is some upside in performance to late branch resolution inasmuch as this allows both the flag-setting instruction and the dependent branch to be issued in parallel.

3.5 Memory System

3.5.1 Memory System Pipeline Overview

The Cortex-A8 memory system consists of the integer load/store pipeline, the level-1 data cache, the level-2 data cache, and the bus interface unit. All data memory transfers are handled by the memory system including all NEON and floating-point load and store operations. All instruction cache misses are also handled by the level-2 cache and bus interface unit.

The load–store pipeline runs in parallel with the two ALU pipelines in stages E1–E3. In the E1 stage, the memory address is generated in the AGU from the base and index register. In the E2 stage, the address is applied to the cache arrays. In the case of a load operation, data is returned for formatting and forwarding in the E3 stage. In the case of a store operation, the store data is formatted and ready to be written into the cache in the E3 stage. Forwarding load data in the E3 stage results in a one-cycle load-use penalty for the common case of load data forwarded to the ALU. The penalty is two cycles if forwarding to a multiply, shift, or address calculation, and the penalty is zero cycles in the case of forwarding to a subsequent store instruction.

In the case of an L1 data cache load miss, a replay sequence occurs as described in the decode section and the miss request is sent to the level-2 cache pipeline. The latency for a level-2 cache access is a minimum of eight cycles. The L1 cycle is an arbitration stage where the next request to process is chosen among all pending requests from the load–store unit, instruction fetch unit, and many other cases. The L2 and L3 stages are where the tag RAM is accessed. In the L4 stage, it is determined whether a cache hit has occurred. If the access resulted in a cache hit, the data RAM is accessed in the L5 and L6 stages. In the L7 stage, data is returned to the load–store unit, and it is formatted and forwarded to the load instruction in the L8 stage. The load–store and L2 pipelines can be seen in the pipeline diagram in Figure 3.5.

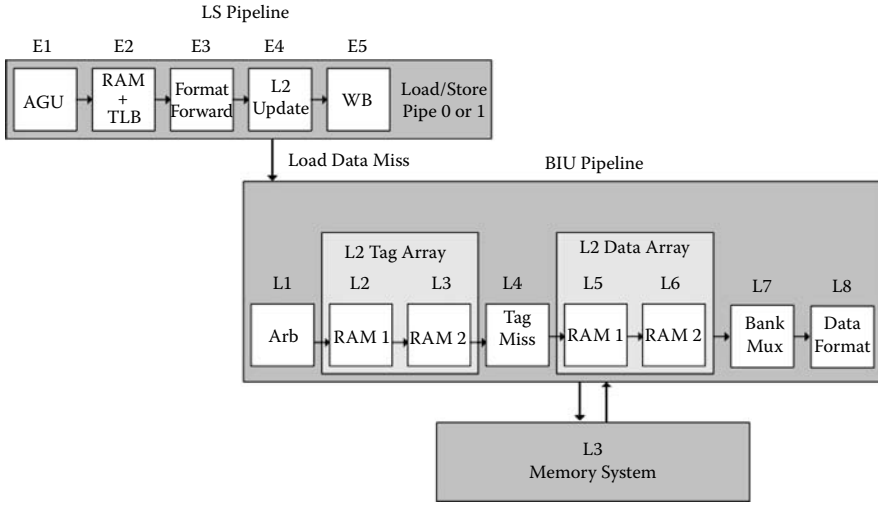


FIGURE 3.5
Memory system pipeline.

3.5.2 Level-1 Data-Side Memory System Structure

The D-side memory system uses many of the same components found in the instruction cache including using the same data array, the tag array, the TLB, and the HVAB array. Like the instruction cache, the data cache size is configurable between 16 KB and 32 KB. Also like the instruction cache, it is four-way set associative, physically addressed, has a 32-entry TLB, and uses a hash virtual address buffer (HVAB) way indicator scheme to improve timing and reduce power dissipation. Although many things are similar, there are also other additional components in the data-side memory system that are not required for the instruction cache. These include the integer store buffer, the NEON store buffer, and the data alignment and forwarding logic.

ARM integer stores pass through a three-entry, 64-bit store buffer with byte gathering (merging) on all entries before entering the data cache. This buffer is needed because stores are still speculative in the E3 stage and data should not be written into the cache until it is nonspeculative two cycles later. In addition, the buffer merges any subsequent stores that are to the same 64-bit location as one of the buffer entries. This store merging saves power because it reduces the number of required cache accesses. Full forwarding from the store buffer to subsequent load instructions is supported. So, no performance is lost from waiting to commit stores to the data cache. NEON stores pass through an eight-entry, 128-bit store buffer. The NEON stores have a deeper buffer than integer stores because the NEON pipeline follows the ARM pipeline and store data will not be provided from the NEON engine, allowing the buffer entry to retire, until the N3 stage of the NEON pipeline. Once a store is placed in the NEON store buffer in the E3 stage, data hazard checking is

performed to compare with all subsequent memory operations (both ARM integer and NEON). Assuming no hazard condition exists, these subsequent operations are allowed to complete and the NEON store is nonblocking. If an address hazard is detected, then the subsequent memory request will replay and wait for the NEON store to complete.

The level-1 data-side memory system supports reading or writing 64 bits per cycle for integer load and store instructions and 128 bits per cycle for NEON load and store instructions. Nonword-aligned reads are supported without additional latency in cases where the entire access falls completely within a naturally address-aligned 128-bit region. Nonword-aligned writes are supported without additional latency in cases where the access falls completely within a naturally address-aligned 64-bit region.

3.5.3 Nonblocking NEON Loads

NEON load operations that miss in the level-1 data cache do not generate a replay sequence. Instead, the missed request is passed from the level-1 memory system to the level-2 memory system. When data is returned from the level-2 memory system, it will be sent directly to the NEON load data queue and will not be returned to the level-1 memory system. In this way, NEON can stream data directly from the level-2 cache as if it were a level-1 cache. This is very useful for NEON applications because media code tends to work on streaming datasets that are more likely to miss in the L1 and often even the L2 cache. Therefore, it is beneficial for NEON performance to stream accesses with multiple misses outstanding with data coming from the L2 and also from the external memory system directly to NEON.

3.5.4 Level-2 Cache Structure

The level-2 cache is physically addressed, has a 64-byte line size, and is eight-way set associative with a random replacement policy. Write-back, write-through, and write-allocate policies are all supported, but write-allocate is the default, high-performance option. The cache size is configurable to be anything from 1 MB to 128 KB in size. A 0-KB size is also supported in the case where no L2 cache is desired. In the 0-KB scenario, the cache control logic is also removed from the design for additional area and power savings. The cache is pipelined to support multiple accesses in flight at once but RAM accesses take at least two cycles so four-way banking is implemented to allow back-to-back access for streaming data. For large cache sizes it is possible to configure wait states to allow more than two cycles to access the tag and data RAMs. The external bus interface is configurable to be either 128- or 64-bits wide and it supports multiple outstanding requests. The interface to the level-1 memory system includes a 128-bit read interface and a 128-bit write interface. This allows for efficient linefill operations that can complete in just four cycles.

3.5.5 Memory System Request Buffers

The memory system contains request buffers that all arbitrate for access to the level-2 cache and to the external memory system. This arbitration is done in the first stage of the level-2 cache pipeline. All requests, whether an initial request from the level-1 memory system, or a linefill request from the external memory system, arbitrate for cache access at this same point and then flow through the L2 pipeline. This simplifies hazard checking and reduces the complexity required to maintain proper memory transaction ordering rules.

There are three types of buffers that arbitrate for cache access: miss buffers, write buffers, and victim buffers.

3.5.5.1 Miss Buffers

The L2 unit includes three sets of miss buffers to track outstanding level-1 read misses for a total of 14 possible pending L2 read requests:

- IMB, Instruction-side Miss Buffer
- DMB, Integer Data-side Miss Buffer
- NMB, NEON (data) Miss Buffer

The 1-entry IMB holds the pending read request from the instruction-side memory subsystem. There can be only one outstanding request on the instruction side. The 1-entry DMB and 12-entry NMB hold pending read requests from the data-side memory subsystem. The DMB contains any pending read request for an ARM integer load instruction. The NMB holds up to 12 outstanding read requests for NEON and floating-point load instructions. ARM integer code can only have one outstanding read miss because the level-1 memory system is blocking after the first load miss for ARM integer loads. However, as mentioned in the earlier section, the level-1 memory system is nonblocking for NEON load misses to allow streaming of data from the L2 cache or external memory to NEON.

3.5.5.2 Write-Combining and Write Buffer

The L2 unit has an eight-entry, 128-bit wide write buffer and a two-entry 128-bit wide write combining buffer to handle subblock writes. Write combining is performed on incoming write requests to the same quadword. If the incoming write request is to a different quadword, the current contents of the write-combining buffer are transferred to the write data buffer and the incoming request is placed in the write-combining buffer. This write-combining capability saves power by reducing multiple writes to the same quadword down to a single store, and it also improves performance for the same reason.

Once valid, an entry in the write buffer will arbitrate for access to the L2 cache. If the write access results in a cache miss, the L2 unit supports write-allocate. Therefore, a victim buffer will be allocated and an L3 linefill request will be initiated similar to an L2 read miss.

As an additional optimization, if the full cache line is written, the level-2 line is simply marked dirty and no external memory requests are required. Of course, write accesses for the victim line may still be required. This optimization results in both a power and performance improvement in store streaming scenarios.

3.5.5.3 Victim Buffer

The L2 unit has a four-entry victim buffer to handle linefills and evictions. Each entry in the victim buffer (VB) can hold a full cache line of data (64 bytes). When a load or store request misses in the L2, a victim buffer is allocated to perform the linefill request. Once data has been returned from the L3 system, the VB will arbitrate for the L2 cache to read out the victim data and write in the fill data, essentially performing a swap. If the L2 victim is dirty, the VB will schedule a write-back of the L2 data to the L3 memory system. Each victim buffer uses a unique ID on the external bus, which allows for multiple requests outstanding at the same time and also allows the data from these requests to be returned out of order.

3.6 NEON Media Processing Engine

3.6.1 NEON Pipeline Overview

The NEON unit processes the advanced SIMD instruction set that consists of 32-bit SIMD integer and floating-point instructions that can operate on 128-bit, 64-bit, 32-bit, 16-bit, or 8-bit data values. These instructions were added to greatly accelerate processing of media-style workloads such as audio and video filters and codecs. The NEON unit also executes all ARM floating-point instructions from the pre-existing VFP instruction set. It makes sense for the NEON unit to execute both the new advanced SIMD instructions and the pre-existing VFP floating-point instructions because they share the same register file and load/store instructions.

The NEON media processing engine's pipeline starts at the end of the main integer pipeline. As a result, all instruction speculation is resolved before instructions reach the NEON pipeline. This reduces the complexity of the NEON Unit and also allows for a zero-cycle load-use penalty in most cases, even when data is returned from the L2 cache, due to the decoupling buffers used to hold pending instructions and data. The cost of putting the NEON engine at the end of the main integer pipeline is the longer delay required for writing values from the NEON register file to the ARM register file.

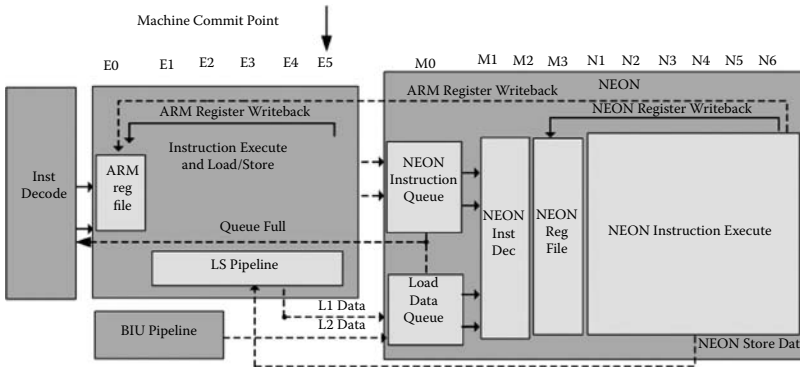


FIGURE 3.6
NEON engine interfaces.

However, this forwarding of data from a NEON register to an ARM register is not commonly seen in media code.

The NEON unit is decoupled from the ARM integer pipeline by the 16-entry NEON instruction queue (NIQ). NEON can receive up to two valid NEON instructions from the integer execute unit. Once there are enough instructions sent to NEON to fill the queue, the decode unit will wait until entries have been deallocated from the NIQ before sending any more instructions. In a similar fashion to what is seen on the instruction side, NEON is also decoupled from the memory system by the 12-entry load data queue. The load data queue can receive data from either the level-1 data cache or the level-2 memory system and this data can be received out of order. Store data is written out from the N3 stage in the NEON execution pipeline to the NEON store buffer. All the interfaces between the NEON engine and the other components of the processor can be seen in Figure 3.6.

3.6.2 NEON Pipeline

The NEON engine has its own pipeline that begins at the end of E5 and is a ten-stage pipeline. NEON instructions are either read from the instruction queue or directly from the E5 pipestage when the queue is empty. Instructions are always issued and retired in order.

NEON has four decode stages, M0–M3, and six execute stages, N1–N6. The four decode stages in M0–M3 are very similar in structure and design to the four decode stages D0–D4 seen in the main pipeline. The first two stages are used to decode the instruction resource and operand requirements and the later two stages are used for instruction scheduling. A static scoreboard with a fire-and-forget issue mechanism is used for the NEON pipeline in a similar way to what is seen in the ARM integer pipeline with the key difference being no requirement for a replay queue because there are no conditions under which a pipeline flush can occur.

The NEON decode logic is capable of dual issuing any LS permute instruction with any non-LS permute instruction. Dual-issuing these combinations of instructions requires fewer register ports than would be needed for dual-issuing two data processing instructions because LS data is provided directly from the load data queue. It is also the most useful pairing of instructions to dual-issue because significant load–store bandwidth is required to keep up with the advanced SIMD data processing operations.

The 32-entry register file is accessed in the M3 stage when the instruction(s) is (are) issued. Once issued, an instruction will be sent to one of seven execution pipelines: integer multiply, integer shift, integer ALU, NFP add, NFP multiply, IEEE floating point, or load/store permute. All execution datapath pipelines are balanced at six stages. The ten stages of the NEON pipeline including all the execution pipelines, can be seen in Figure 3.7.

3.6.3 NEON Execution Pipelines

NEON has three SIMD integer pipelines, a load–store/permute unit, two SIMD single-precision floating-point pipelines, and a nonpipelined IEEE compliant floating-point engine. All NEON and floating-point instructions are processed by one or more of these execution pipelines.

3.6.3.1 NEON Integer Pipelines

There are three execution pipelines responsible for executing NEON integer instructions, an integer multiply–accumulate (MAC) pipeline, an integer shift pipeline, and an integer ALU pipeline. The MAC unit consists of two 32 × 16 multiply arrays with two 64-bit accumulate units. The 32 × 16 multiplier array can perform four 8 × 8, two 16 × 16, or one 32 × 16 multiply operation in each cycle. The accumulate units have dedicated register read ports for the accumulate operand. The MAC unit is also optimized to support one multiply–accumulate operation per cycle for high performance on a sequence of MAC operations with a common accumulator.

The shift pipeline consists of three stages and is therefore a bit shorter than the other NEON pipelines. When only a shift result is required, it is made available early for subsequent instructions at the end of the N3 stage. Some instructions do a shift and accumulate operation. For these instructions, the result from the shift pipeline is forwarded to the MAC pipeline to complete the accumulate operation.

The NEON integer ALU consists of two 64-bit SIMD ALUs operating in parallel, with four 64-bit inputs. The first stage of the ALU pipeline, N1, is responsible for formatting the operands to be used in the next cycle. This includes inverting operands as needed for subtract operations, multiplexing vector element pairs for folding operations, and sign/zero-extend of operands. The N2 stage contains the main ALU which is responsible for all NEON

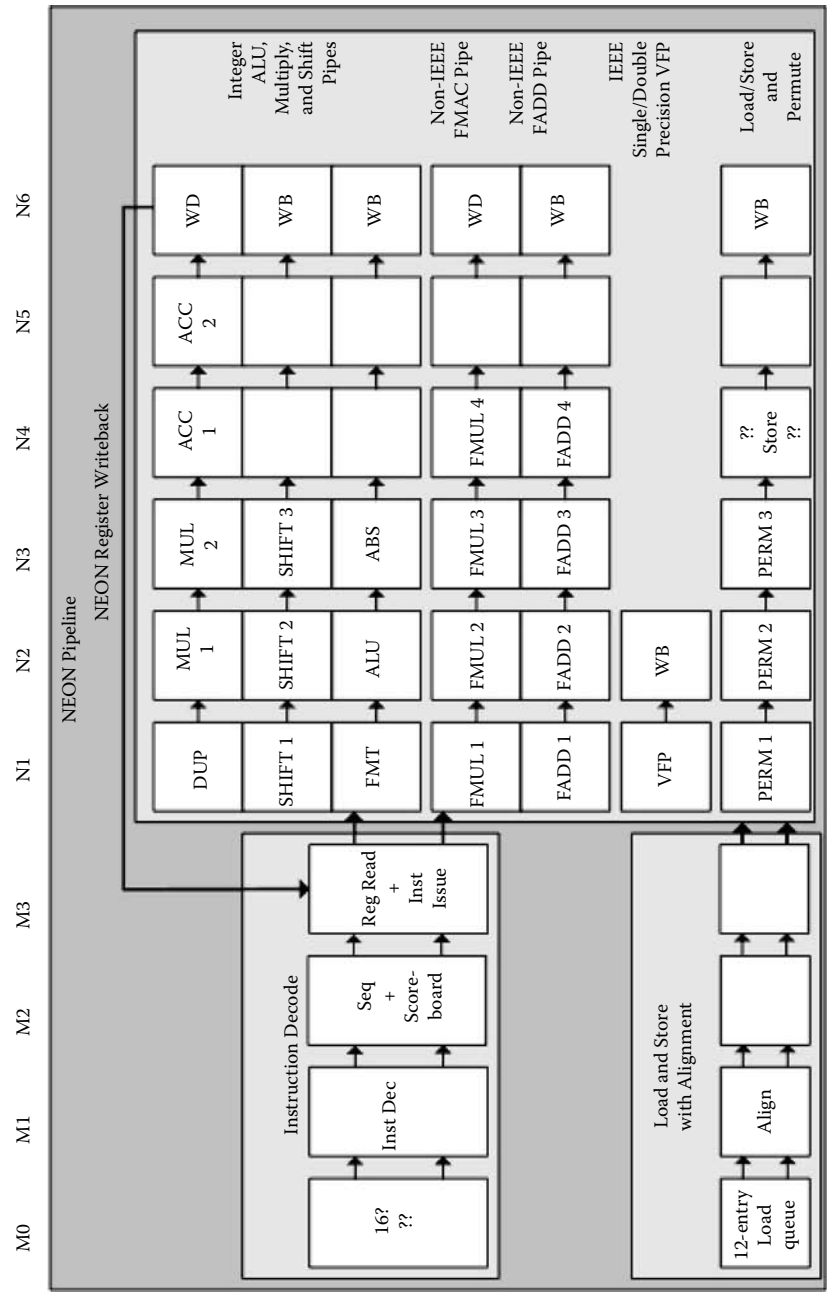


FIGURE 3.7
NEON pipeline stages.

SIMD integer add, subtract, logical, count leading-sign/zero, count set, and sum of element pairs operations. If this is the only operation required by the instruction, then the result is available for forwarding at the end of the N2 stage. The flags are also calculated in the N2 stage, to be used in the following stage. The N3 stage is responsible for the compare, test, and max/min operations and for saturation detection. It also has a SIMD incrementor for generating two's complement and rounding operations, and data formatter for high-half and halving operations. Similar to the shift pipeline, the ALU pipeline can also make use of the MAC accumulator in stages N4 and N5 for completing the final stages of the absolute-difference-accumulate operation.

3.6.3.2 NEON Load-Store/Permute Pipeline

The permute pipeline is fed by the load data queue (LDQ). The LDQ holds all data associated with NEON load accesses prior to entering the NEON permute pipeline. It is 12 entries deep and each entry is 128-bits wide. Data can be placed into the LDQ from either the L1 cache or L2 memory system. Accesses that hit in the L1 cache will return and commit the data to the LDQ. Accesses that miss in the L1 cache will initiate an L2 access. A pointer is attached with the load request as it proceeds down the L2 memory system pipeline. When the data is returned from the L2 cache, the pointer is used to update the LDQ entry reserved for this load request.

Each entry in the LDQ has a valid bit to indicate valid data returned from the L1 cache or L2. Entries in the LDQ can be filled by L1 or L2 out of order, but valid data within the LDQ must be read in program order. Entries at the front of the LDQ are read off in order. If a load instruction reaches the M2 issue stage before the corresponding data has arrived in the LDQ, then it will stall and wait for the data.

L1 and L2 data that is read out of the LDQ is aligned and formatted to be useful for the NEON execution units. Aligned/formatted data from the LDQ is multiplexed with NEON register read operands in the M3 stage before it is issued to the NEON execute pipeline.

The NEON LS/permute pipeline is responsible for all NEON load–stores, data transfers to or from the integer unit, and data permute operations. One of the more interesting features of the NEON instruction set is the data permute operations that can be done from register to register or as part of a load or store operation. These operations allow for the interleaving of bytes of memory into packed values in SIMD registers. For example, when adding two 8-byte vectors, you may wish to interleave all of the odd bytes of memory into register A and the even bytes into register B. The permute instructions in NEON allow you to do operations such as this natively in the instruction set and often with only using a single instruction.

This data permute functionality is implemented by the load–store permute pipeline. Any data permutation required is done across two stages, N1–N2. In the N3 stage, store data can forwarded from the permute pipeline and sent to the NEON store buffer in the memory system.

3.6.3.3 NEON Floating-Point Pipelines

The NEON floating-point (NFP) has two main pipelines: a six-stage multiply pipeline and a six-stage add pipeline. The add pipeline adds two single-precision floating-point numbers, producing a single-precision sum. The multiply pipeline multiplies two single-precision floating-point numbers, producing a single-precision product. In both cases, the pipelines are two-way SIMD, which means that two 32-bit results are produced in parallel when executing NFP instructions. Most classic ARM VFP single-precision floating-point instructions can also be executed in the NFP pipeline when IEEE-compliant mode is not enabled. When executing multiply–accumulate instructions, both pipelines are used back-to-back to produce the final result.

3.6.3.4 IEEE-Compliant Floating-Point Engine

The IEEE-compliant floating-point engine is a nonpipelined implementation of the ARM floating-point instruction set targeted for medium performance IEEE 754-compliant and double-precision floating-point. It is designed to provide general-purpose floating-point capabilities for a Cortex-A8 processor. This engine is not pipelined for most operations and modes, but instead iterates over a single instruction until it has completed. A subsequent operation will be stalled until the prior operation has fully completed execution and written the result to the register file. The IEEE-compliant engine will be used for any floating-point operation that cannot be executed in the NEON floating-point pipeline. This includes all double-precision operations and any floating-point operations run with IEEE precision enabled.

3.7 Implementation and Deployment

ARM is a provider of intellectual property. This means ARM does not manufacture any silicon, but instead provides complete processor designs to the customer, typically a silicon vendor. The silicon vendor will then integrate the ARM processor as an embedded component in one or more of their system-on-chip (SoC) products. ARM will typically provide the same processor design to multiple customers. This creates additional design challenges because each silicon vendor will have a different fabrication process and therefore will need a slightly different mask set to fabricate the design. This is normally dealt with by providing "soft" deliverables for the design. Soft IP means the customer is provided with an RTL-level description of the design along with a set of implementation scripts that can be used to synthesize, place, and route the design in a way specific to their design libraries and process.

Delivering soft IP gives a lot of flexibility to the customer, including the ability to configure some features in the design such as external bus widths and cache sizes. However, there are limitations as well. Delivering soft IP

means that the design will need to be fully synthesized and not make use of any of the advanced implementation techniques that are commonly used in the development of high-performance processors.

For Cortex-A8 to hit the top performance targets desired by its customers, support for advanced implementation techniques would be required. At the same time, Cortex-A8 would need to still support the traditional soft IP delivery method for customers for whom time to market and flexibility were more important than achieving top performance. Customers who wish to quickly implement and deploy the processor with a small team can do so using the soft IP flow. However, customers who wish to target higher performance levels can do so as well by using a larger team to build portions of the design using an advanced implementation flow. Designs implemented using both the soft and advanced flows are logically identical and share the same RTL code base. In this way, the requirements of the full customer base can be met with a single design. To help minimize the cost of building an advanced implementation, ARM provides a reference design to customers who want to take advantage of the advanced implementation flow. This reference design minimizes the effort required from the customer to only the activities that are unique to their specific process. Also, to keep the cost of the advanced implementation as low as possible, the advanced techniques are used strategically only in the areas that provide the most benefit. A large majority of the design can still be implemented using a standard synthesis flow and the additional benefits from using the advanced implementation will still be realized.

3.8 Conclusion

This chapter has gone through the details of the Cortex-A8 microarchitecture describing the design along with some of the background behind the decisions made along the way. The end result is a processor design that is uniquely placed in the market because of the fine balance it achieves between high performance and low power operation.

Acknowledgments

Writing this chapter has not been a singular effort and has involved the help of many others involved in the design and development of the Cortex-A8 processor. Specifically, I would like to thank Stephen Hill, Gerard Williams, Glen Harris, Matt Elwood, Ann Chin, Barry Williamson, and James Hardage for their input and help in the development of this chapter.

4

A Rotated Array Clustered Extended Hypercube Processor: The RACE-H™ Processor

Gerald G. Pechanek
Lightning Hawk Consulting Inc. and Priest & Goldstein, PLLC

Mihailo Stojancic
ViCore Technologies Inc.

Frank Barry
Onward Communications Inc. and Appalachian State University

Nikos Pitsianis
Duke University

CONTENTS

4.1 Introduction

Many products require efficient high-performance processing to meet the growing computational requirements of numerous media applications. Tailoring a processor's architecture and system interfaces to minimize processing overhead and inefficiencies in data transfers is required to provide efficient processing for these media applications. In addition, efficiency and low power require the use of techniques that remove the dependency on the

processor clock speed to obtain adequate performance. Applications such as high-definition (HD) multistandard video processing require almost continuous processing at the highest performance level. Because power use is highly dependent on frequency, very little power savings can be achieved in these high-compute applications by varying clock frequencies to minimize power use during less-demanding program segments. Use of a flexible parallel architecture is an important means to provide high performance without having a high dependence on the clock rate.

As an example, the high-profile H.264/AVC video encoding standard for picture sizes of 1920×1080 and larger represents one of the most computationally intensive algorithms to be implemented in future commercial and consumer products. The performance requirements greatly exceed the capabilities of current generation multigigahertz general-purpose processors. In addition, the video encoding standards require a great deal of flexibility to support the numerous coding tools, such as the discrete cosine transform, support for adaptive block sizes, intraspatial prediction, intertemporal prediction, support for interlaced coding and lossless representation, and deblocking filtering, to name only a few [1]. This flexibility requirement imposes programmable capability in the encoding hardware. To address these video requirements and provide an alternative to obtaining performance from a high-speed clock, a processor requires a highly flexible approach to parallel processing. The RACE-Hypercube™ processor, running at relatively low clock frequencies, provides multiple forms of selectable parallelism and an architecture that is moldable to an application.

To meet the performance and flexibility requirements in a cost-effective manner the RACE-H™ processor provides a hybrid architecture consisting of a scalable array processor enhanced with a scalable array of application-specific hardware-assist coprocessing units. To make such a hybrid processor widely available at low cost requires the use of standard design practices that allow the design to be fabricated at multiple semiconductor suppliers. This means that custom-designed SOCs, optimized to a particular manufacturing process, cannot be easily used. Consequently, the standard approach of increasing clock speed on an existing design in an attempt to meet higher performance requirements is not feasible.

The need to support multiple standards, such as MPEG-2, MPEG-4, and VC-1 standards as SMPTE 421M, and to quickly adapt to changing standards, has become a product requirement [2]. To satisfy this need, programmable DSPs and control processors are being increasingly used as the central SOC design component. These processors form the basis of the SOC product platform and permeate the overall system design including the on-chip memory, DMA, internal buses, and the like. Consequently, choosing a flexible and efficient processor, which can be manufactured by multiple semiconductor suppliers, is arguably the most important intellectual property (IP) decision to be made in the creation of an SOC product.

In recent years, a class of high-performance programmable processor IP has emerged that is appropriate for use in high-volume embedded applications

such as digital cellular, networking, communications, video, and console gaming [3,4,5]. This chapter briefly describes the RACE-H™ processor as an example of the architectural features needed to meet the highest demands of the H.264.AVC high-profile video encoding requirements. The next section provides a brief description of the RACE-H™ architecture. Section 4.2, titled "The RACE-H™ Processor Platform," describes how the RACE-H™ architecture fulfills system requirements, with a focus on the DMA subsystem and development tools. The "Video Encoding Hardware Assists" section briefly discusses examples for processor element hardware assists. The "Performance Evaluation" section presents performance results and projections and Section 4.6 concludes the chapter.

4.2 The RACE-H™ Architecture

In numerous application environments, there is a need to significantly augment the signal-processing capabilities of a MIPS, ARM, or other host processor. The RACE-H™ processors provide streamlined coprocessor attachment to MIPS, ARM, or other hosts for this purpose. The RACE-H™ architecture offers multiple forms of parallelism that are selectable at each stage of development, from SOC definition through software programming. Through selectable parallelism, the RACE-H™ processors achieve high performance at relatively low clock rates, thereby minimizing power use.

These forms of parallelism include 14 degrees of parallelism that may be selected. The first degree concerns the number of very long instructional word (VLIW) slots that are executed. The total number of slots available for execution is determined at implementation time where up to eight instruction slots may be implemented in the RACE-H™ architecture. The number of slots that may be executed up to the maximum number implemented can then be selected and vary instruction by instruction during program execution. The second degree includes the determination of supported application-specific instructions and the selection of the appropriate instructions for algorithmic operations. The third degree concerns the number and type of selectable application-specific hardware assists implemented in each processing element (PE), allowing program control over the enabling and operation of each hardware assist. By attaching hardware assists to each PE, the hardware assist capability of the core scales with the RACE-Hypercube dimensions. The fourth degree is determining the number of PEs to implement and for specific algorithmic use to mask the PE array as needed for optimum power and performance efficiency. The fifth degree concerns operating the core as a single-issue uniprocessor for control single-thread operation. The sixth degree concerns operating the core as a variable-length indirect VLIW (iVLIW) uniprocessor for increased performance in uniprocessor tasks. The seventh degree supports operating each PE as a single-issue PE for all enabled PEs. The eighth

degree supports operating each PE as a variable-length iVLIW issue PE for all enabled PEs. The ninth degree allows the selection of 32-bit packed data operations, including single 32-bit, dual 16-bit, and quad 8-bit operations which can be mixed on each instruction slot in a VLIW. The tenth degree allows the selection of 64-bit packed data operations, including single 64-bit, dual 32-bit, quad 16-bit, and octal 8-bit operations that can be mixed with each other and with packed 32-bit data operations on each instruction slot in a VLIW. The eleventh degree concerns conditionally executing instructions independently within a VLIW and independently within each PE. The twelfth degree supports the independent selection of mesh, torus, hypercube, and hypercube-complement PE-to-PE communications concurrently on the array with DMA, load, store, and the other execution unit operations. The thirteenth degree allows for independent threaded array operations across the number of implemented and enabled PEs to be controlled by a single controller and the fourteenth degree supports background DMA operations that may be scaled across the number of DMA lanes implemented to match the dimensions of the RACE-Hypercube.

To provide for these multiple degrees of freedom, the RACE-H™ processor is organized as an array processor using a sequence processor (SP) array controller and an array of distributed indirect VLIW PEs. The SP and each PE is provided with a small VLIW memory (VIM) that stores program-loaded VLIWs, where the VLIWs may be indirectly selected for execution. By varying the number of PEs on a core, an embedded scalable design is achieved with each core using a single architecture. This embedded scalability makes it possible to develop multiple products that provide a linear increase in performance and maintain the same programming model by merely adding array processor elements as needed by the application. As the processing capability is increased, the PE memory interface bandwidth is increased, and the system DMA bandwidth may be increased accordingly. Embedded scalability drastically reduces development costs for future products because it allows for a single software development kit (SDK) to support a wide range of products.

In addition to the embedded scalability, RACE-H™ processors are configurable in hardware organization and in software use of on-chip resources. For example, the processor hardware may be configured in the number and type of processor cores included on a chip, number of VLIW slots, supported instructions, the sizes of the SP's instruction memory, the distributed iVLIW memories, the PE/SP data memories, and the I/O buffers, selectable clock speed, choice of on-chip peripherals, and DMA bus bandwidth. The processor software may configure the use of the hardware dynamically instruction by instruction. Multiple RACE-H™ processors may also be included on a chip and organized for data pipeline multiprocessing between the multiple SP/PE-array processors. Within a RACE-H™ processor, the parallelization of subapplication tasks may also use forms of thread parallelism with a centralized host-based control, described later in this chapter.

Figure 4.1 shows the major architectural elements that make up a typical RACE-H™ processor. The RACE-H™ processor combines PEs in clusters that

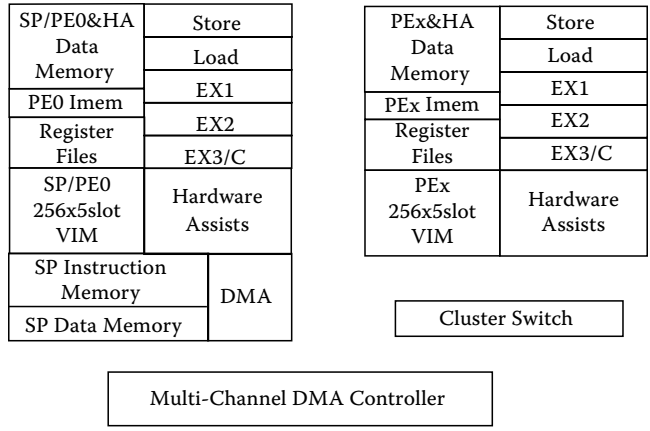


FIGURE 4.1
RACE-H processor architecture elements.

also contain a sequence processor (SP), uniquely merged into the PE array, and a cluster-switch. The SP provides program control, contains instruction and data address generation units, and dispatches instructions fetched from the SP instruction memory to the PEs in the array. Both the SP and PEs each include a common set of execution units using the same indirect VLIW architecture for all processing elements. Figure 4.1 illustrates a five-issue variable length VLIW organization where the instruction set is partitioned into store, load, execute 1 (EX1), execute 2 (EX2), and execute 3 and communicate (EX3/C) instruction slots. The architecture is easily expandable to support the addition of another load unit and additional execution units.

The RACE-H™ processor is designed for scalability with a single architecture definition and a common toolset. The processor and supporting tools are designed to optimize the needs of a SOC platform by allowing a designer to balance an application's sequential control requirements with the application's inherent data parallelism. This is accomplished by having a scalable architecture that begins with a simple uniprocessor model, as used on the SP, and continues through multi-array processor implementations. The RACE-H™ architecture supports a reasonably large array processor, as well as a simple stand-alone uniprocessor, the SP. In more detail, a RACE-$H_{2\times2}$™, RACE-$H_{4\times4}$™, and RACE-$H_{4\times4\times4}$™, are shown in Figure 4.2. The RACE-H™ architecture allows organizations with multiple SPs where each SP controls a subcluster of PEs. For example, a 4×4 may be made up of four 2×2 clusters each with their own SP or a 4×4 may be configured with only a single SP controlling the 16 PEs. In a similar manner, a $4 \times 4 \times 4$ 3D-cube may have multiple configurations of SPs and PEs depending upon a product's requirement.

The RACE-H™ architecture uses a distributed register file model where the SP and each PE contain their own independent compute register file (CRF),

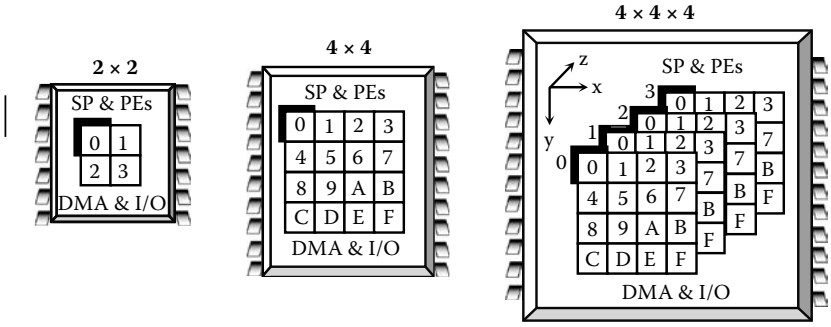


FIGURE 4.2
RACE-H_{2x2}, RACE-H_{4x4}, and RACE-H_{4x4x4} scalable products.

up to eight execution units (five shown), a distributed very long instruction word memory (VIM), local SP instruction memory, local PE instruction memory, local data memories, and an application-optimized DMA and bus I/O control unit. The CRF is reconfigurable dynamically to act as a 32 × 32-bit or 16 × 64-bit register file instruction by instruction and can vary within a VLIW and is totally integrated into the instruction set architecture. An 8 × 32-bit address register file (ARF) and a 24 × 32-bit miscellaneous register file (MRF) are also defined in the instruction set architecture. Extending to support 128-bit wide and larger width register files is also architecturally supported. The balanced architectural approach taken for the CRF provides the high- performance features needed by many applications. It supports octal byte and quad halfword operations in a logical 16 × 64-bit register file space without sacrificing the 32-bit data-type support in the logical 32 × 32-bit register file. Providing both forms of packed data independently on each execution unit allows optimum usage of the register file space and minimum overhead in manipulating packed data items.

In the RACE-H™ architecture, the address registers are separated from the compute register file. This approach maximizes the number of registers for compute operations and guarantees a minimum number of dedicated address registers. It does not require any additional ports from the compute register file to support the load and store address generation functions and it still allows independent PE memory addressing for such functions as local data dependent table lookups.

The RACE-H™ instruction set is partitioned into four groups using the high two bits of the instruction format: a control group, an arithmetic group, a load–store group, and a reserved proprietary instruction group. Figure 4.3 shows 32-bit simplex instructions in groupings that represent an example of five execution unit slots of a VLIW plus a control group (01). Up to eight execution slots are architecturally defined allowing for additional load and execution units. Group (00) is reserved for future use. The execution units include store and load units and a set of execution units. For example, a first

SP	Five Slot VLIW Instruction Types				
Control	**Store**	**Load**	**EX1**	**EX2**	**EX3/C**
Group 01	Group 10	Group 10	Group 11	Group 11	Group 11
Call	Base+Disp.	Base+Disp.	ADD/SUB	ADD/SUB	Copy
Jump	Direct/Indirect	Direct/Indirect	Butterfly	Butterfly	Shift/Rotate
Loops	Table	Table	Compare	MPY/MPYA	Permute
Return	ARF group	ARF group	AbsoluteDiff	MPYCmplx	Bit Operations
Load VLIW		Immediate	Min/Max	MPYCmplxA	SP Send/Rcv
Execute VLIW		Broadcast	Logicals	SUM2P/SUM2PA	PE to PE Exchange
		TCM			TCM

FIGURE 4.3
RACE-H instructions.

execution unit is EX1, supporting arithmetic instructions. A second execution unit is EX2, supporting, for example, multiply, multiply accumulate, and other arithmetic instructions. A third execution unit is EX3/C, supporting data manipulation instructions such as shift, rotate, permute, bit operations, various other arithmetic instructions, PE-to-PE communication, and hardware-assist interfacing instructions.

Providing a common set of execution units is also supported by the architecture. The load and store instructions support base plus displacement, direct, indirect, and table addressing modes. In addition, many application-specific instructions are used for improved signal-processing efficiency. An example of these instructions is the set of multiply complex instructions for improved FFT performance described in reference [6]. In addition, the load unit and EX3/C unit have tightly coupled machine (TCM) instructions which are used to control hardware-assist coprocessors attached to each PE. Any of the five slots of instructions can be selected on a cycle-by-cycle basis, for single-, two-, three-, four-, or five-issue VLIWs, in this particular five-issue VLIW example. The single RACE-H™ instruction set architecture supports the entire RACE-H™ family of cores from the single merged SP/PE0 1×1 to any of the highly parallel multiprocessor arrays (1×2, 2×2, 2×4, 4×4, $4 \times 4 \times 4$, …); for more details see reference [7].

The control and branch instructions are executed by the SP and in the PEs, when the PEs are operating independently. The SP and PEs are also capable of indirectly executing VLIWs that are local to the SP and in each PE. Note that VLIWs are stored locally in VLIW memories (VIMs) in each PE and in the SP and are fetched by a 32-bit execute VLIW (XV) instruction. To minimize the effects of branch latencies, a short variable pipeline is used consisting of Fetch, Decode, Execute, and ConditionReturn for non-iVLIWs and Fetch, PreDecode, Decode, Execute, and ConditionReturn for iVLIWs. The PreDecode pipeline stage is used to indirectly fetch VLIWs from their

local VIMs. It is noted that an XV instruction supplies an offset address to a local VIM control unit, which computes a VIM address based on a local VIM base address register. In addition, the VLIWs located at the same VIM address in different PEs do not have to be the same VLIW. These architectural features allow for independent program action at each PE, referred to as synchronous multiple instruction multiple data (SMIMD) operation. In addition, an extensive scalable conditional execution approach is used locally in each PE and the SP to minimize the use of branches.

All loads–stores and arithmetic instructions execute in one or two cycles with no hardware interlocks. The TCM instructions initiate multicycle operations that execute independently of other PE instructions. Furthermore, all arithmetic, load–store, and TCM instructions can be combined into VLIWs, stored locally in the SP and in each PE, and can be indirectly selected for execution from the small distributed VLIW memories. In one of the supported architectural approaches, a Load iVLIW (LV) instruction is used by the programmer or compiler to load individual instruction slots making up a VLIW with the 32-bit simplex instructions optimized for the algorithm being programmed. These VLIWs are used for algorithm performance optimization, are reloadable, and require only the use of 32-bit execute VLIW (XV) instructions in the program stored in the SP instruction memory.

Many algorithms require an additional level of independent operations at each PE. The RACE-H™ architecture supports independent and scalable program thread operations on each PE. Figure 4.4 illustrates a scalable thread flowchart of independent and scalable thread operations for the RACE-$H_{4\times4}$™. The SP controls the thread operation by issuing a thread start (Tstart) instruction. The Tstart instruction is fetched from SP instruction memory and dispatched to the SP and all PEs. Based on the Tstart, each PE

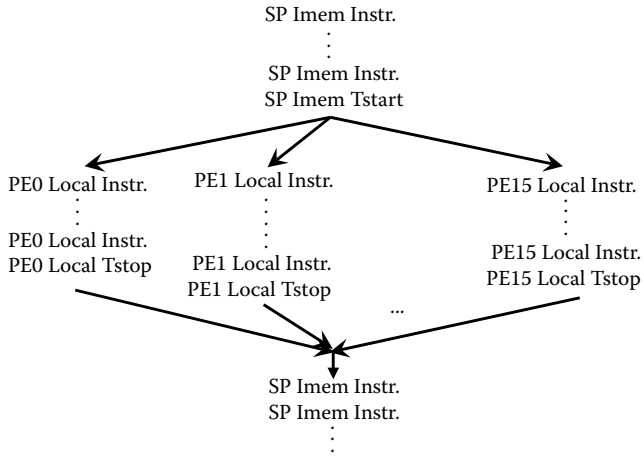


FIGURE 4.4
4 × 4 independent PE threads.

switches to local independent PE operations fetching PE local instructions from PE instruction memory. The PE instruction memory may store all types of PE instructions including PE branch instructions and a thread stop (Tstop) instruction. As shown in Figure 4.4, each PE operates independently until its operations are complete, at which point each PE fetches a Tstop instruction. A Tstop instruction causes the PE to stop fetching local PE instructions and then to wait for SP dispatched instructions. Once all PEs have completed their local independent operations, the SP continues with its fetching operation from the SP instruction memory and dispatches PE instructions to the array. In SIMD operations, a dedicated bit in all instruction formats controls whether an instruction is executed in parallel across the array of PEs or sequentially in the SP.

To more optimally support the PE array containing the distributed register files, the RACE-H™ network is integrated into the architecture providing single-cycle data transfers within PE clusters and between orthogonal clusters of PEs. The EX3/C communication instructions can also be included into VLIWs, thereby overlapping communications with computation operations, which in effect reduces the communication latency to zero. The RACE-H™ network operation is independent of background DMA operations which provide a data streaming path to peripherals and global memory.

The inherent scalability of the RACE-H™ processor is obtained in part through the advanced RACE-Hypercube™ network which interconnects the PEs. Consider, by way of example, a two-dimension (2D) 4×4 torus and the corresponding embedded 4D hypercube, written as a 4×4 table with row, column, and hypercube node labels. (See Figure 4.5A.)

In Figure 4.5A, the $PE_{i,j}$ cluster nodes are labeled in Gray-code as follows: $PE_{G(i),G(j)}$ where G(x) is the Gray code of x. The array of Figure 4.5A is transformed by a series of operations that rotate columns of PEs. First, columns 2, 3, and 4 are rotated one position down. Next, the same rotation is repeated with columns 3 and 4 and then, in a third rotation, only column 4 is rotated. The resulting 4×4 table is shown in Figure 4.5B.

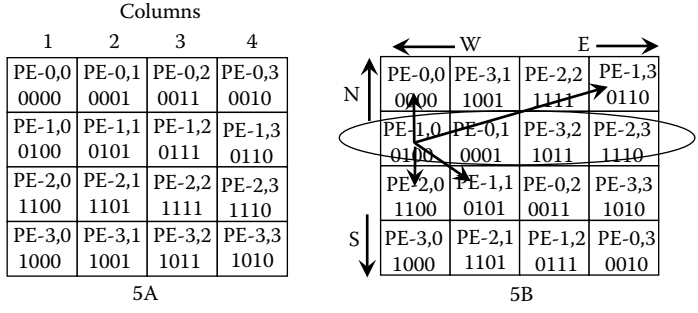


FIGURE 4.5
Hypercube RACE-H network.

Notice that the row elements in Figure 4.5B, for example {(1,0), (0,1), (3,2), (2,3)}, contain the transpose PE elements. By grouping the row elements in clusters of four PEs each, and completely interconnecting the four PEs in a cluster, connectivity among the transpose elements is obtained and extends the connectivity beyond the connectivity provided by a hypercube. In the new matrix of PEs, the east and south wires, as well as the north and west wires, are connected between adjacent clusters. For example, using Figure 4.5A note that node (2,3) connects to the east node (2,0) with wraparound wires in a torus arrangement. Node (2,3) also connects to the south node (3,3). Now, using Figure 4.5B, note that nodes (2,0) and (3,3) are both in the same row cluster adjacent to the row cluster containing node (2,3). This means that the east and south wires can be shared and, in a similar manner, the west and north wires can be shared between all clusters. This same pattern occurs for all nodes in the transformed matrix. This effectively cuts the wiring in half as compared to a standard torus, and without affecting the performance of any SIMD array algorithm.

Because the rotating algorithm maintains the connectivity between the PEs, the normal hypercube connections remain. For example, in Figure 4.5B, PE (1,0/0100) can communicate to its nearest hypercube nodes {(0000), (0101), (0110), (1100)} in a single step. With the additional connectivity in the clusters of PEs, the longest paths in a hypercube, where each bit in the node address changes between two nodes, are all contained in the completely connected clusters of processor nodes. For example, the circled cluster contains node pairs [(0100), (1011)] and [(0001), (1110)] which would take four steps to communicate between each pair in previous hypercube processors, but takes only one step to communicate in the RACE-H™ network. These properties are maintained in higher-dimensional RACE-H™ networks containing higher-dimensional tori, and thus hypercubes, as subsets of the RACE-H™ connectivity matrix. The complexity of the RACE-H™ network is small and the diameter, the largest distance between any pair of nodes, is two for all d where d is the dimension of the subset hypercube [8]. The distance between mesh, torus, hypercube, and hypercube-complement nodes is one.

4.3 The RACE-H™ Processor Platform

The RACE-H™ processors are designed to act as independent processors that may attach to ARM, MIPS, or other hosts. The programmer's view of a RACE-H™ multiprocessor is a shared memory sequentially coherent model where multiple processors operate on independent processes. With this model, an SOC developer can quickly utilize the signal-processing capabilities of the RACE-H™ core subsystems because the operating system already runs on the host processors. In its role as an attached coprocessor, the RACE-H™ core is subservient to the host processor. A core driver running on the host operating system manages all the RACE-Hypercube processor resources on the

core. A RACE-H™ system interface allows multiple RACE-H™ cores to be attached to a single-host processor.

To complement the configurable hardware, there is a RACE-H™ library of both DSP and control software routines. In addition, existing host-optimized compilers may be used for the sequential code that remains resident on the host allowing the parallel code to be optimized for the RACE-H™ cores.

Data and control communication between thread coprocessors and system components (such as a host control processor, memories, and I/O) is carried out using a DMA subsystem, one or more system data buses (SDBs), and a system control bus (SCB). The SDB provides the high bandwidth data interface between the cores and other system peripherals including system memory. The SDB consists of multiple identical lanes, and is scalable by increasing the number of lanes or increasing the width of the lanes. The SCB is a low-latency coprocessor-to-coprocessor/peripheral messaging bus that runs independently and in parallel with the SDB. This system of multiple independent application task-optimized cores is designed to have each core run an independent system-level thread supported by the programmable DMA engines.

The DMA subsystem consists of two or more transfer controllers, the DMA bus, SDB, and SCB interfaces. Each transfer controller manages data movement between the SDB and coprocessor memories, one direction at a time, across a single data "lane" of the DMA bus. Transfer controllers operate independently of each other, fetching their own transfer instructions from core memories once initiated by either the SP or the host control processor. Transfer signaling instructions and hardware semaphores may be used to synchronize data transfer with array processing and host processor activities. Figure 4.6 shows

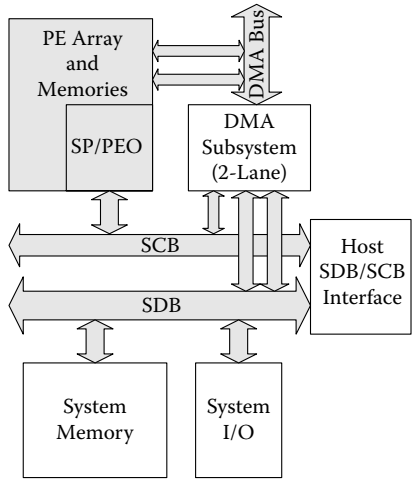


FIGURE 4.6
Two-lane DMA subsystem.

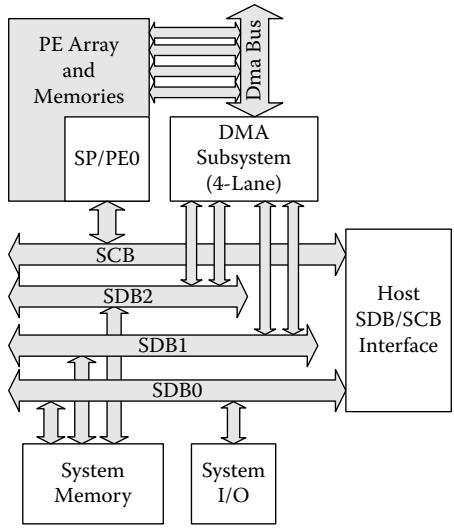


FIGURE 4.7
Four-lane DMA subsystem with two SDB-memory buses.

a configuration with a two-lane (two transfer controllers) DMA subsystem, and Figure 4.7 shows a four-lane DMA subsystem. The transfer controllers support instructions that allow independent addressing modes for the SDB side and DMA bus side. For example, in an image encoding application a group of picture macroblocks may be read from system memory using a "stride" addressing mode and distributed to array memories in various patterns using only two transfer instructions.

Each SDB is a high-bandwidth bus based on the AMBA 2.0 standard [9] and scalable in width and maximum burst length. It is used primarily for data flow between array coprocessor memories and system memories or I/O. In a multi-SDB configuration (used for very high data bandwidth), the DMA transfer controller and host interface connections will be allocated between buses to match performance requirements.

The SCB is a low-latency control bus used for communication between the host control processor, the SP, and the DMA subsystem. It is used for enabling and configuring runtime system options and also allows the SP to configure and control the DMA transfer controllers.

To streamline development, verification, and debugging, a range of modeling and prototyping platforms support system modeling, and software and hardware system development, including:

- A cycle-accurate C-simulator, which can be used to develop RACE-H™ processor software. This can be used directly with other C simulations, or with control processor tools and bus models to provide a software simulation model of an entire system.

- A software development toolkit (SDK) consisting of the GNU GCC compiler and binary tools, the RACE-H™ ANSI-C Parallel C compiler, a back-end compiler to enable high-level assembly programming with register allocation, instruction scheduling, and packing of instructions into VLIWs, and VIMA [10], a whole-program scheduler of iVLIW instructions.
- An emulation board, can be used to model RTL of an entire SOC system.
- A debugger.
- An assortment of software libraries including a real-time framework, user application program interface, debug I/O, C program runtime, mathematical libraries, and specialized DSP libraries.

The RACE-H™ debugging GUI is shared by the C-simulator and emulation board. With this development environment the software can be integrated and tested. Verification tools and supporting scripts and guidelines for the physical design are available.

4.4 Video Encoding Hardware Assists

A number of algorithmic capabilities are generally common between multiple video encoding standards, such as MPEG-2, H.264/AVC, and SMPTE-VC-1. Motion estimation/compensation and deblocking filtering are two examples of compute-intensive algorithms that are required for video encoding.

Motion estimation is computationally the most expensive part of a video encoding process. On average it can take about 60–80% of the total available computational time, thus having the highest impact on the speed of the overall encoding process. It also has a major impact on the visual quality of encoded video sequences.

The most common motion estimation algorithms are block-matching algorithms operating in the time domain. Here motion vectors are used to describe the best temporal prediction for a current block of pixels to be encoded. A time domain prediction error between the current block of pixels and the reference block of pixels is formed, and a search is performed to minimize this value. The best motion vector minimizes a cost function based on the prediction block distance and the block pixel difference.

A block of pixels of the current video frame, which is in the search range of a passed frame (in temporal order), is compared with all possible spatial positions within the search range, looking for the smallest possible difference. For the best matching reference block, a motion vector is derived which describes the relative position of the two blocks.

Multiple different criteria have been proposed for the best match evaluation. They are of different complexity and efficiency in terms of finding the

global optimum over a given search area. The sum of absolute differences (SAD) is the most commonly used criterion for the best match evaluation.

A hardware assist (HA) for block-matching search may be attached to each PE and is capable of performing full search (within a search window of programmable size) for integer pixel motion vectors calculation. It is capable of simultaneous extraction of results for 16×16, 16×8, 8×16, 8×8, and 4×4 motion search based on the SAD criterion for each particular block size and given search range. The search range window may vary. For example, a search range such as 64×64 or 128×96 may be used. The hardware assist is also capable of setting up a coarse hierarchical search (through use of a special TCM instruction) by automatically decimating pixels of a larger search range (64×64, for example) and bringing the decimated 32×32 search area into the pipelined compute engines of the hardware assist. Partial search results (SAD for each current block position within the search range) may be stored locally in each PE for further processing, or stored in the local HA/PE memory.

A similar hardware assist for deblocking filtering may be attached to each PE of the RACE-H™ processor, providing for a major offloading of the most compute-intensive operations, and allowing for real-time full-feature HD video encoding.

4.5 Performance Evaluation

To illustrate the power of the highly parallel RACE-H™ architecture, a simple example is presented: Two vectors are to be added and the result stored in a third vector.

For ($I = 0$; $I < 256$; I ++)
$A[i] = B[i] + C[i]$.

In a sequential-only implementation, there would be required a loop of four instructions, two load instructions to move a B and a C element to registers, an add of the elements, and a store of the result to register A. The sequential implementation takes (4*256) iterations = 1024 cycles, assuming single-cycle load, add, and store instructions.

Assuming the data type is 16-bits and quad 16-bit packed data instructions are available in the processor, the vector sum could require (4*64) iterations = 256 cycles.

Furthermore, assuming an array processor of four PEs where each PE is capable of the packed data operations, then the function can be partitioned between the four PEs and run in parallel requiring (4*16) iterations = 64 cycles.

Finally, assuming a VLIW processor such as the 2×2 RACE-Hypercube processor, a software pipeline technique can be used with the VLIWs to minimize the

instructions issued per iteration such that (2*16) iterations = 32 cycles are required. This represents a 32× improvement over the sequential implementation.

With use of the 14 different levels of parallelism available on each core, the benchmarks shown in Figure 4.8 can be obtained on a RACE-Hypercube processor, with the array size shown in parentheses. (The 4 × 4 numbers are extrapolated from coded 2 × 2 numbers.)

The RACE-H™ architecture allows a programmer or compiler to select the level of parallelism appropriate for the task at hand. This selectable parallelism includes packed 32-bit and 64-bit data operations. With each additional PE, the packed data support on a processor core grows such that a 2 × 2 effectively provides four PEs each with five 64-bit execution units providing 4*5*64 bits = 1280-bits/cycle of packed data support. A 4 × 4 provides four times this for 4*1280 bits = 5120 bits/cycle (640 bytes/cycle) of packed data support which at 250 MHz provides a performance of 160 gigabytes/sec. A 4 × 4 × 4 3D cube effectively provides four times this for 4*5120 bits = 20,480 bits/cycle (2560 bytes/cycle) of packed data support which at 250 MHz provides a performance of 640 gigabytes/sec. With at least three 64-bit hardware-assist functions operating independently and in parallel in each PE, an additional 3*(8 bytes/cycle)*64 PEs*250 MHz = 384 gigabytes/sec of performance. The 4 × 4 × 4 3D cube provides 1.024 trillion bytes/sec at a relatively low clock

Benchmark	Data Type	Performance
256 pt. Complex FFT (4×4)	16-bit real & imaginary	189 cycles
256 pt. Complex FFT (2×2)	16-bit real & imaginary	383 cycles
256 pt. Complex FFT (1×1)	16-bit real & imaginary	1115 cycles
1024 pt. Complex FFT (4×4)	16-bit real & imaginary	580 cycles
1024 pt. Complex FFT (2×2)	16-bit real & imaginary	1513 cycles
1024 pt. Complex FFT (1×1)	16-bit real & imaginary	5221 cycles
2048 pt. Complex FFT (4×4)	16-bit real & imaginary	1350 cycles
2048 pt. Complex FFT (2×2)	16-bit real & imaginary	3182 cycles
2D 8×8 IEEE IDCT (4×4)	8-bit	18 cycles
2D 8×8 IEEE IDCT [11] (2×2)	8-bit	34 cycles
2D 8×8 IEEE IDCT (1×1)	8-bit	176 cycles
256 tap Real FIR filter, M samples (4×4)	16-bit	4*M + 86 cycles
256 tap Real FIR filter, M samples (2×2)	16-bit	16*M + 81 cycles
256 tap Real FIR filter, M samples (1×1)	16-bit	64*M + 78 cycles
4×4 Matrix * 4x1 vector (4×4)	IEEE 754 Floating Point	2-cycles/4-output vector
4×4 Matrix * 4x1 vector (2×2)	IEEE 754 Floating Point	2-cycles/output vector
3×3 Correlation (720col) (4×4)	8-bit	145 cycles
3×3 Correlation (720col) (2×2)	8-bit	271 cycles
3×3 Median Filter (720col) (4×4)	8-bit	360 cycles
3×3 Median Filter (720col) (2×2)	8-bit	926 cycles
Horizontal Wavelet (N Rows = 512) (4×4)	16-bit	370 cycles
Horizontal Wavelet (N Rows = 512) (2×2)	16-bit	1029 cycles

FIGURE 4.8
1 × 1, 2 × 2, and 4 × 4 RACE-Hypercube processor benchmarks.

frequency of 250 MHz with short execution unit pipelines based on 64-bit data types and an architecture that is programmer friendly.

4.6 Conclusions and Future Extensions

The pervasive use of processor IP in embedded SOC products for consumer applications requires a stable design point based on a scalable processor architecture to support future needs with a complete set of hardware and software development tools. The RACE-H™ cores are highly scalable, using a single architecture definition that provides low power and high performance. Target SOC designs can be optimized to a product by choice of core type, 1×1, 1×2, 2×2, 4×4, $4 \times 4 \times 4$, and by the number of cores. The tools and SOC development process provides a fast path to delivering verified SOC products. Future plans include architectural extensions, such as improved VIM loading techniques, extensions to 128-bit datapaths effectively doubling performance, programmable hardware-assist engines, and other extensions representing supersets of the present design that would further improve performance in intended applications.

Trademark Information

References

[1] G. J. Sullivan, P. Topiwala, and A. Luthra, "The H.264/AVC Advanced Video Coding Standard: Overview and Introduction to the Fidelity Range Extensions," SPIE Conference on Applications of Digital Image Processing XXVII, Special Session on Advances in the New Emerging Standard: H.264/AVC, August, 2004.

[2] Alan Gatherer et al., "DSP-Based Architectures for Mobile Communications: Past, Present, and Future," *IEEE Communications Magazine*, pp. 84–90, January, 2000.

[3] Krishna Yarlagadda, "The Expanding World of DSPs," *Computer Design*, pp. 77–89, March, 1998.

[4] Ichiro Kuroda, and Takao Nishitani, "Multimedia Processors," *Proceedings of the IEEE*, Vol. 86, No. 6, pp. 1203–1227, June, 1998.

[5] Jan-Willem van de Waerdt et al., "The TM3270 Media-Processor," *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (Micro'05)*, pp. 331–342, 2005.

[6] Nikos P. Pitsianis and Gerald G. Pechanek, "High-Performance FFT Implementation on the BOPS ManArray Parallel DSP," *Advanced Signal Processing Algorithms, Architectures and Implementations IX*, Volume 3807, pp. 164–171, SPIE International Symposium, Denver, CO, July 1999.

[7] Gerald G. Pechanek and Stamatis Vassiliadis, "The ManArray Embedded Processor Architecture," *Proceedings of the 26th Euromicro Conference: "Informatics: Inventing the Future,"* Maastricht, The Netherlands, September 5–7, 2000, Vol. I, pp. 348–355.

[8] Gerald G. Pechanek, Stamatis Vassiliadis, and Nikos P. Pitsianis, "ManArray Interconnection Network: An Introduction," *Proceedings of EuroPar '99 Parallel Processing*, LNCS 1685, pp. 761–765, Toulouse, France, Aug. 31–Sept. 3, 1999. www.arm.com/products/solutions/AMBA_Spec.html.

[9] Nikos P. Pitsianis and Gerald G. Pechanek, "Indirect VLIW Memory Allocation for the ManArray Multiprocessor DSP," *ACM SIGARCH Computer Architecture News*, Vol. 31, Issue 1, March 2003, pp. 69–74.

5

A High-Throughput Self-Timed FPGA Core Architecture

Brian C. Gaide and Lizy Kurian John
The University of Texas at Austin

CONTENTS

5.1 Introduction

Configurable logic devices have myriad applications. Since their advent in 1985 by Xilinx Corp. [2], FPGAs (Field Programmable Gate Arrays) have advanced significantly. Initially, FPGAs were used primarily for relatively simple applications such as ASIC (Application-Specific Integrated Circuit) prototyping and glue logic. Complex designs were not feasible simply because

the number of logic cells on one chip were on the order of a few hundred. Now, however, transistor density increases have allowed the number of logic cells on a single device to grow to tens of thousands, and speed has increased by orders of magnitude. With this kind of logic density, it is now possible to implement such designs as microprocessors, multimedia accelerators, and data routers. Additionally, it has become commonplace to see ASIC-like blocks, such as large embedded memory, DSPs, high-speed IOs, and processor blocks embedded into FPGA cores. Entire systems-on-a-chip can now be fabricated from one FPGA. Therefore the FPGA itself becomes a critical component in achieving high-speed timing constraints for the system.

Although technology shrinking has allowed for consistent area reductions, it has not come for free. Transistors have become velocity saturated. Interconnect parasitics have not scaled proportionately. Noise margins have dropped. Power density, especially DC (Direct Current) power density, is growing substantially, so much so that FPGA designers now have to optimize their designs to the triplicate of area, speed, and power.

Designing high-speed circuits in an FPGA is challenging, because any area or power increase that comes from a new circuit implementation in the core logic area gets multiplied by a factor of the array size. Thus, designs that may work in a microprocessor realm that allow a large performance gain, while incurring a large one-time area or power hit, are not feasible in an FPGA core. To make matters worse, unlike ASICs, the critical path cannot be predetermined and optimized in hardware, because the FPGA by nature of its programmability can have any number of critical paths.

Due to the increasing constraints of current CMOS technologies, and due to the fact that FPGAs have been optimized and reoptimized for 20 years now, it is the belief of the authors that any large-scale hardware performance increases in FPGAs will need a radical departure from our current circuit designing methodologies. One of these radical departures is asynchronous design.

5.2 Asynchronous and Synchronous Systems

What is asynchronous design? Before that question can be answered, we must first define the nature of synchronous design, so that we can differentiate between the two.

5.2.1 Defining Synchronous Design

Synchronous design revolves around the idea that operations performed inside a circuit are synchronized to a common reference signal. This reference signal is usually referred to as the clock, and it is this clock that is used to simultaneously update all state-holding elements in a circuit. The use of a common reference signal (in theory) reduces the complexity of a design. Assuming the exact time the transition of the clock signal is known, then the

design can go through all kinds of intermediate glitching as long as the data is valid some small window around when the clock pulse arrives. Because there are numerous possible paths from one source to one destination, and because the environment (process, temperature, voltage, noise variations) in which the signal propagates is not always known, it is difficult to determine precisely what the delay will be from that specific source to its destination. Instead of tightly controlling all of these variables, the designer simply assumes worst-case conditions, and assures that the data will be valid before the arrival of the clock pulse. Figure 5.1 illustrates a block-level diagram of a synchronous system, showing the global clock signal fanning out to multiple storage elements.

5.2.2 Defining Asynchronous Design

There are several types of design methods that are considered "asynchronous." Truly asynchronous designs use no synchronization signals at all. They rely solely on delays through logic paths to determine when the next wave of data will propagate through the system. This kind of asynchronous design is known as "wave pipelining" [25]. It is used very little (or not at all) in complex systems due to the fact that delays through a system can vary drastically based on environmental factors. Wave pipelining is especially impractical to use in a reconfigurable device because the delay on any given path will change depending on the placement and routing of the design in software.

A second type of asynchronous design is known as "self-timed." Self-timed systems use control signals to notify a logic block's computational status. These control signals are known as handshaking signals. The handshaking signals generally communicate either a data request (req) or data complete/acknowledge (ack) status signal (see Figure 5.2). Data is processed on an availability and need basis. As soon as one block is done processing data, it notifies the next block that it is finished and ready to transmit its data. As soon as the receiver block is done processing its data, it takes the data from the transmitting block and processes it. The basic idea is that data flow is dictated by local control signals instead of one global signal. Because self-timed design is the most common form of asynchronous design, the terms are often used interchangeably.

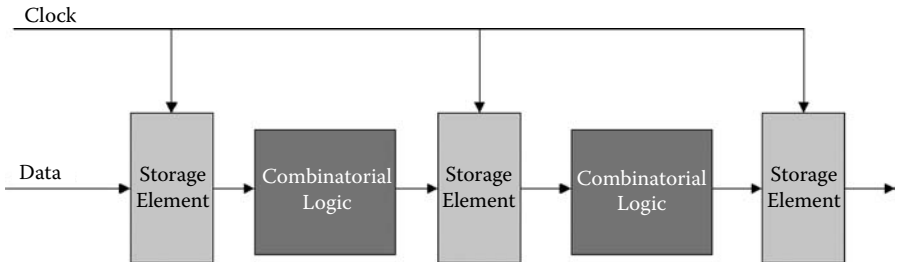


FIGURE 5.1
Block diagram of a synchronous design.

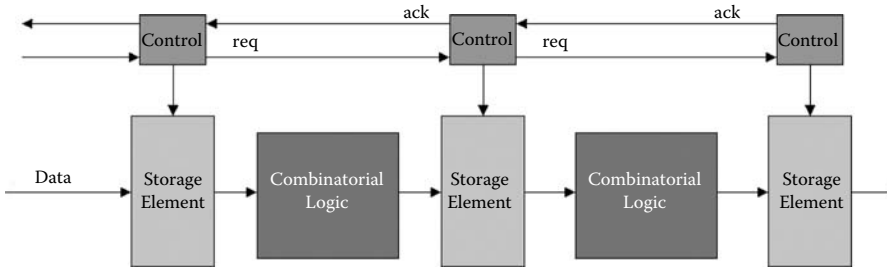


FIGURE 5.2
Block diagram of an asynchronous design.

There are a few subtypes of self-timed systems. The first is known as "delay-independent." Delay-independent circuits will output valid data regardless of the delays through the circuit. Although these circuits are extremely robust, they rarely exist in practice. Another subtype is known as "speed-insensitive." Speed-insensitive circuits hold as long as the delays on all routing wires are negligible. Because routing parasitics are nonnegligible in today's processes, often the control lines are delayed to allow the data to stabilize before the control signals appear at the destination block.

5.2.2.1 Problems with Asynchronous Design

From the discussion above on synchronous design, the reader may guess that synchronous designs allow "glitches" (intermittent logic transitions between states) to occur, but asynchronous designs do not. This is in fact largely true, and a major reason why synchronous design is much more common than asynchronous design. Inasmuch as there is no reference clock signal that has a known frequency, it is assumed that the time it takes a logic block to process data can vary greatly. Thus, any transition that occurs at the output of a logic block is determined to be an acknowledge signal that the block has finished its computation. Logic must be guaranteed by the designer to be hazard free. There are systematic methods for accomplishing this [5], but it is an extra level of design effort that the synchronous designer can usually avoid. In implementation methods where control and data signals are separate and the control signals are delayed relative to the data, glitching is allowed to occur in a self-timed system. However, as is explained shortly, the benefits of a self-timed system with these constraints are reduced.

Secondly, the area overhead of asynchronous control circuits is often nontrivial. For instance, the AMULET1* design team reported an estimated 20% area overhead for all of the asynchronous control circuits that went into their design [6]. However, the gap between synchronous and asynchronous

* The AMULET is an asynchronous version of the ARM5 processor.

overhead may narrow if recent trends continue in clock tree complexity. Typically, clock trees use multiple levels of large drivers to overcome the RC delays of the long routes they are driving. Additionally, large-scale synchronous designs employ PLLs, DLLs, a variety of clock gaters, and clock dividers in order to maintain effective clock distribution. All of these consume significant chip area, and none of them would be necessary on an asynchronous device.

The semiconductor arena has already largely been biased towards synchronous design. This translates to the abundant availability of CAD tools, software synthesizing algorithms, and research that is already behind the optimization of synchronous systems. There is comparatively little support for asynchronous systems at this time, so optimizing an asynchronous design is an uphill battle.

Lastly, there are some systems that are modeled more efficiently in a synchronous methodology. For multimedia applications, for instance, that require a constant throughput (say 30 frames per second of video to your computer monitor), it makes more sense to model this from a synchronous perspective.

5.2.2.2 Advantages of Asynchronous Design

One of the biggest advantages of asynchronous design is that system throughput is not based on the worst-case critical path. In a traditional synchronous circuit, one can only clock the system as fast as the slowest path in the system. Modern synchronous designs get around this fact by employing pipeline stages, where the critical path is broken up into pieces so that a higher clock frequency can be used. Pipelining has problems of its own, however. Overall latency increases because data in a pipeline will not be available at the output until N clock cycles later, where N is the number of pipeline stages. With each restart of the application, the pipeline will incur an N cycle penalty to refill. Also, the pipeline latches themselves add to the latency as well, due to their finite propagation delays and data setup time requirements. Even in pipelined designs, the designer still must take into account worst-case PVT (Process, Voltage, Temperature) conditions on the path between pipeline stages to guarantee functionality.

More often than not, in a synchronous system, a computation will finish with some amount of slack time left over before the clock pulse, simply because all PVT variations must be considered. More over, large-scale clocked systems always have some amount of clock skew and clock jitter that eats into every computation cycle. Not all paths through a block will be the worst-case path. Many computations finish well before the clock pulse arrives, but as long as one computation is slower than the rest, all of the other computations must wait until it is finished. Much of the available computation time is wasted in this sense.

Speed in an asynchronous system is based on average-case path delay. Asynchronous systems pass their data on to the next stage the moment a computation is finished, regardless of any environmental effects. The only

overhead that one pays when a computation is finished is in the delays through the handshaking logic. If one sequence of inputs causes a path to have a 1-ns delay, for instance, and another sequence of inputs causes the same path to have a 2-ns delay, a synchronous system would operate at 500 Mhz. However, assuming both paths are equally likely to be chosen, an asynchronous system could operate at an effective 667 Mhz.

Power consumption is a big issue with synchronous designs. In some devices, the clock tree alone can account for +40% of total chip power [7]. Elaborate clock-gating schemes are used in many modern architectures, but this costs area and often results in higher clock skew or latency. If fine-grained clock gating is not used, the chip will burn power even when portions of the chip are not processing data.

Asynchronous systems have a natural clock gating effect. Only the portions of the chip that are computing activate their handshaking signals, and the rest of the chip automatically stays dormant. Several commercial products have been developed using asynchronous design simply for the power savings it employs [8].

One of the largest headaches that the synchronous designer encounters is ensuring that his or her circuits function over all PVT variations. Although self-timed design does not guarantee PVT invariance, it does alleviate many of the race conditions found in its synchronous counterpart. Many practical asynchronous circuits have local race conditions of which the designer must be aware. However, using proper design techniques (such as synchronizing elements, eliminating hazards), these race conditions can be avoided or constrained to exist only in basic circuit elements. The synchronous designer, on the other hand, most often has to simulate his design to ensure that paths have adequate slack with respect to the clock.

A last benefit of asynchronous design is that it has a flatter and lower noise spectrum than a synchronous design. This is because the clock in a synchronous design switches a very large total chip capacitance at a very regular frequency, thus creating large peaks in the frequency response. Asynchronous circuits on the other hand have a more irregular signal switching pattern, and the capacitance that switches is at a more localized level. Consequently, the frequency response more closely resembles random noise, and at a lower overall level. Devices that require very low electromagnetic radiation signatures could benefit greatly from this. As an example, Figure 5.3 illustrates the spectral response of a microcontroller implemented both synchronously and asynchronously.

5.2.2.3 Asynchronous Design and FPGAs

The first and foremost advantage to creating an asynchronous FPGA lies in the nature of the FPGA's critical path. By design, an FPGA's critical path cannot be known if it is to have the ability to implement a large number of designs. Because of this, it is impossible to a priori optimize the hardware for a specific critical path, as is done on ASIC chips. Figure 5.4 shows an example of what a critical path might look like.

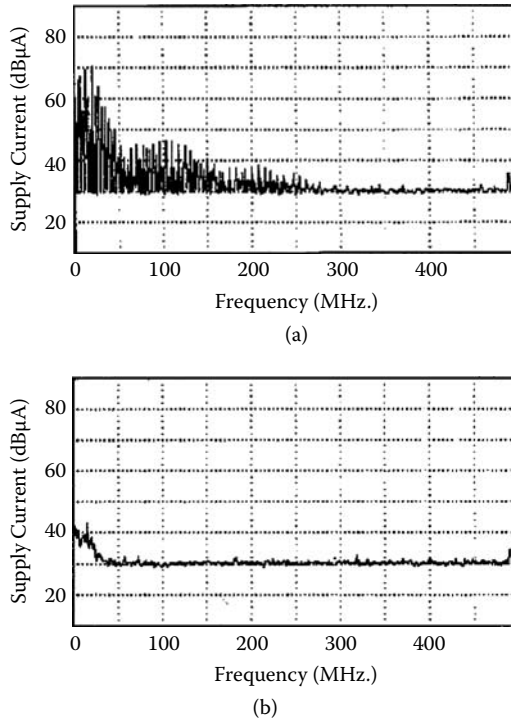


FIGURE 5.3
Frequency response of two 80C51 microcontrollers, synchronous implementation, and asynchronous implementation. (Courtesy of Phillips Semiconductors.)

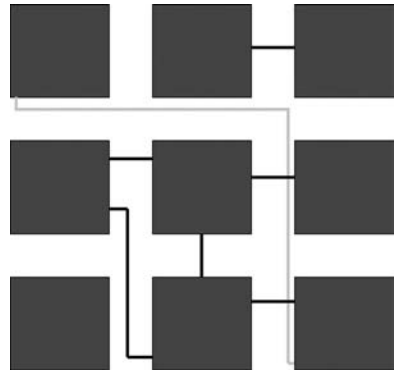


FIGURE 5.4
Illustration of a critical path in a 3 × 3 logic cell array.

With designs that approach the packing limits of a chip, routed paths are fitted in wherever free slots remain, often leading to serpentinelike routes. A smart router will try to avoid this by routing critical speed paths first, but this only partially alleviates the problem. Modern designs also attempt to employ pipelining of critical paths, but pipelining in an FPGA incurs a large delay overhead. Pipelining not only adds a register delay, but also the routing delays to get to and from that register. These routes are likely not a straight shot from source to destination, and must travel down to the active layer and back up to the higher metal layers in the process.

The FPGA often suffers from a high critical path delay to average path delay ratio. The higher this ratio is, the more of a potential benefit asynchronous design brings, because asynchronous design operates at the average case delay. As long as the overhead of the handshaking logic is less than the net speed gain achieved by asynchronous design, performance is gained.

The power gating effect of asynchronous FPGAs is also noteworthy. FPGAs typically do not have 100% utilization of their logic cells. In a synchronous design, the clock would be switching even unused cells. Programmable clock gating is often used as a solution, but this is usually not done on a level of granularity that can approach the benefits of asynchronous design. It would likely require too many added levels of logic in the clock tree and performance would suffer.

Several large corporations have recently taken an interest in asynchronous design, including Intel, Sun, Phillips, and IBM [3],[12]. It is reasonable to believe that future ASIC asynchronous designs will need to be prototyped. Asynchronous prototyping can be done on synchronous FPGAs, but it is extremely difficult to eliminate local race conditions and hazards when the FPGA does not have some asynchronous elements designed into the circuitry [13]. The need for asynchronous FPGAs just for protyping in the near future could be significant.

5.3 Basics of Self-Timed Design

5.3.1 Asynchronous Finite State Machines

Finite-State Machines (FSM) can be used to model any realizable circuit. A finite-state machine consists of combinatorial logic and state-holding elements (registers). Registers are typically clocked periodically to update the state of the system. Outputs of the registers are fed back into the combinatorial logic. The kind of FSM just described is actually a subset of the most generic FSM. The generic FSM does not require state holding elements, only delays in the feedback loops so that the outputs and inputs cannot change simultaneously. In effect, the synchronous FSM with clocked registers basically guarantees this constraint. In this sense, the asynchronous FSM is

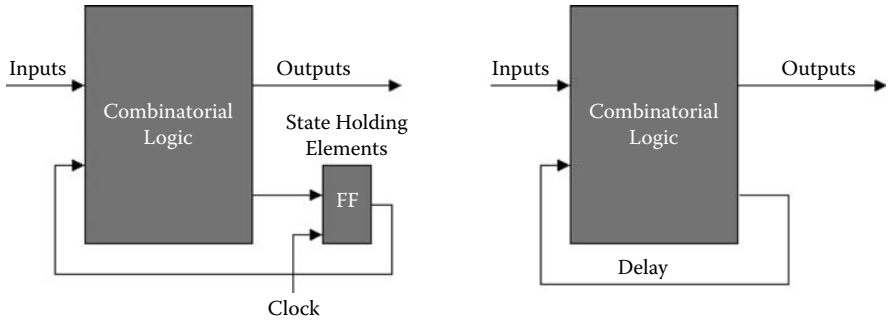


FIGURE 5.5
Synchronous and asynchronous state machines.

actually a simplified and more generic version of its synchronous counterpart (see Figure 5.5).

For the case of wave pipelining, the designer would estimate the intrinsic delays of the combinatorial logic and feedback routes and this delay would limit the max throughput of the system. However, as mentioned earlier, logic delays are difficult to estimate with high precision, and without on-chip sensing elements, the wave pipeliner has to account for worst-case conditions before the next wave of data can be sent through the system.

In self-timed FSMs, handshaking elements control when the feedback loop updates the system. This method has the advantage of allowing data to self-propagate, without any need to calculate externally when the next data should be made available.

5.3.2 Handshaking Protocols

There are a few fundamental protocols used to implement handshaking in self-timed designs. The most common approaches are known as "bundled data" and "dual-rail." Each of these protocols can be implemented in either a 2-phase or 4-phase fashion. There are variations on these protocols, such as push or pull channels, that are also discussed.

5.3.2.1 4-Phase versus 2-Phase

The 4-phase protocol is the most commonly implemented protocol due to its simplicity. It uses an interlocking "return to zero" method of handshaking. The transmitter pulses the request line high. When ready to transmit data, the receiver processes the new data, and pulses the acknowledge line high, signaling to the transmitter that it is finished and ready for more data. The rising acknowledge signal triggers the request line to reset, and the acknowledge line is similarly reset by the falling request line. Figure 5.6 illustrates this series of transitions.

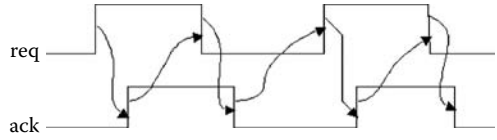


FIGURE 5.6
4-phase handshaking waveforms for two data transfers.

The main drawback with the 4-phase protocol is that the line resetting portion of each cycle has to be performed before the handshake logic can process the next bit of data. Often times, however, the resetting portion can be hidden by latching the state of the req/ack line and resetting it in parallel to the next stage's computation.

The 2-phase protocol uses a non-return-to-zero approach. Instead of using only rising-edge transitions as indicators, 2-phase uses rising or falling edge transitions to indicate a state change. Signals require no reset operation; they simply hold their state until the next computation is performed (see Figure 5.7). Because of this, the 2-phase protocol has a potential speed advantage.

However, the 2-phase protocol complicates the handshaking logic in the fact that the receivers must be dual-edge detecting. Often times, the completion signal that is generated by internal logic is single-edge active, so the transmitter must also include some logic to convert the pulse into a 2-phase signal. The author notes that in practice, this usually involves trading off SR latches in the 4-phase case for T-flops in the 2-phase. Note also that 2-phase has half the number of communication signal transitions, resulting in a potentially lower power solution.

5.3.2.2 Push-Channel versus Pull-Channel

A channel is basically just a word used to describe a datapath. Data can either be "pushed" through the channel by the transmitter sending data with its request, or can be "pulled" through the channel by the transmitter sending data with its acknowledge signal. In the push version, the transmitter makes the request, and receiver acknowledges. In the pull version, the receiver requests and the transmitter responds. Another way to think of it is, in the

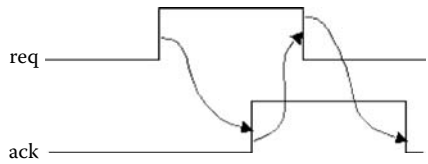


FIGURE 5.7
2-phase handshaking waveforms for two data transfers.

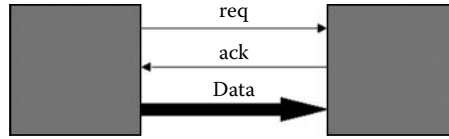


FIGURE 5.8
Bundled data diagram.

push case, the input side is saying, "I'm sending new data, process it." In the pull case, the output side is saying, "I'm ready for new data, send it to me."

5.3.2.3 Bundled Data versus Dual-Rail

The most straightforward way to implement a handshaking system is to use a data bus (which can consist of one or more data lines), a request line, and an acknowledge line. This is in fact, the bundled data approach. It is called bundled data because the request and acknowledge lines are bundled together with the data lines. See Figure 5.8.

A major caveat to the bundled data approach is that it is not delay insensitive. In order to function correctly, the request line (in the push channel case) must change state after the data becomes valid. Otherwise, the receiver could read old or metastable data. This constraint is known as the "bundling constraint." Most designs that used the bundled data approach account for this in one of two ways: by using delays on the control lines or by mimicking datapath worst-case parasitics. In the latter case, a copy of the critical path is made through which the acknowledge signal propagates. Note that only the gates in the specific critical path are copied, not the whole logic block.

Another major drawback is that if the data from a block finishes earlier than the propagation of the ack signal, it has to wait. From a delay performance side, this nulls out one of the main advantages of asynchronous design, because the design has been reduced to a worst-case propagation delay instead of an average-case. However, the bundled data approach is the simplest and can also be useful in interblock communications, where the delays are sometimes negligible compared to the intrablock delays. One benefit to the bundled data approach is that glitches are allowable if they happen on the data lines and occur prior to the arrival of the request signal to the receiver.

The dual-rail protocol, on the other hand, is a delay-insensitive alternative. It successfully combines the request and data lines so that any switching on the data lines also signals a request. This way, there is no bundling constraint between the request and data line transitions, because they are now the same wire(s). Figure 5.9 illustrates this method.

In order to encode both the request and data signals into one unit, the data/request signal must be able to represent three states: a null (empty)

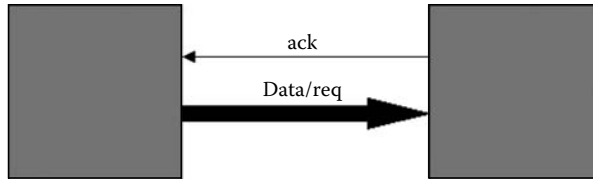


FIGURE 5.9
Dual-rail diagram.

state, and then a data high or data low state. If the receiver senses a null state, it does nothing, because there is no new data to process. If it senses a data low or data high state, the receiver treats this as a request and processes the data. To accomplish this, dual-rail encoding requires two data lines for each data bit, or four possible states per data bit. The possible states are 00, 01, 10, and 11. Here 00 denotes the null state, 01 denotes data low, 10 is data high, and 11 is an illegal (unused) state.

The advantage of dual-rail is that it fully utilizes the average-case performance benefit of asynchronous design, because a request signal is generated as soon as the data is ready. There is also no need for delay lines or critical path-matching circuitry. For the one-bit case, the number of data/control lines required is identical to bundled data. However, for larger buses, dual rail requires twice as many data signals as bundled data.

5.3.3 Completion Generation and Detection

A completion signal is simply a signal that denotes when a logic block is finished computing its current data. Synchronous circuits simply wait a clock cycle before latching in the next data value, under the assumption that the data has indeed been computed during that time. However, asynchronous designs require a signal to flag when a block is finished computing. Assuming the input constantly toggled (i.e., high, low, high, low) with each new value, this task would become automatic; one could just monitor output transitions. If one receives two logic 1s in a row or two logic 0s, however, there is no way to detect the end of the second computation.

The bundled data and dual rail protocols are not simply limited to communication between blocks. They are also methods of computation inside a logic block. Both protocols can generate completion signals.

In the bundled data approach, a request signal goes in one block, and generates a "start" signal, telling the datapath logic to start computation. The request signal propagates through a delay line and generates a completion signal, naively assuming the data processing has already finished.

Typically, dual-rail logic is implemented in some form of dual-rail dynamic logic. A true form of a Boolean expression is realized, along with its complement. Both the true and complement circuits are charged high during the

precharge phase. When the evaluation phase comes around, one of the two circuits will drop low, indicating the end of a computation and the output value. If the true portion drops low, the output is a one, and if the complement portion drops low, the output is a zero. As soon as the data is latched by the next stage, the circuit goes back into precharge mode. Thus, all that is necessary to generate a completion signal is to OR the true and complement outputs.

So far, we have discussed methods of completion generation. There are also methods of completion detection. The most common method is known as CSCD, or Current-Sensing Completion Detection. CSCD operates on the principle that the current drawn from the power supplies during a computation is at least an order of magnitude higher than when it is finished.* An analog current-sensing circuit is placed in the block that senses this current delta after the arrival of the start signal, and flags completion.

One advantage of CSCD is that one is not required to use dual-rail or even dynamic logic at all. There is an area trade-off between the CSCD circuitry and the complement copy of dynamic logic implementation.

Most realizations of CSCD draw significant amounts of static power due to the current sources embedded in the analog detectors. In many cases, multiple detectors must be used because the current through any one stage may be too small a change to measure. Because of this, it seems to make more sense to use CSCD only on larger block sizes.

5.3.4 Synchronization Elements

Synchronization logic requires hazard-free logic, because any glitch could cause an unwanted acknowledge or request. Glitches occur when multiple input combinations can cause the same output value. An OR gate does not function well as a synchronization element. For instance, a two-input OR gate could output a 1 if its inputs were 01, 10, or 11. Therefore, when the inputs transition from 01 to 10, the OR gate could go low momentarily, and then back high, causing a low glitch. AND gates have similar problems.

A circuit that is almost universally used in asynchronous design is the Muller C-element. The C-element has the property that it only transitions to 0 if all inputs are zero, and only transitions to 1 if all inputs are 1. Otherwise, it retains its previous state.

Therefore, the C-element meets the criteria above for being a synchronization element. There are other forms of logic that can meet the synchronization criteria but [1] claims that when dissected, they still boil down to the same basic function as the C-element. Figure 5.10 illustrates a C-element and one possible implementation. C-elements are used to implement any kind of synchronization between blocks, including both 2-phase and 4-phase protocols. Shown in Figure 5.11 is the 2-phase implementation.

From the diagram, the reader can see that the request signal only propagates when the acknowledge signal allows it to pass. Initially, data ready

* For more information on CSCD techniques, see [17].

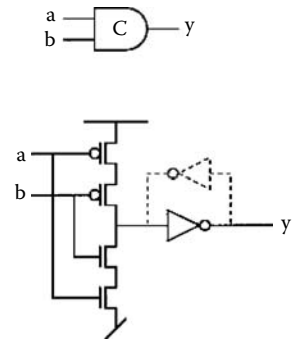


FIGURE 5.10
C-element symbol and a transistor implementation.

is zero, req is zero, and ack is zero. Data ready transitions, causing one to appear on req. If data ready transitioned again while the acknowledge signal had not yet changed, it would not propagate to the req line. This way, only one request signal can occur before the acknowledge transitions again.

C-elements can also be used to create an asynchronous FIFO (First In First Out) buffer. FIFOs are useful for storing data between an asynchronous domain and a synchronous one. Another common use is for storing values from a previous stage while the current stage computes data, so that when the current stage finishes computing, it doesn't have to take time to fetch the next piece of information. Chained together, they form multiple FIFO stages, where data propagates to the next stage if that stage is empty, and otherwise stays put. Figure 5.12 illustrates an asynchronous FIFO and its control signals.

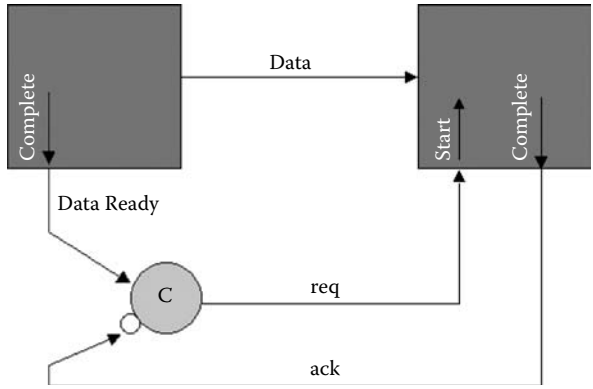


FIGURE 5.11
2-phase protocol implementation with C-element.

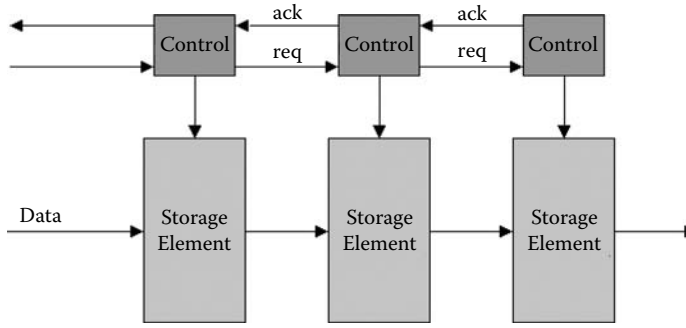


FIGURE 5.12
Asynchronous FIFO.

5.4 Prior Asynchronous FPGA Architectures

This section introduces several pre-existing asynchronous FPGA architectures, their methodologies, and structure. The previous asynchronous FPGA architectures that have been proposed take on a variety of approaches. Some approaches try to synthesize elements needed by asynchronous systems in a pre-existing synchronous FPGA through software. Still others have taken a synchronous design and have added more hardware to it to implement asynchronous elements. The remaining approaches involve building an asynchronous architecture from the ground up.

5.4.1 Triptych/Montage

The Montage architecture [4] is credited as being the first creation of an asynchronous FPGA. It is actually an architecture built within the synchronous Triptych architecture created at the University of Washington.

The Triptych architecture is a synchronous architecture that combines routing and logic into one programmable block that can be used for either. The core block consists of three input muxes that feed a LookUp Table (LUT) and a D flop, and three output muxes that receive the inputs and the LUT and D flop outputs. The idea is that the blocks not used for computation can still use their routing resources to pass along signals. These blocks are called RLBs, and are depicted in Figure 5.13.

Montage is the name given to the asynchronous qualities of the Triptych architecture. Montage takes a very bare-bones approach of adding a few key features into a pre-existing synchronous architecture to allow for software synthesis of asynchronous designs. The following is a description of these additions.

5.4.1.1 Asynchronous Circuit Synthesis

In order to synthesize asynchronous storage elements (such as C-elements or SR latches) out of FPGA logic, a fast intrablock feedback path is required.

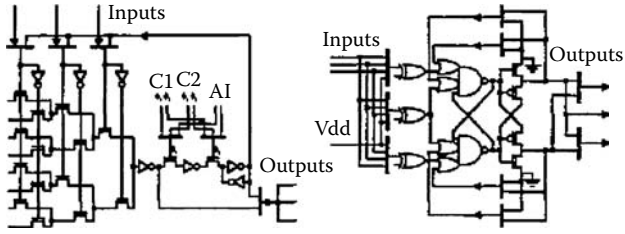


FIGURE 5.13
Schematic of Triptych/Montage logic block and arbiter unit.

This is because if the feedback path is routed outside the block and back in, the feedback delay may become large enough to generate race conditions between itself and the data propagation from a previous block. Montage solves this problem by introducing a hardwired feedback path from the output of the RLB back to the three inputs. The inputs are muxed with the feedback path so that any input into the RLB can select either the normal inputs or the feedback path. This path is shorter than any interblock connection so it ensures race-free storage elements.

5.4.1.2 Initialization

Asynchronous elements must be initialized on startup to some known state, similarly to synchronous designs. The Triptych architecture has D-latches that can be initialized to some value with a built-in initialization control signal. These D-latches can also be used to initialize the synthesized asynchronous elements without having to route a control signal to each block and thus increase the number of RLBs required to implement an element.

5.4.1.3 Hazard-Free Logic

Unless control lines are delayed through a block (as in a bundled data approach), circuits must be designed to be hazard free. The Montage architecture uses a lookup table decoder that guarantees if only one input changes, no glitching will occur, due to the fact that it switches once between two LUT memory cells. Although a requirement for proper functionality, many decoder designs ensure this. The burden of allowing only one input to change simultaneously is left to the software designer; that is, there are no measures made in hardware to prevent this.

5.4.1.4 Low-Skew Routing

A large class of asynchronous designs are not delay insensitive, and thus require isochronic fork constraints in some circuit building blocks. Isochronic forks are a set of physical locations that require negligible skew to

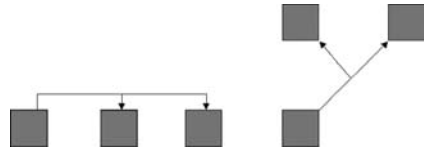


FIGURE 5.14
Typical 2-destination route, and Triptych/Montage low-skew version.

function properly. A major problem with many synchronous FPGAs is that interblock routing often has unpredictable delays. Especially if one signal is traveling to multiple destinations, it becomes very difficult to ensure that one destination is reached before another. Montage's claim is that its routing architecture offers many routes that have little skew between all of the possible destinations. For instance, consider a route that spans two blocks vertically. It would be difficult to ensure an symmetric isochronic fork constraint in a route such as this because there is an inherent wire delay Sskew in the times the first and second block receive the signal, namely, the time it takes for the signal to propagate one more block. Montage has routes that reach two RLBs simultaneously from a routing delay standpoint; see Figure 5.14.

For bundled-data systems, where the control line must change after the data line, Montage must use RLB gate delays to slow down a control path if it happens to be faster than the datapath, because there are no embedded delay lines for this purpose.

5.4.1.5 Synchronous/Asynchronous Interface

It is often desirable to allow a synchronous block to interface with an asynchronous one. Often times, this requires the use of more than one clock domain, as in the case of synchronous FIFOs. The Montage D-latches can select from one of two clocks.

Also, it is often necessary to determine the order of incoming signals at a synchronous/asynchronous interface. It is possible to build arbiters and synchronizers from general logic, but metastability issues and arbitration errors increase due to the larger delays incurred inside synthesized arbitration/synchronization blocks. Montage decided to include embedded arbiters that replace RLBs every so often in the core logic.

5.4.2 STACC Architecture

5.4.2.1 Globally Asynchronous, Locally Synchronous

The STACC (Self-Timed Array of Configurable Cells) [12] is a hybrid synchronous and asynchronous architecture developed by Robert Payne. The main idea of the STACC architecture is that it takes an existing synchronous FPGA architecture, strips out the global clocking, and replaces it with *timing*

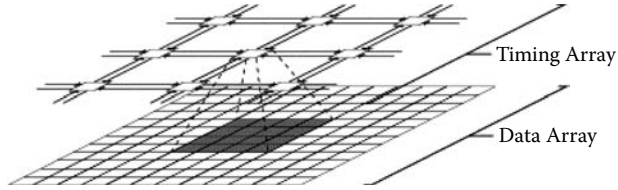


FIGURE 5.15
STACC architecture.

cells. A timing cell is a cell that controls the data flow between itself and other timing cells. It is, in a sense, overlaid on top of a small logic block array (say 4 × 4 or 8 × 8). Instead of making every logic block in this array self-timed, an array of logic blocks acts in a synchronous manner internally, using a local clock to control flip-flops in the array. Then at the interface between timing cells, it uses asynchronous request/acknowledge signals to transfer data across the boundary to the next timing cell. The local clock itself is generated from the timing cell. Figure 5.15 shows how each timing cell contains multiple data cells.

5.4.2.2 Timing Cells

The timing cell integrates synchronization, signal, merging, arbitration, and delay control into one cell. Timing cell communication uses a 4-phase bundled data approach. Programmable delay lines used for the control signals (used to satisfy the bundling constraint) are embedded into the timing cells. Payne chose to implement a bundled data protocol for two reasons: routing track area efficiency and synchronous design leverage. Assuming a data bus is greater than 1 bit, the bundled data approach requires less routing tracks. Bundled data requires $N + 1$ tracks for an N-bit bus. Dual-rail requires $2N$ tracks for an N-bit bus. Additionally, Payne states that synchronous designs can more easily be ported over in a bundled-data environment. Figure 5.16 illustrates the STACC timing cell.

5.4.2.3 Initialization

The STACC architecture has the unique ability of partial reconfiguration. Partial reconfiguration is when only a portion of a chip is reconfigured, and the rest of it remains untouched. Many asynchronous devices require the device to be globally reset to be initialized. STACC has a memory cell in each timing block that controls whether that timing cell will be reconfigured. Using this method, only selected timing cells can be re-initialized.

5.4.3 PGA-STC Architecture

PGA-STC (Programmable Gate Array for implementing Self-Timed Circuits) [14] is an architecture geared at implementing only asynchronous systems.

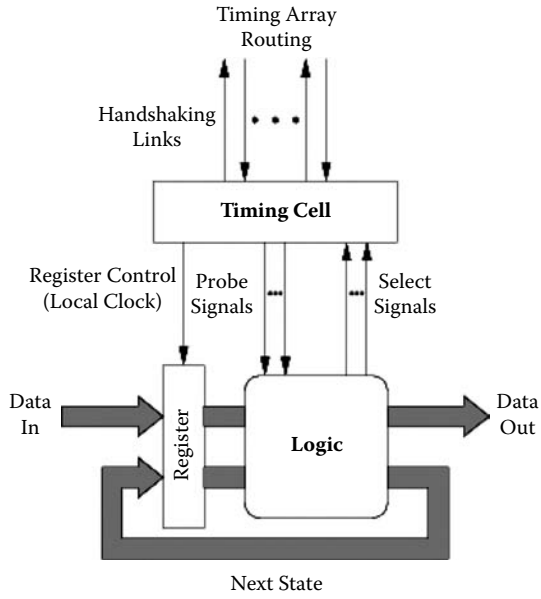


FIGURE 5.16
STACC timing cell.

However, it does not contain dedicated asynchronous handshaking elements. They must be synthesized from logic cells, similar to Montage.

The PGA-STC logic cell consists of a LUT, a feedback path (similar to Montage), and a mux that serves the purpose of putting all storage elements in the right state at initialization. The PGA-STC indicates that it uses a 2-phase bundled data protocol for routing, although details are not described. Dedicated arbiters are used as well. Figure 5.17 depicts the basic elements of the logic cell.

A main difference between the PGA-STC and Montage logic cells is the existence of a programmable delay element. Instead of routing control signals

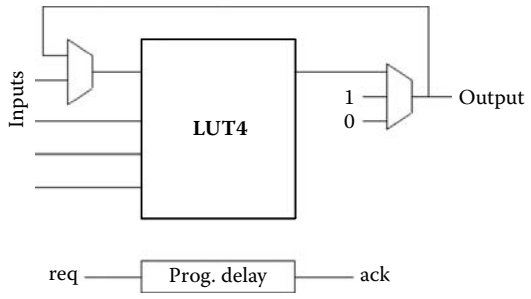


FIGURE 5.17
PGA-STC logic cell.

through unused blocks to satisfy the bundling constraint, the programmable delay element is programmed so that the control and data signals are closely delay matched and the bundling constraint is satisfied. Also, the programmable delay element is used to generate a completion signal for the block, and is closely matched to the worst-case delay through the logic cell.

The author makes the claim that the programmable delay element has a granularity that is fractions of a typical gate delay.

5.4.4 PAPA Architecture

John Teifel and Rajit Manohar developed a fully self-timed FPGA architecture at the ground level, known as PAPA (Programmable Asynchronous Pipeline Array) [15]. This architecture most closely resembles the design presented in this work, so it is used as the primary comparison/differentiation source.

5.4.4.1 Logic Cell

The PAPA logic cell consists of several functional blocks found in asynchronous designs. In addition to the traditional LUT4s, PAPA includes dedicated blocks for token splitting, merging, copying, sourcing, and sinking. For our purposes here, tokens are simply data bits that are processed in an asynchronous system.

In synchronous FPGAs, it is very common to have one signal fan out to multiple destinations, so that each destination is updated with the same data simultaneously. All that is required to do this is some kind of signal multiplexing and demultiplexing. In most commercial FPGAs, wires have multiple destination possibilities, which is the equivalent to a demux. Large muxes are used to "gather in" the demuxed signals. Each cycle, the previous data is overwritten under the assumption that it has been processed in the previous cycle.

In asynchronous design, muxing and demuxing are not that simple. In order to send the next bit of data to multiple destinations, the sender must receive an acknowledge signal from every receiver; otherwise, some data that had not yet been processed could get overwritten. This requires that the transmitter receive completion signals from each receiver before any of them can receive the next piece of data. Alternatively, the data could be copied and each receiver could interface with its own copy of the data, reducing logic fanout and complexity. This is in fact what the PAPA architecture does. It uses "copy" cells to make copies of the data token to each receiver.

Merge units are basically a dynamically controlled mux. A control signal decides which token to pass through to the output. Split units perform the opposite function of conditionally sending input data to one of two output channels, depending on the control value. Source units provide a constant valued token, and sink units consume nonessential tokens in the system.

The optimized PAPA logic cell consists of two LUT4s, a splitter, a merger, output copy units, and input copy units that feed the LUT4s (see Figure 5.18).

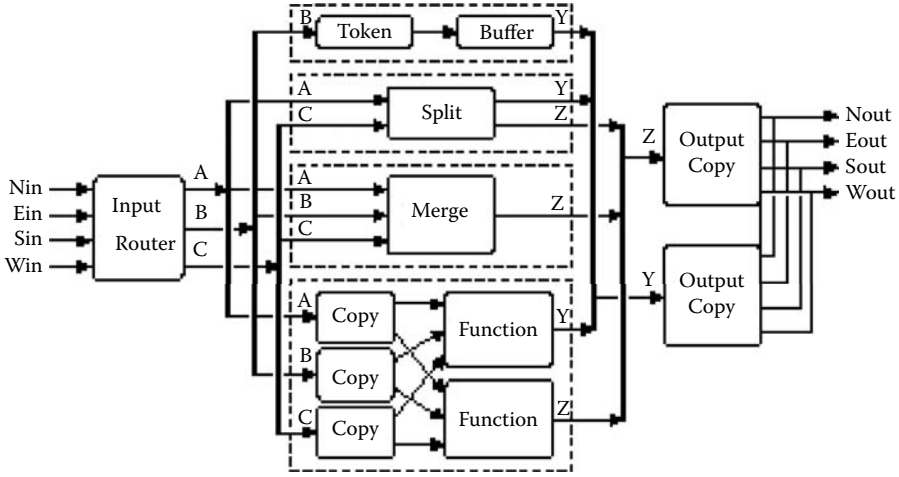


FIGURE 5.18
PAPA base architecture logic cell.

Each block consists of four inputs and four outputs. All of the inputs include asynchronous FIFO registers. The pipelines are useful because a given output token might not be processed by downstream logic the moment it has been computed. In the meantime, the input FIFOs can fill up with data, so that when the output token is fired, the next computation's latency is only intrablock (as opposed to waiting for the previous block to compute and then send its data).

The LUT4s use dual-rail logic to generate a completion signal. The LUT4 is implemented in a two-stage dynamic logic method that initially precharges all nodes high, and then either the true output or the complement output drops low during the enable phase, signaling the end of the computation. Initially, the entire LUT4 was implemented in a one-stage version, but charge sharing became such an issue that a two-stage version became a necessity (discussed more in Section 5.7).

Synchronous FPGAs often provide an arithmetic LUT mode, where the LUT is used as an adder, subtractor, comparator, and so on. To create arithmetic functions larger than one LUT, the carry-out of one LUT must be fed into the carry-in of another. To speed up the arithmetic function, a fast-carry path is usually built into the LUT, because the LUTs are tied together in a ripple fashion, where the carry propagation controls the max frequency of operation. The PAPA architecture implements a fast-carry chain that allows the carry to propagate to the next stage immediately if it is not dependent on the new set of input values. Otherwise, it functions as a normal ripple adder carry propagation. The PAPA logic cell also contains a state-holding element that generates a single token at reset, after which it behaves as a FIFO or a token initializer.

5.4.4.2 Routing

The PAPA routing architecture consists of a 4-phase dual-rail protocol. All of the routing muxes are pipelined as are the logic cells. Routing pipelining allows routing between logic cells to be broken up into multiple pipeline stages, potentially allowing a higher throughput rate. All routing connections are point-to-point; that is, each mux output drives only to one possible location. If a signal is required by multiple blocks, the signal must first be copied and then transmitted, as in the above description of the copy circuit. Single-pass gates are used to connect or disconnect a path between two blocks.

The routing fabric consists of pairs of lines (because it is dual-rail) that traverse four directions to the most adjacent logic cells to the north, south, east, and west. As mentioned earlier, each logic cell has a fanin of four and a fanout of four.

5.5 Problems with Previous Architectures

In the previous section, we examined the major asynchronous FPGA architectures to date. Here, problems with previous architectures are explored, and details are given on how the proposed architecture attempts to solve these problems.

5.5.1 General Issues

The approach of the author was to develop an asynchronous architecture from the ground up, as opposed to leveraging components of synchronous designs, under the assumption that what one gains in potential evolutionary usability, one loses in revolutionary performance.

The STACC architecture is an interesting one, but is more along the lines of an evolutionary architecture, inasmuch as all block-level computations are synchronous. There is a fair amount of overhead paid in area and delay to implement an asynchronous design at a localized level, so the idea of locally synchronous, globally asynchronous, is a valid one. However, the STACC architecture only offers the benefits of asynchronous design at a multiple-block-based level. STACC is geared towards synthesizing synchronous modules that communicate with other modules asynchronously. If any computation finishes inside this synchronous module before the next clock pulse arrives, the remainder of the cycle is wasted. Any performance gain from an asynchronous system in the STACC architecture is limited to the boundaries of the timing cells.

Although the Montage architecture includes some built-in asynchronous building blocks, it still lacks some of the basic components necessary to make a performance-competitive asynchronous system. The requirement of using "unused" logic cells as delay cells in the bundled data approach is

probably the most cumbersome. Elements such as C-elements and SR latches require an entire RLB for synthesis. These elements are common enough and necessary enough in asynchronous designs that they warrant dedicated hardware. For instance, an RLB block contains at least 40 transistors and 26 SRAMs, when a dedicated hardware C-element takes only 8 transistors and no SRAM cells.

Montage banks on a "low skew" routing architecture, which makes interblock delays more predictable. However, the routing will still suffer from PVT variations. In delaying control lines for the bundled data approach, the software must delay the control lines assuming worst-case conditions, thus losing a significant amount of performance. PGA-STC has the same issue, although the dedicated fine-granularity programmable delay could potentially increase performance and reduce die size if used often.

The architecture the authors envisioned fit somewhere in between the coarse granularity of the STACC architecture and the fine granularity of Montage. Because FPGA routing paths vary greatly in terms of delay, it seems reasonable to make each interlogic cell communication asynchronous, where the actual flow of data determines the max throughput, not the worst-case conditions of any given path.

The PAPA architecture most closely resembles the architecture the author designed. It seems to offer excellent performance, but at the cost of a large area (discussed more in Section 5.8). In addition, its extensive use of dual-rail domino circuits may become a power concern if it were to be designed in a smaller feature technology. Because the routing architecture is based on a dual-rail approach, it suffers from problems that are described in the next section.

5.5.1.1 Routing Fabric Issues

Commercial synchronous routing fabrics are extremely rich in metal connectivity. Current leading-edge chips use at least ten levels of metal interconnect to obtain the necessary interblock connectivity without causing the die to become metal-limited in area. The larger the number of metal tracks that are available, the greater is the potential interblock connectivity. If a logic cell has more connections to its neighbors, it is less likely to incur a detour delay penalty. For instance, in a very simple routing architecture, there may be one direct path from logic block A to logic block B. If that path is used, then in order to go from A to B, the software must find another less direct route. In this way, FPGA performance hinges greatly on the probability of being able to connect from point A to B with the most direct route. On the other hand, there are limits to the benefits of rich connectivity. At some point, the gate delays due to high fanin/fanout muxes and extra wire loading of a very connectivity-rich design bog down the system performance.

The biggest issue in making an asynchronous routing fabric as rich as its synchronous counterparts is the fact that with current communications protocols, each communication channel usually requires three wires (data, req,

ack or two data and one ack),* as opposed to one for a synchronous design. Right from the start, one loses 2/3 of the available nonlocal metal tracks. If the FPGA is equally active and metal-limited in area, this results in a loss of 2/3 of the routing connectivity, or alternatively, a 67% increase in area to obtain the original connectivity pattern.

Second, commercial synchronous routing fabrics use high fanout signals. Instead of point-to-point routing, signals have the option of going to one of multiple destinations. This further reduces the number of metal tracks and muxes needed to implement a connectivity-rich design.

Suppose one were to implement a routing fabric with multiple fanout destinations for each signal. Assuming the dual-rail or bundled-data protocols, the layout designer would have to closely match at least a pair of signals that travel to multiple destinations, often through multiple layers of metal. Otherwise, the two data/control signals would suffer either from performance degradation or loss of functionality if the delta between the two became too great.

What the authors proposed to do was find a method of combining the control and data lines into one signal. There are several key benefits to combining the control and data signals. The first is that the number of routing tracks required for a routing fabric is cut in half or 2/3, thus enabling the theoretical potential of two to three times the connectivity. Secondly, it eliminates the need for programmable delay elements to match the data and control signals, assuming a bundled-data protocol is in use. Because the control is embedded into the data signal, there is no question as to the order of signal arrival. Everything happens sequentially. The control always initiates a transfer, the data follows, and the control responds. It is possible to use a dual-rail routing fabric to avoid delay elements, but the area consumed by dual-rail logic is large, and two metal lines are still required for every one bit of data. Thirdly, combining control and data signals allows the metal routes to go easily to multiple fanout destinations, inasmuch as one only has to route one signal to each destination.

In summary, combining control and data lines between FPGA logic cells results in a potentially richer routing fabric, thus alleviating blockages and detours when the software attempts to route a design, leading to a faster overall system speed. There is a catch, however: the active area consumed by the new control logic must be smaller than the area overhead one pays with dual-channel routing. Also, the delay penalty incurred by the signal combination must be small enough to still outweigh the penalty paid for taking a detour in the routing for this idea to be beneficial.

5.5.1.2 New Architecture Proposal

At a high level, the authors set out to design an architecture that employs an asynchronous method of communication between logic blocks. The communication method uses a combined control and data line approach to reduce

* It is possible to use two wires if the one of the wires is bidirectional.

routing congestion and potentially improve on performance. Additionally, it was desirable to keep the logic cell smaller and more power-friendly than the PAPA approach, but still computationally efficient, and make additional improvements and discoveries while designing at the 90-nm technology node.

In the following two sections, the new architecture is described in detail. In summary, a new FPGA core is created. A combined data and control signal transmitter/receiver pair is developed to communicate between logic cells. A fully asynchronously pipelined logic cell is introduced to take advantage of the new communication protocol and also maximize throughput through the logic cell itself. The logic cell contains an area-efficient completion-generating LUT, with high-speed arithmetic capabilities, as well as fast feedback paths to minimize loop iteration bound (feedbacks that limit throughput rates). In addition, a routing architecture is proposed that maximizes resource utilization by allowing logic cells to function either as computation nodes, signal routers, or both. The name chosen for this proposed architecture is "RASTER," an acronym for Reconfigurable Array of Self-Timed Elements for Rapid Throughput.

Section 5.6 describes a new block-to-block communication method. Section 5.7 then introduces the new logic cell, and details a routing architecture using the communication blocks developed in Section 5.6.

5.6 RASTER Intercell Communication

The RASTER architecture revolves around the idea of combining handshaking and data communication signals into one wire. What is required is a protocol that has the ability to signal an empty state, along with a logic high and logic low. There are several methods that can accomplish this feat, such as using tertiary logic (three voltage levels), frequency modulation, multiple current drive levels, and others. After evaluation of a few of these, the method chosen was a pulse modulation encoding scheme. Figure 5.19 shows a diagram of a data transaction using this scheme.

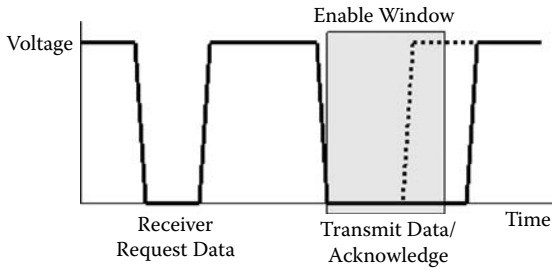


FIGURE 5.19
Pulse encoding protocol.

A bidirectional transmission line is used to communicate both control and data. Initially, the transmit and receive drivers are tri-stated and the transmission (comm) line is held high by weak PMOS keepers. When the receiver is ready to receive data, the receiver engages and pulses the line low and then back high. The transmitter interprets the rising edge of the pulse as a data request. When the transmitter has data ready for transmission, it pulls the line low again. Depending on whether the data is logic high or low, the transmitter keeps the line held low for different amounts of time. In the meantime, the receiver triggers off the falling edge the transmitter generates, starting an enable window. If the line is pulled to high by the transmitter before the enable window times out, the logic is regarded as a zero, else it is a one.

Both the transmitter and receiver rely on delay elements to generate the required pulses. In order to ensure the delay elements in both track each other well, the same type of delay element is used. This way, if the transmitter pulse lasts longer than its nominal delay, the receiver window will track it, assuming process/temperature/voltage gradients from source to destination are minimal. The implementation simulated used simple inverter chain delay lines, but for higher accuracy, more advanced delay lines could be used (current-starved inverters using bandgap references, for instance). Figures 5.20 and 5.21 show the transmitter and receiver pair implementation, respectively.

The transmitter and receiver both use a similar circuit to drive the transmission line. The communications (comm) line is first pulled low with an NMOS driver for a given pulse width. An edge detector detects the end of the internal pulse, which turns on a PMOS driver to actively drive the line back high again. When the driver senses that the line has been pulled high, it shuts off the PMOS driver, tri-stating itself. When the line is at logic high, PMOS keepers engage to weakly hold the line high. In this way, only the transmitter or receiver is driving the line at any given time.

On a request, the receiver generates a pulse that pulls the comm line low. In order not to treat its own request pulse as data, the receiver detector is activated on the rising edge of the request pulse. The transmitter contains a

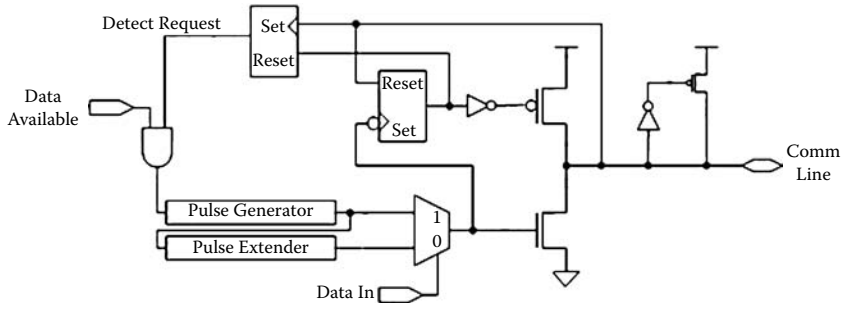


FIGURE 5.20
Pulse encoding transmitter.

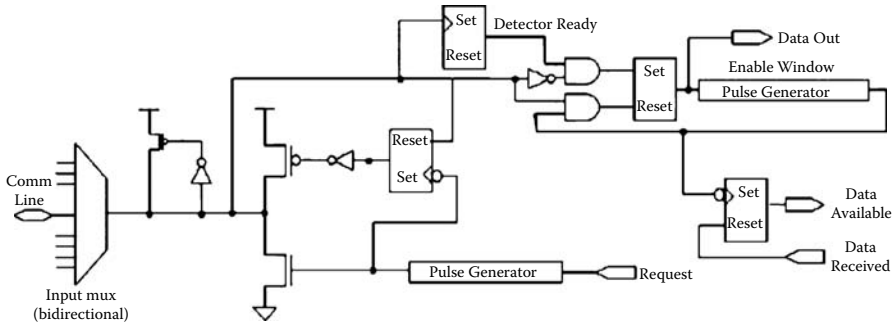


FIGURE 5.21
Pulse encoding receiver.

rising edge detector to sense when the receiver has requested data. On the rising edge of the request pulse, the transmitter activates itself, and if data is available, pulls the line low using a short pulse for logic 0 and long pulse for logic 1. The receiver initiates the enable window pulse generator after detecting the falling edge of the transmitter pulse. If the comm line is pulled high before the enable pulse times out, the state SR latch will be reset and store a logic 0; otherwise, the enable window will time out first and the SR latch will keep its logic 1 state. Each comm line wire fans out from one transmitter to 16 possible destinations, allowing for a fairly rich connectivity envirionment.

5.7 RASTER Logic Cell Architecture

In the last section, we went through a detailed description of how the interblock communication worked in the RASTER architecture. Now we explore the logic cell and routing architecture in greater detail, and how the communication blocks previously described fit into this architecture.

5.7.1 Logic Cell

5.7.1.1 Lookup Table

The fundamental unit of most any FPGA logic cell is the lookup table. The first question when designing a logic cell is, "What size lookup table should I use?" This is a complex problem that involves LUT delay, number of logic cells necessary for synthesis, and average LUT area. Several papers have been written that suggest the 4-input LUT (LUT4) is the most efficient size in terms of system speed per unit area [2]. In addition, most commercial products use the LUT4 as their baseline, so it makes comparisons more straightforward. Because of this, the LUT4 was chosen for the RASTER architecture.

The next choice was how to implement the LUT4. Because the unit was to be self-timed, it was necessary to decide whether to use dedicated completion generation circuitry or a programmable delay.

The advantage of the programmable delay method is that it potentially saves area. When circuitry must generate its own completion signal, it is usually done via a dual-rail approach, and thus large portions of the circuit must be duplicated. However, the programmable delay method suffers in performance, especially across PVT variations, because the delay must be programmed for worst-case conditions. Ideally, for a high-performance device, one would design the LUT4 in a way that generates its own completion signal at a minimal area cost.

The PAPA architecture initially set out to use dual-rail domino logic for their LUT4 design, but ran into "pull-down stack complexity and noise problems." Instead they broke the LUT into an address decoder stage that generates a logic 1 on one of 16 address lines, and the address lines feed 16 decode transistors that connect any one memory cell to the output rails (see Figure 5.22).

The RASTER architecture took a somewhat similar approach in breaking down the decoder into two stages, although the implementation was quite different. In order to minimize noise and multiple transistor stacking, a LUT was created that buffer isolates the decoder into small segments. The LUT is composed of a two-stage pass gate decoder with buffers placed in between each stage. Each segment has a much smaller capacitive load than a full dynamic logic approach, and has a maximum of two transistors in any path from supply to end load.

In addition, while using a dual-rail approach, area could be saved by duplicating only a portion of the LUT decoder. The 16 memory cells necessary in

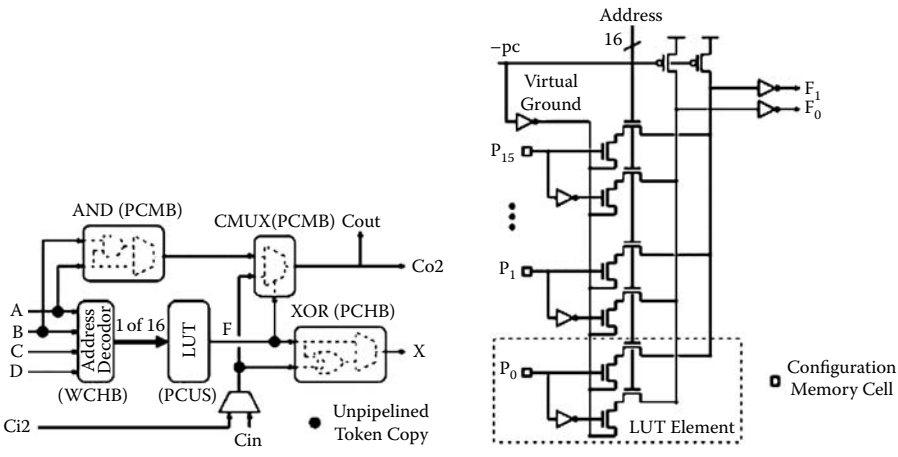


FIGURE 5.22
PAPA LUT4 implementation.

a LUT4 also need not be replicated. The predecoder decodes the four inputs (A, B, C, and D) into initial decoding signals. The predecoder then feeds two 2-level pass gate decoders, one for the true outputs of the memory cells, and one for the bar outputs. Buffers are inserted between pass gate stages to boost performance and isolate nodes. Half latches are connected to the buffers to act as level-restorers.

During precharge, the internal pass gate decoder nodes are pulled high and held there by the PMOS level-restorers. Therefore, the output is high. The predecoder is initially off, and none of the pass gate stages are turned on. When a start signal arrives, the predecoder is enabled, and turns on a path from a given memory cell to the output in both pass gate decoders. One of the pass gate decoder outputs remains high, and the other drops. The signals are NANDed together to generate a completion signal. On completion, the LUT4 resets itself by first latching the data, then turning off the predecoder, and finally pulling all internal nodes high again. Figure 5.23 illustrates a high-level diagram of the RASTER LUT4.

The area trade-off between the programmable delay method and dual-rail implementation is then fairly small. In the dual-rail approach, the only replicated circuitry is the pass gate decoder, which is probably comparable to the overhead of a programmable delay, while still retaining higher throughput.

The asynchronous LUT allows for multiple inputs to change simultaneously, unlike the restrictions placed on the Montage architecture and others. This is because the start signal will not toggle until a token has been received on each LUT input. After the last LUT input value has arrived, the start signal toggles.

Another benefit to the two-level dual-rail LUT4 scheme is in the way the fast ripple logic can be implemented. This is described further in the next section.

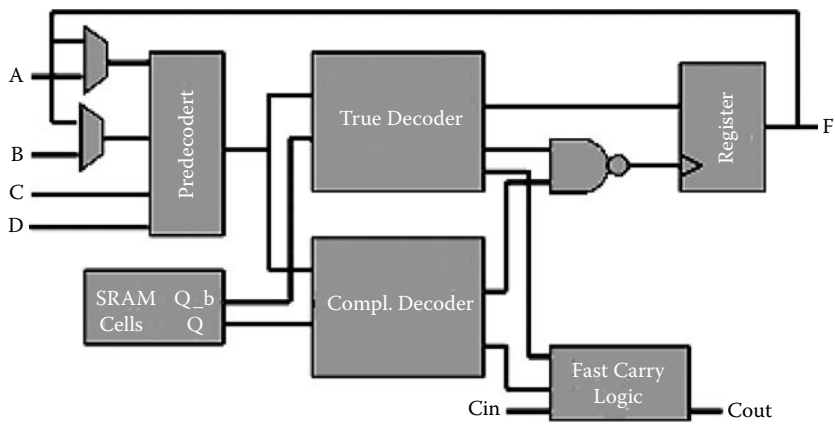


FIGURE 5.23
Block diagram of the main components of the RASTER LUT4.

5.7.1.2 Fast Ripple Logic

In addition to synthesizing logic functions and storage elements, most computation-intensive FPGAs support a mode of operation known as "arithmetic mode." Arithmetic mode allows one to synthesize logic that uses a carry chain. This includes elements such as adders, subtractors, counters, and comparators. Although these can be built without a fast carry chain, they quickly can become performance bottlenecks. For instance, the worst-case path through a 16-bit counter is when the counter rolls over from all ones to all zeros. In this case, a carry signal must ripple through all 16 bits of the counter. In a synchronous system, the clock frequency would be limited to this one path. A solution typically involves a short carry chain path between select logic cells, and potentially dedicated lookahead carry logic so that each carry-out does not depend directly on the block's carry-in.

In an asynchronous path, the speed is based on average case performance, which is a significant savings in itself compared to a synchronous system. Considering the previous 16-bit counter example, the worst-case path only occurs once in 65536 (2 ^ 16) cycles. However, for this architecture, further carry-chain delay reduction was desired.

A carry-skip approach to the fast carry logic was implemented. For every two logic cells, there exists dedicated logic that minimizes the carry-in of the first cell to the carry-out of the second cell path to 39 picoseconds. The fast path contains only a two-gate delay. In this way one can build large ripple logic structures with little impact on maximum throughput.

As alluded to previously, the dual-rail LUT has the ability to reduce the carry-in to sum path as well. Half of the LUT is programmed with a sum function assuming a carry-in of 1, and the other, with a sum function assuming carry-in of 0. The incoming carry-in signal then selects the appropriate LUT output through a 2:1 mux, without having to travel through the LUT itself. This method is similar to a method used in fast adders known as "carry-select." With a faster carry-in to sum path, the LUT can generate a completion signal quicker, allowing the carry chain to reset earlier and the LUT to transmit its data sooner.

It should be noted that it is possible to run an arithmetic unit at max frequency without the aid of a fast carry path through the use of pipelining. One could stagger the inputs into the unit in a staircase fashion (each A0 value leads the A1 value by 1, which leads the A2 value by 2, and so on). However, this requires a large number of pipeline stages and is somewhat cumbersome to implement. With the dedicated carry chain, all inputs can arrive at the unit simultaneously (with no requirements on how many inputs change), as they would in a synchronous environment.

5.7.1.3 Fast Feedback Path

Most commercial FPGA logic cells contain some sort of storage element, usually in the form of a D flip-flop. Because the RASTER logic cell is self-timed, the logic cells by nature require some kind of storage element to retain their

states until new data is transmitted or received. Therefore, each logic cell has its own built-in flip-flop.

However, there are some functions that require tight feedback loops. What this means is that the storage element feedback path loops back only one or two levels in upstream logic. In a high-throughput system, feedback loops can quickly become the bottleneck of a design, because they cannot always be pipelined without changing the function being implemented. For example, an FIR filter of the form $y(n) = x(n) + x(n - 1)$ could easily be pipelined to speed up operation. However, an IIR filter in the form of $y(n) = y(n - 1) + x(n)$ requires a feedback loop to the previous stage, and its throughput is limited by this feedback loop (iteration bound), no matter how many levels of pipelining one might add. An adder and an accumulator have similar problems. In order to add the next number to what is stored in the accumulator, one must feed the previous data back to the adder input before another operation can continue. State machines also frequently require tight feedback loops.

In the RASTER architecture, any nonpipelineable feedback loop would halve the throughput of any logic associated with that loop. Therefore a dedicated feedback loop was added to allow either the A or B input to be used as a state feedback. Instead of having to route to a neighboring logic cell and then return, the fast feedback loop can be utilized.

The logic for the fast feedback is minimal. All that is needed are two 2:1 muxes and a few inverters. There is no need for the feedback signal to interface with the handshaking logic, as long as the feedback loop is fast enough to stabilize the data at the inputs of the LUT before the next wave of data arrives. This constraint is easily met because the architecture requires a few hundred picoseconds to reset.

5.7.2 Routing Architecture

The routing architecture is straightforward. Each block has four transmitting channels and four receiving channels. Each transmitter spans four logic cells and taps into each logic cell along the way. At each tap, the transmitter channel connects to all four receivers in that logic cell, for a total of 16 possible destinations. Note that each transmitter can have only one active receiver; the other connections are all turned off.

Every routing channel travels through a 16:1 mux that feeds into the input of a receiver block. Because there are only 16 signals that go to any one logic cell, each receiver has a full crossbar connectivity pattern, meaning that it is exhaustive. Figure 5.24 shows the output connections from one logic cell.

Each of the four transmitters in a logic cell drive one orthogonal direction: north, south, east, and west. The LUT can drive any combination of the four transmitters. This way, in addition to being able to drive a signal in multiple directions, the signal can be copied up to four times in a given cell, as opposed to using dedicated token copying circuitry as in the PAPA approach.

To enhance the connectivity further, any receiver input that is not used by the LUT in its corresponding logic cell can route directly to the logic cell's

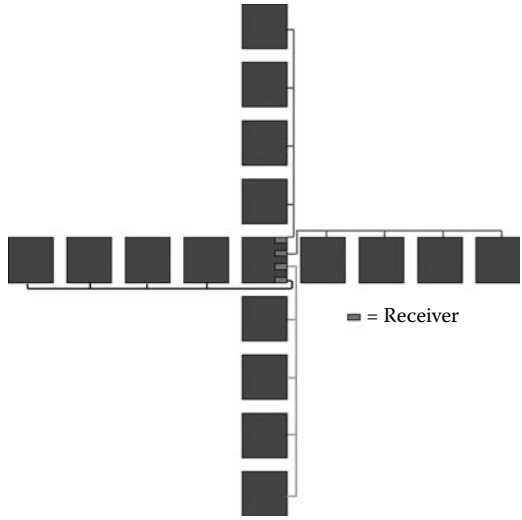


FIGURE 5.24
Routing connectivity for one RASTER logic cell.

transmitters. These paths are known as "passthroughs." For instance, if a logic cell is programmed to be a 2-input lookup table, two of the inputs are unused. These two unused inputs can be routed to any transmitters that are not used by the LUT output. This functionality allows designs to be packed into a fewer number of logic cells by allowing any unused elements to function as routing resources, similar to Montage's dual-nature cell approach. Note, however, in this mode each receiver input is hard-wired to only one corresponding transmitter, in order to reduce circuit complexity. However, each input can still be routed in any of the cardinal directions because the input muxes are full-crossbar, allowing the software to choose on which receiver to bring the input signal in.

If further routing resources are required to route a design, logic cells can be used only as routing cells; that is, the cell is programmed to use all four passthroughs. In this way, routability can arbitrarily be increased simply by spacing out computational cells between more and more routing cells, albeit at the cost of an increasing number of unusable LUTs.

By combining the signal copying of the LUT mode, routability of the routing cell mode, and packing potential of combining both modes, a routing architecture is developed with a fairly small number of channels needed per logic cell. Figure 5.25 illustrates three modes of operation for which the RASTER cell can be programmed.

5.7.3 Internal Logic Cell Synchronization

A four-phase internal logic cell synchronization method was used for minimal impact on the area-delay product. Two-phase is potentially faster

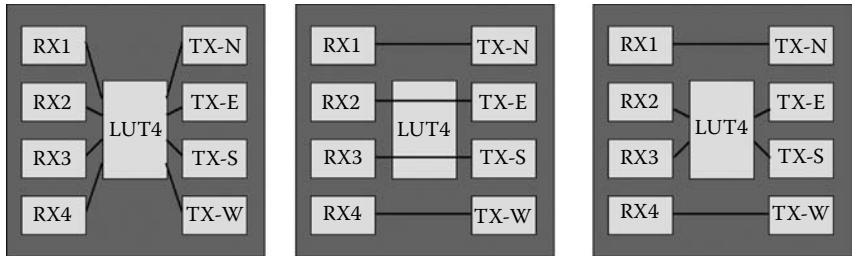


FIGURE 5.25
Three possible modes of operation for the RASTER logic cell.

but most sources claim a significant area overhead in choosing two-phase synchronization with minimal speed increase [1]. In addition, the receiver/transmitter communication between two blocks became the bottleneck of the system early on, such that all internal signals inside the logic cell had enough time to reset in the background while the next bit of data was fetched by the receiver.

The logic cell requires that all the LUT input data arrive before the LUT is allowed to start computing. After the inputs have arrived, the LUT can process the data assuming any transmitters coupled with the LUT have finished transmitting their data. If not, the LUT waits. After the transmitters have emptied their data, they send a signal to the LUT enabling it to start. From this point, the LUT computes the data and sends the output to its associated transmitters. The LUT resets itself while the data is being transmitted, and is now ready to receive the next wave of data.

Passthrough paths need only wait for their associated transmitter to fire before they pass their data from receiver to transmitter. So in summary, there are interlocking mechanisms for the receivers and transmitters associated with the LUT, and separate interlocking mechanisms for any passthroughs in the logic cell, so that passthrough and LUT data signals can move in parallel through the logic cell with no interdependence.

5.7.4 Internal Pipelining

In order to maximize throughput in the logic cell, it was necessary to add a pipeline stage between each receiver and the LUT. This way, the receiver could deposit its data into the pipeline register and immediately fetch another data bit. Otherwise, the receiver would have to wait for the LUT to process the last piece of data, transmitters to transmit it, and the LUT reset before fetching the next piece of data. With the pipeline registers in place,

the throughput of the transmitter/receiver transactions can be made roughly equal to the throughput of the LUT.

The pipeline registers are of a dual-rail variety. Dual-rail registers were necessary in order for the pipeline register to send the receiver a signal to fetch new data as soon as the pipeline register was empty. The dual-rail register generates its own empty signal for this purpose. Because the passthrough paths have no extra logic between transmitter and receiver, they are fast enough to bypass the pipeline registers and go directly to the transmitters without any throughput penalty.

Figure 5.26 illustrates the three types of paths through the logic cell that determine the cell's throughput. The dotted lines indicate the delay cycle that occurs for a transmit/receive transaction. For conciseness, the lines are shown with the transmitters feeding back to the receivers of the same block. In actuality, these paths would be coming from different blocks. Notice that the cycle only includes the delay through the transmitter, receiver, and pipeline register. The dashed line shows the LUT cycle which includes the delay from the LUT through the transmitter. The transmitter delay in the transmit/receive cycle is different from the one in the transmit/receive transition. The LUT transmit delay includes only the delay to transfer the data from the LUT to the transmitter and the transmitter signal back to the LUT that the data was received. Finally, the solid line indicates the passthrough delay path, which essentially includes a transmit receive cycle plus a small propagation delay to the transmitter (bypassing the pipeline register). All three of these cycles were optimized to provide maximum throughput without any one of them being significantly slower than the others.

5.7.5 Power-Up Initialization

Because there are many state-holding elements in the proposed architecture, such as SR latches, flip-flops, and LUT feedback paths, it is essential

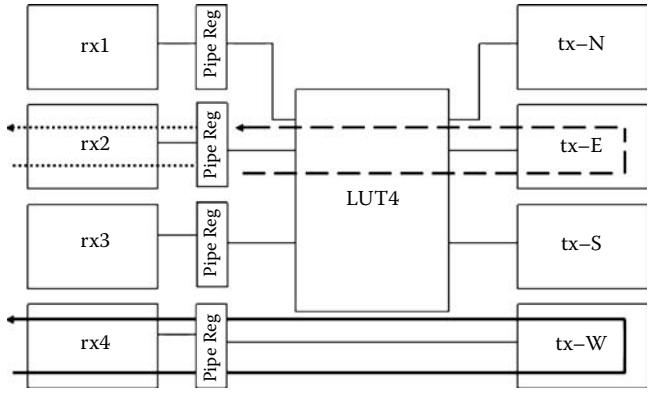


FIGURE 5.26
The three critical cycles that determine max throughput in the RASTER logic cell.

that each storage element comes up in the appropriate state upon power-up of the device. Instead of adding large amounts of circuitry to each storage element (i.e., converting flip-flops to asynchronous reset flops, etc.), a Power-On Reset (POR) signal was used. Most storage elements in the architecture contain one NMOS transistor whose gate is fed by the POR signal. The POR signal tracks VCC as the device is powered on. The NMOS transistors in the storage elements overdrive any feedback loops to initialize to the appropriate state. As long as the power supply is ramped at a reasonable rate (milliseconds), the feedback loops cannot overdrive the POR transistors. After VCC has reached full rail, the POR signal releases and shuts off all of the POR transistors. The logic cell POR circuitry was verified assuming a VCC ramp rate of one millisecond.

5.7.6 Implementation Notes

All schematics were created with custom logic (i.e., no standard cell libraries). The logic cell was simulated in its entirety in SPICE. The carry-chain speedpath, passthrough paths, and communication routes were simulated separately with estimated wire parasitics.

5.8 Simulation Results

There are three main metrics that need to be addressed when dealing with FPGAs: performance, area (cost), and power. Beyond these, requirements such as routability and packing density are also important. However, both of the latter require software to evaluate in a proper manner, which is not presently in existence for this architecture. Therefore, we focus here only on performance, area, and power.

5.8.1 Maximum Throughput

When estimating the performance of any system, one must consider the performance of the system as a whole. Therefore, speed estimators such as clock frequency in synchronous microprocessors are inadequate. For instance, the clock speed tells you nothing about how many operations can occur simultaneously, how many operations occur in a given cycle, and so on. The best test for performance is actually running a realistic software program and determining how long it takes to run. Similarly, in an FPGA, the best test of performance is to program the FPGA with a series of benchmark designs and determine system speed. This type of benchmarking is left for Section 5.9.

In the meantime, however, it is useful to find the throughput of an individual logic cell, if for nothing else than to put a ceiling on the maximum system throughput. It also decouples the logic cell's performance with the

performance of the routing architecture so that we can ascertain in the benchmarks to what the system throughput constraints are mainly due.

Because this is an asynchronous system, there is no maximum clock frequency that sets a ceiling on operation frequency. Instead, maximum data throughput sets the system speed.

Here, we focus on three main cycle paths that determine max throughput in the logic cell. It is necessary to take into account a cycle because the next piece of data cannot progress until after the previous data has propagated through the downstream logic and sent a completion signal to the upstream logic. The three cycle paths under analysis were previously described towards the end of Section 5.7. If any one of these paths is significantly slower than the others, it will likely dictate the maximum throughput of the system.

In order to characterize the logic cell, a list of necessary timing arcs that describe the delay from one point in the cell to the next was developed. In addition to being useful for throughput measurements, timing arcs are used by software to determine potential race conditions and maximum frequency estimations. All delays were obtained from SPICE simulations using 90-nm transistor models. Operating conditions assumed 1.2-V core voltage, 25°C temperature, and typical process corner.

In summary, the transmit/receive communication cycle takes 737 ps to complete. The LUT cycle takes 650 ps, and the passthrough cycle requires 769 ps. Taking the worst of the three, the logic cell max throughput rate is 1.3 GHz. Considering the main 90-nm competitors (Xilinx's Vertex 4 and Altera's Spartan II) have a 500-Mhz clock frequency, this is a data throughput increase of more than five times, assuming data throughput is 1/2 the clock frequency. To put it another way, the proposed architecture has an equivalent maximum clock frequency of 2.6 GHz.

The only other cycle of interest is the carry propagation cycle, which cited earlier, takes 39 ps per two logic cells. The latter half of the cycle (the acknowledge portion) actually happens in the background: while the carry propagates to the remaining cells, the previous cells reset. The only addition to the 39 ps per two logic cells is the resetting of the last cell in the carry chain, which takes an additional 64 ps.

The PAPA architecture was designed in a 0.25-µm technology, so direct speed comparisons are difficult.* At the 0.25-µm node, it could run several benchmark designs at a peak throughput of 400 Mhz. If we assume constant field scaling over three technology jumps (0.25 -> 0.18 -> 0.13 -> 0.09), assume that each technology has a scale factor of square root of two, and that speed is proportional to the scale factor, then the PAPA architecture would run at 400 Mhz * (2 ^ 1/2) ^ 3 = 1.1 GHz. Therefore the RASTER architecture has an 18% higher throughput rate in an equivalent technology.

* PAPA also quotes some performance results in a 0.18-µm technology, but does not report exact area or power numbers to go along with it. Therefore, the 0.25-µm technology comparison is used here.

Note that the PAPA architecture has several specialized blocks inside the logic cell that may give it a performance advantage at the system level. Although PAPA logic cells contain dedicated copy and state element blocks, RASTER can perform signal copies and state feedbacks fairly efficiently. Merge elements would require using a LUT, which may be slower than the dedicated merge element that the PAPA logic cell has. A split element would require the use of two LUTs in the RASTER architecture because each logic cell only has one output coming out of the LUT. If split and merge elements are common enough in asynchronous designs, it may be advantageous to allocate dedicated circuitry to perform these functions. However, it is potentially at a high area cost to add it into each cell when only a fraction use it, so the throughput increase would have to be substantial.

5.8.2 Area

Commercial FPGA logic cores are laid out in a custom fashion. Any area increase in one logic cell gets magnified by the number of cells in the array, and quickly becomes a significant portion of the overall FPGA area.

In order to treat the architecture as realistic, an area estimate based on custom layout rules was required. Time constraints did not allow for actual physical layout, so estimation methods were employed. The first method used was based on lambda rules. Lambda rules assume that the critical dimensions that determine layout packing density scale linearly with channel length from one technology node to the next. The logic cell was assumed to be near random logic, so a value of 1000 lambda^2 per transistor was used [23]. For a transistor count of 2536, this method gave an area of 4058 µm^2. Although not an extremely accurate method, it is still useful for comparison. The PAPA optimized architecture tile occupies 2.6 Mega-lambda^2, and the RASTER architecture occupies 2.5 Mega-lambda^2, thus slightly smaller by the lambda method. The PAPA area estimate is based on the 4-track per block routing architecture. Increasing the number of routing tracks bumps up the area estimate significantly. For instance, the optimized PAPA architecture with 30 routing tracks occupies 8.3 Mega-lambda^2 area.

In order to obtain more accurate results, a custom layout estimation method was used based on a paper by Moraes et al. [21]. It takes into account actual limiting design rules and average transistor sizes. Based on this estimation, the RASTER logic cell size is 2582 µm^2 per logic cell, or 51 microns per side, assuming a square tile. Note that the lambda estimation method is in the general vicinity (64 µm per side) as a sanity check.

5.8.3 Power

In deep submicron technologies, power has two major components: active (switching) power and leakage power. An 85ºC junction temperature was assumed in order to capture the effects of leakage power along with the active power component.

At DC, all internal nodes in the logic cell are driven to the VSS or VCC rail. Therefore, the power consumed when the logic cell is either unused or in a wait state is fairly low. Based on SPICE simulations at 1.2 V, 85°C junction temperature, each logic cell consumes 65 μW of leakage power. About 55% of the leakage power comes from the transmitters and receivers, mainly due to the large output drivers. The rest of the power lies in the LUT, pipeline registers, and handshake logic.

During active operation, power consumption is fairly high due to a few factors. The transmitters and receivers have to switch levels twice with every new input, as compared to a synchronous design switching only once. The LUT must be reset after every computation, also doubling its switching compared to a synchronous design. Handshaking logic fires request and acknowledge signals each cycle. Overall, most of the logic must transition twice in a cycle.

Active power calculations assume that the average active logic cell switches two of its four inputs every cycle, and two of the transmitters also switch. Under this assumption, the transmitters and receivers consume about half of the total active power.

At maximum throughput (1.3 GHz), the active power consumption per logic cell is 6.2 mW. Added to the leakage power, the total power consumption per logic cell is 6.3 mW. At lower system throughput frequencies, the active power drops to 2.4 mW (500 Mhz) and 1.2 mW (250 Mhz) per cell.

When calculating power for an asynchronous design, one must take into account the average system throughput, and what percentage of the chip is switching at any given time. Average system throughput is sensitive to (a) the design one programs into the array, and (b) how fast the inputs are allowed to change. The actual programmed design will often set the ceiling on the system throughput, even when individual system components could run faster, as we show in the next section. When trying to keep power minimal, as opposed to a synchronous environment, one cannot simply slow down a global clock to reduce the operational frequency of the chip. However, the maximum operational frequency can be controlled by the input switching frequencies. To think of it another way, after each wave of data passes from input to output, the external signals wait before sending the next wave, even though the chip could have processed them sooner.

Power numbers were estimated based on chipwide activity factors. The chipwide activity factor assumes what percentage of the chip is switching on average at a given time. Without software and actual design implementations, it is difficult to ascertain what the range of activity factors could be. However, most synchronous designs assume an activity factor around 10–25%, and because the authors are unaware of an average activity factor for self-timed designs, this was the chosen range. At this activity factor, power is bearable. Assuming higher activity factors, however, power gets out of hand rapidly for large array sizes, due to the multiple orders of magnitude delta between the static and active power components. For a device with 100 K active LUTs (LUTs used by the design) running at maximum throughput

and 10% activity factor, power consumption is around 70 watts. However, at a data frequency of 250 Mhz, power drops to 20 watts. For the rest of the array sizes, the power can range from 2 watts for a 250 Mhz design with 10 K active LUTs, up to 138 watts for a 1.3 GHz with 200-K LUTs active.

Now we compare the PAPA architecture with the RASTER power budget. PAPA consumed an estimated 26 pJ per cycle. Again, assuming that constant field scaling for the migration from 0.25-µm to 0.09-µm technologies, active power per block would remain unchanged. However, leakage power would increase dramatically, but because dynamic circuits are used extensively, we assume that the leakage component is small in comparison. If we normalize the cycle time to 1.3 GHz, the PAPA logic cell and associated routing would consume 33.8 mW. This is over five times higher than the RASTER architecture on a per logic cell basis.

In summary, we have found that, if the RASTER logic cell can run at maximum throughput rates, it is over five times faster than current synchronous FPGAs. When compared to the PAPA architecture, it is 18% faster, consumes less area, and uses 1/5 of the power that PAPA does.

5.9 Benchmarking

In the previous section, we examined performance on a logic cell scale. Now we explore the performance of this architecture from a system perspective.

In the absence of software tools for the RASTER architecture, benchmark designs had to be limited both in complexity and number. However, the small designs chosen give good indicators of the relative performance of this architecture versus the 90-nm competition assuming chip routability is not a major bottleneck.

The RASTER architecture was benchmarked against Xilinx's Virtex 4 and Altera's Stratix II, both from the 90-nm technology node. Three benchmark designs were chosen from the PREP benchmarking suite [26]. PREP was a nonprofit organization that was created to benchmark FPGAs in an unbiased manner by providing a suite of benchmark designs for each company to run on their own products. From the PREP suite the datapath, small state machine, and 16-bit accumulator were chosen. In addition to the PREP desings, an asynchronous state machine and an array multiplier were included.

No modifications were made to the PREP designs. The same design files were run on all devices. Xilinx ISE 7.1 software was used to test the Virtex 4 part, using the fastest speed grade. Altera's Stratix II device was tested using the Quartus 5.0 software package, using the fastest speed grade, with the optimized for speed option turned on. For the RASTER architecture, because no software yet exists, all designs were hand-routed.

Logic cell counts are normalized to a LUT4 basis. Virtex 4's base cell is the slice, which contains two LUT4s. Altera uses the ALUT, which also

contains two LUT4s. Note however, that Stratix II uses extensive input sharing between LUT4s under the assumption that LUT5s and LUT6s are more efficient block sizes, and thus LUT4 packing density will be less than a strict LUT4 architecture.

All operational frequencies are effective clock frequencies. For Stratix II and Virtex 4, this is simply the max clock frequency of the routed design. For the asynchronous architecture, this is the average data throughput multiplied by two, because the synchronous architecture's data frequency is half that of the clock.

5.9.1 Datapath Design

The datapath design was chosen because it is a singular serial path, and therefore is a good representative path for maximum throughput. There are no feedback loops or need to wait on intermediate inputs for the data to propagate through the design. The datapath design starts with a 4:1 mux that feeds a register, which in turn feeds an 8-bit shift register. This sort of design lends itself very well to pipelining.

Both Virtex 4 and Stratix II were able to run the design at their maximum clock frequency of 500 MHz. Because the datapath design allows for maximum throughput and there were no routing issues, the RASTER architecture could run it at an equivalent 2.6-GHz clock frequency. Inasmuch as each logic cell has a pipeline stage embedded in it, the number of logic cells required to implement the design was also low, only 11 logic cells. In contrast, Virtex 4 required 24 logic cells and Stratix II required 40. Figure 5.27 illustrates the datapath design.

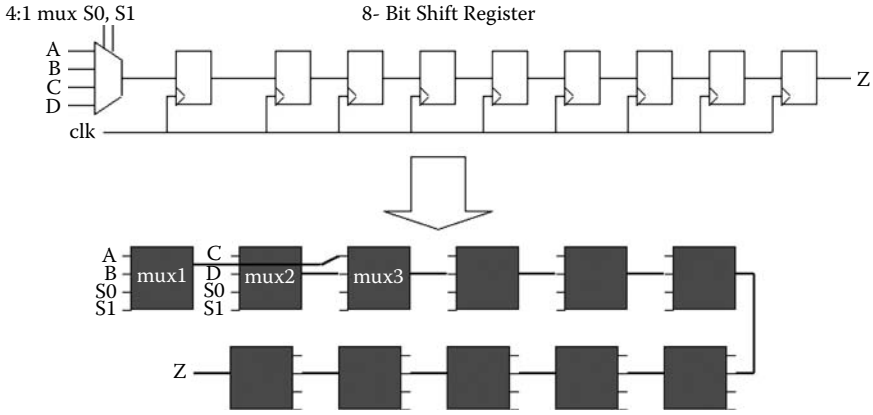


FIGURE 5.27
Datapath design and its synthesis into RASTER logic cells.

5.9.2 Synchronous State Machine

The small state machine design is a ten-state synchronous state machine, half of which resembles a Moore machine (having output transitions only on state changes), and the other half a Mealy machine (output transitions on state changes and input changes). There are eight inputs and eight outputs.

In order to run the synchronous state machine on an asynchronous architecture, the state machine first had to be converted to be hazard free. A one-hot state encoding method was used to ensure race-free state changes. For the initial estimate, it was assumed that the inputs could only switch after the outputs had stabilized. This is a worst-case assumption as far as propagation delay goes. For this assumption, the operational frequency was about 219 Mhz. This frequency was calculated by taking the average case path through the state machine, from input to output (assuming all inputs and states are equally likely). We can see here that the individual routing delays, being longer than the synchronous routing delays, start to add up. If we assume, however, that the inputs can change directly after the state registers change, then the frequency increases to 577 Mhz. Stratix II was able to run the design at max frequency (500 Mhz), but Virtex 4 was only able to run it at 487 Mhz.

If an asynchronous architecture was taken into account up front, it would be possible to further streamline the state machine through the use of a technique such as using a burst-mode state machine, and significantly increase speed. It is also possible to pipeline the state machine so that inputs can be fed in a constant stream manner, but this requires all internal feedback loops and input paths to have the same number of pipeline stages, otherwise the functionality would change. This approach is possible but puts difficult constraints on the routing of the design. Figure 5.28 shows the synchronous state machine and its implementation in RASTER cells.

The design implementation required 26 logic cells to perform actual computations, and 69 additional logic cells for routing. Most of the routing blocks were not fully used, and their spare tracks and LUTs could be potentially reused by neighboring modules.

It should also be noted that the synchronous design tools do not consider a logic cell "used" if only routing tracks and muxes in that cell are consumed.

5.9.3 Asynchronous State Machine

Because the previous state machine was geared for synchronous devices, it is educational to also implement an asynchronous state machine in all three environments in order to demonstrate the potential benefits the asynchronous architecture might have in this arena of circuits.

The asynchronous state machine chosen was a simple three-state pulse subtractor state machine [19]. Because it was not possible to use clocked registers for this design, asynchronous set-reset latches had to be synthesized into LUTs. Both Xilinx and Altera's software were not able to run this design near maximum frequency. Because there were no clocked registers to act as pipeline stages between logic, the timing analyzers assumed the worst-case

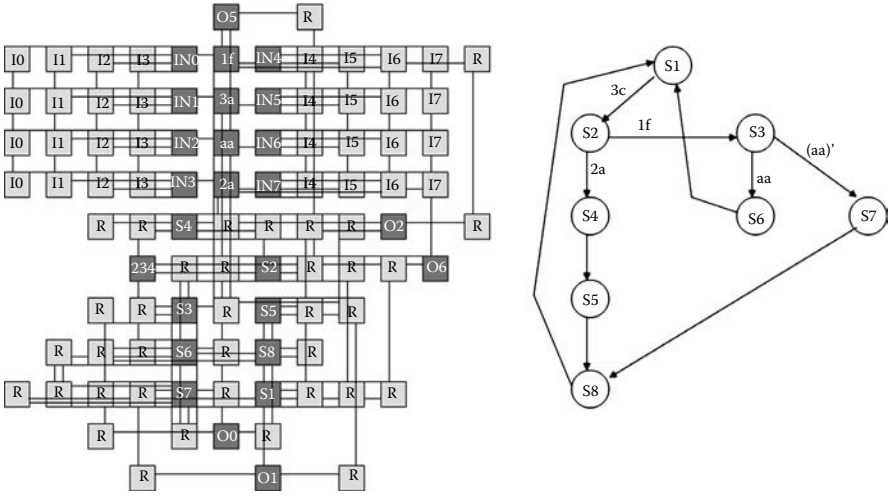


FIGURE 5.28
Synchronous state machine graph and implementation.

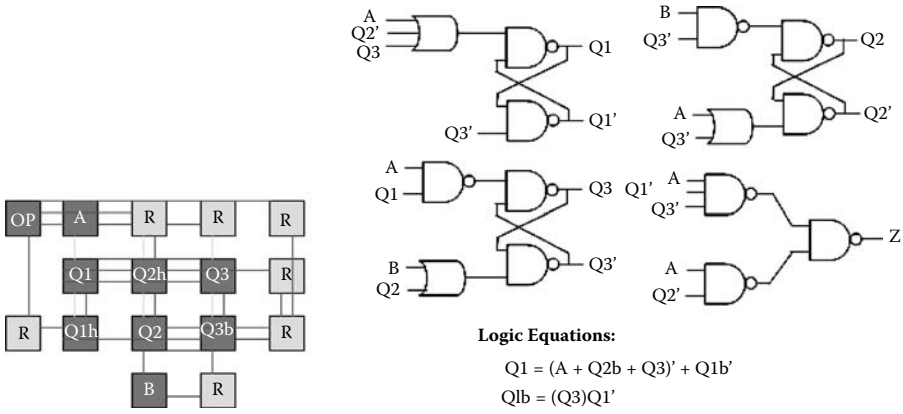
path through the whole state machine. In contrast, the RASTER architecture only has to worry about the average-case path through the state machine. In addition, with proper signal feedback, the asynchronous state machine could immediately send in the next wave of data inputs as soon as the state registers stabilized, whereas the synchronous architectures would have to wait until the next clock cycle. Stratix II was able to run the design at 416 Mhz, and Virtex II could run it only at 287 Mhz. The RASTER architecture could run it at 1030 Mhz. Figure 5.29 shows the asynchronous state machine and its mapping.

This design was more symmetrical and required less interconnection between computation elements, both lending to higher packing density. Out of 16 logic cells, 7 were required for routing only, and the rest performed computations. Stratix II and Virtex 4 both required only 8 logic cells.

5.9.4 Arithmetic Design I

For the arithmetic design, a 16-bit accumulator was chosen. The 16-bit accumulator is a combination of a 16-bit adder connected in parallel to a 16-bit register, with the register outputs fed back into the adder. The design also calls for a reset signal to initialize the register. This design demonstrates the efficiency of the RASTER architecture in a typical arithmetic environment.

Both Stratix II and Virtex 4 were able to route their designs at maximum frequency, as they both employ dedicated fast-carry chains for arithmetic mode support. Virtex 4 required 32 logic cells to implement this design, and Stratix II also required 32. RASTER has the advantage of having innate storage capability inside the logic cell, because the same value will be stored in



Logic Equations:

Q1 = (A + Q2b + Q3)' + Q1b'
Qlb = (Q3)Q1'
Q2 = (B + Q3b) + Q2b'
Q2b = (A + Q3b)' Q2'
Q3 = ((A)(Q1))' + Q3b'
Q3b = (B + Q2)' Q3'
OP = (A)(Q1b)(Q3b) + (A)(Q2b)

FIGURE 5.29
Asynchronous state machine and logic implementation.

the logic cell until all of the inputs change. Therefore, the number of logic cells required for the accumulator is only 16. However, the reset signal has a fanout of 16, so to avoid 16 individual reset signals, 16 more logic cells are required to route the reset signal to each cell. RASTER was able to run the design at the max frequency of 2.6 GHz. The fast feedback paths were used in this design for the accumulator feedback in order to support the maximum throughput rate. Figure 5.30 depicts the implementation of the accumulator.

Although the PAPA architecture does not quote a design speed for an accumulator, it does have data on a 16-bit bit-aligned adder. Assuming the difference between the two designs is slight, and assuming the same technology scaling factors as discussed in Section 5.8, the bit-aligned 16-bit adder design in PAPA would run at about 2.1 GHz.

5.9.5 Arithmetic Design II

For the second arithmetic design, a 4 × 4 array multiplier was chosen. Multipliers are frequently used in signal-processing applications in conjunction with fast adders. Owing to the self-timed nature of the RASTER logic cells, the multiplier implementation is fully pipelined by default. In addition to the pipelining done in the multiplier itself, internal pipeline stages must be inserted to distribute the input values to each partial product of the multiplier at the proper time. The worst-case path in the multiplier consists of the

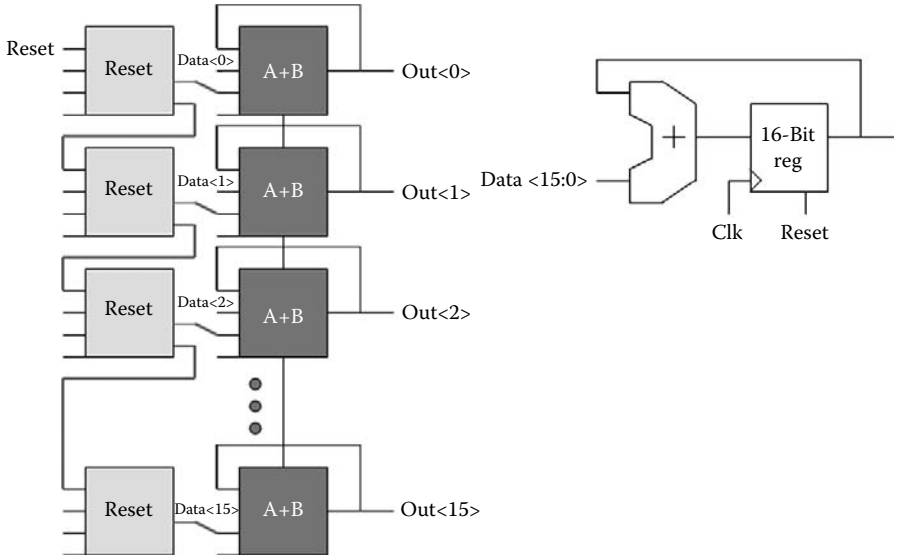


FIGURE 5.30
16-bit accumulator block diagram and synthesis.

baseline transmit to receive delay plus two carry propagation delays, yielding a max frequency of about 2.3 GHz.

From a packing density perspective, the multiplier is fairly compact in comparison to the more random logic of the synchronous state machine. 86 logic cells were used to create the multiplier, with 28 logic cells performing computations, and the remaining used for pipelining and input signal routing distribution. See Figure 5.31 for details.

Once again, in comparison, the PAPA architecture would be able to run at process-equivalent speeds of about 2.1 GHz for a hand-routed 4 × 4 multiplier. However, the number of logic cells needed was only 21.

Figure 5.32 summarizes the data recorded on each benchmark design. In conclusion, it appears that the RASTER architecture has significant speed advantages over the synchronous competition in the areas of datapath and arithmetic structures. As long as a given state machine is constructed with asynchronous behavior taken into account, state machines look potentially faster as well. For small designs, packing efficiency looks to be decent, but may be an issue with larger design implementations, and would need careful evaluation from a software place and route engine. Much of the routing efficiency depends on the ability of cells to be used both as computation and routing cells simultaneously.

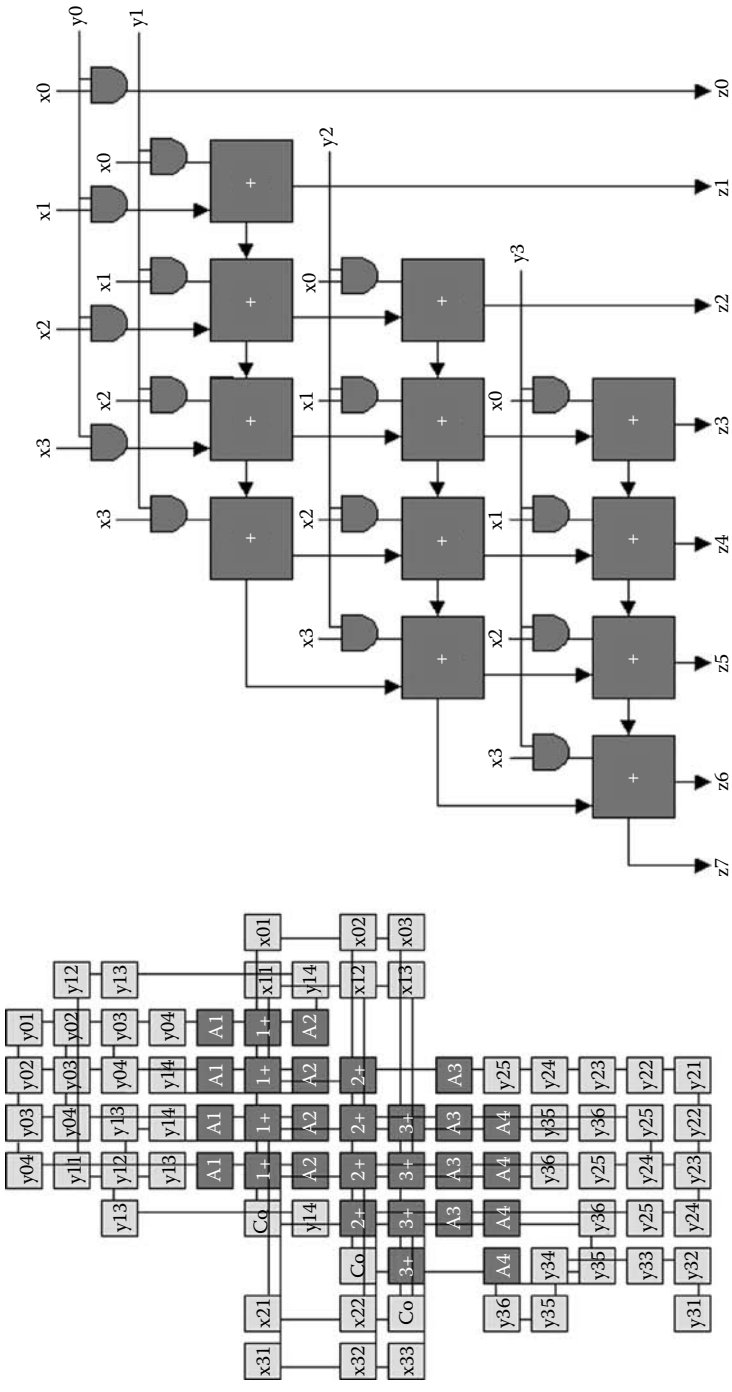


FIGURE 5.31
4 × 4 Array multiplier and synthesis into logic cells.

Max Frequency

	RASTER	Stratix II	Virtex 4	PAPA
Datapath	2601 MHz	500 MHz	500 MHz	x
Small Synchronous State Machine	204-537 MHz	500 MHz	487 MHz	x
Small Asynchronous State Machine	1030 MHz	416 MHz	287 MHz	x
16-bit Accumulator	2601 MHz	500 MHz	500 MHz	2132 MHz*
4×4 Array Multiplier	2300 MHz	x	500 MHz	2099 MHz

Equivalent Logic Cell Usage

Usage

	RASTER	Stratix II	Virtex 4	PAPA
Datapath	11 (11)	40	24	x
Small Synchronous State Machine	26 (95)	40	24	x
Small Asynchronous State Machine	9 (16)	8	8	x
16-bit Accumulator	16 (32)	32	32	16*
4×4 Array Multiplier	26 (86)	x	22	21

Altera Alut = 2 Lut4s
Xilinx Slice = 2 Lut4s

( )'s Denote Logic Cell Count with Logic Cells Used Only for Routing Included

Altera Software-1.2V Core, Speed Grade 3 (fast),Optimize for Speed Place & route, 25C
Xilinx Software-1.2V Core, Speed Grade-12 (fast), 85C (25C unavailable at current time)

* PAPA design is a 16-bit adder, not a 16-bit accumulator. Accumlator may require extra cells and incur more delay penalty.

FIGURE 5.32
RASTER benchmark analysis.

5.10 Conclusion and Future Research

After having completed this project, there are many areas that merit further research, negating any time constraints. In addition, there is also a short list of potential uses for the architecture.

5.10.1 Further Research

5.10.1.1 Power Reduction

In order to use this architecture for large array sizes in a cost-effective manner, it is likely necessary to reduce the active power consumption of the logic cells. This as we saw in Section 5.8 depends greatly on the overall activity factor of a given design, and also on the average number of input and output switching in a given logic cell. However, if power becomes an issue, the following are some potential power saving methods that could be researched.

Because the transmitter and receiver blocks account for 55% of overall power, it seems obvious that this would be the first place to start. Assuming all internal nodes fully charge or discharge in a cycle, active power is proportional only to the total switching capacitance on a node. Therefore reduction

of switching capacitance, especially on the communication channel lines, is critical. Leakage power in comparison to active power is low, so it may be worthwhile to introduce a lower Vt driver transistor for the communication blocks. Then the drivers could be sized smaller for equal delay, and thus lowering the switching capacitance on the communication channels. In addition, the entire transmitter and receiver blocks may be able to be sized down without significant performance losses. The design was initially started with an average transistor width of 1 μm for process variability reasons, but the 90-nm technology node allows for sizing down to around 0.2 μm. Note that after postlayout extraction, internal nodes may require such sizes to be driven appropriately, but a substantial reduction may still be possible.

For the logic cell, it may be worthwhile to switch to a typical LUT4 design that does not use self-resetting dual-rail logic. A simple delay-line completion generation approach may be better for power and area reduction. Because the routing throughput is low in comparison to the LUT4, performance reductions may not be noticeable.

The pipeline registers are surprisingly power-hungry for the amount of logic they contain, most likely due to the C-elements requiring ratio logic to overdrive previously stored values. In fact, throughout the design, all feedback elements used require the overdriving of feedback inverters. This was done to reduce area, inasmuch as there is a fair amount of C-elements and edge-triggered registers. However, area could be sacrificed here for power reduction by employing extra transistors to disable the feedback path when writing in new data.

5.10.1.2 High-Fanout Signals

It was obvious compared to synchronous architectures that the RASTER architecture had to use significantly more logic cells to route high-fanout nets. Because synchronous multiplexers typically have tens of destinations, most of which can be active simultaneously, they have an inherent area advantage to an asynchronous protocol that requires only one active destination. This problem was seen up front and was mitigated by providing each logic cell with the potential to copy a signal four times. However, control signals such as those used for reset and initialization are problematic. The POR signal provides power-up initialization, but any subsequent reinitialization would require powering down the part and then raising the supply again. In addition, some Petri net-based designs (used for transition-driven asynchronous state machine synthesis) require specific nodes to be initialized to contain tokens of a certain value.

A potential solution is to use one or two signals that act more on a global level. Each logic cell could select from one of the two lines, and have programmable inverts on the lines of a section that was to have some cells initialized high and low. If the signals are totally global, the solution is fairly simple in that there are no requirements to wait for individual LUTs to complete operations; everything is just interrupted and reinitialized. However, if one

were to want to reinitialize only a portion of the device (as in the resettable accumulator benchmark design), this would require handshaking signals to and from the global input source. Something that involves a row- or column-based global signal approach is probably the most efficient.

Because the routing architecture does incorporate multiple fanout destinations for each signal, it is possible to make more than one active at a time, thus increasing routability. However, a similar problem to the one described above is encountered for global signals in that it would require the routing and interfacing of more handshaking signals between nearby blocks, potentially defeating the pulse-encoding method of communication. Limited fanout allowance, once again on a small row or column basis, might be worthwhile.

5.10.1.3 Software Place and Route Tools

Given the time, a tailored software place and route engine should be made for the architecture to truly test it on a larger scale. Early limitations of the architecture surfaced in the hand placement and routing of the benchmark designs that allowed the authors to go back and make improvements. An automated tool would greatly enhance productivity and allow for more iterations and the uncovering of bottlenecks at a higher level.

5.10.1.4 Better Mux Performance

Although typical NMOS pass gates make very efficient muxes, they have their drawbacks. NMOS pass gates pass a logic high signal poorly, and because of this reduce path performance and robustness over extreme temperatures and voltages. To combat this problem, one can use a higher voltage supply on the gate connection of the transistor, as long as the gate oxide can withstand the higher electric field. This is often done in practice, and the RASTER communication paths would likely have sped up significantly had this method been used, and also would have been more robust on extreme PVT corners.

5.10.2 Potential Uses

There is a variety of potential uses for this architecture. Here is a list of a few.

1. *Stand-alone architecture for high-throughput designs*. RASTER seems to have significant advantages to implementing asynchronous designs; this alone warrants potential stand-alone use. The RASTER architecture could also be used in the same traditional products as a typical synchronous architecture, provided that the software is intelligent enough to translate synchronous designs into asynchronous versions.
2. *Prototyping ASIC asynchronous designs*. Although synchronous FPGAs can and have been used to synthesize asynchronous components, it

is a complex and inefficient process. Using an inherently self-timed architecture seems like a much more efficient alternative for asynchronous ASIC prototypes.

3. *Using as glue logic for multicore chips or systems on a chip (SOCs).* Large embedded systems are becoming commonplace. For complex designs that require several different bus transaction standards and block interconnections, it may be very valuable for time-to-market deadlines to use a reconfigurable high-speed interconnect between IP cores, processor cores, and the like. In addition, this would give the SOC designer the benefit of actually changing bus protocols if a more efficient one were to come along at a later date, or allow "patches" to be made to the design once it was completed.
4. *Using as an embedded block in a synchronous device.* Similar to the use of embedded cores such as DSP blocks and processor cores, the RASTER logic cells could be embedded within a synchronous architecture in small arrays. This would provide synchronous FPGAs the ability to offload high-speed logic into the RASTER array without having to increase the entire chip's clock frequency. These arrays could be interfaced via FIFOs or by using handshaking signals that selectively gate the clock when the array gets overburdened.
5. *Signal-processing applications.* Digital signal processing requires an abundance of fast adders, fast multipliers, and delay cells. The RASTER logic cells are able to implement all three of these blocks efficiently. Using an FPGA as a digital processing unit instead of a dedicated DSP chip would offer the advantage of embedding additional logic in with the datapath processing elements, such as a control unit.

5.11 Conclusion

The RASTER architecture offers potentially significant system performance increases over synchronous FPGAs. There are still potential issues that may need to be investigated before using this architecture in a commercial setting, such as active power consumption, routability, packing density, and robustness across all kinds of process variations and corners. In general, though, it offers some solutions to the problems associated with synchronous design, and makes potential improvements on previous asynchronous FPGA architectures. The hope is that ideas presented in this chapter will profit the reader as well as pave the way for future advances in the area of asynchronous FPGA design.

References

[1] J. Sparso and S. Furber. *Principles of Asynchronous Circuit Design—A System Perspective*. Boston: Kluwer Academic, pp. 4, 16, 20, 2001.

[2] S. Brown, R. Francis, J. Rose, and Z. Vranesic. *Field-Programmable Gate Arrays*. Norwell, MA: Kluwer Academic, pp. 4, 103, 1992.

[3] http://www.electronicsweekly.com/Article18998.htm, Wednesday 16 February 2000.

[4] G. Borriello, C. Ebeling, S. Hauck, and S. Burns. The Triptych FPGA architecture. *IEEE Transactions on VLSI Systems*, vol. 3, no. 4, pp. 492–497, December 1995.

[5] C. Meyers. *Asynchronous Circuit Design*. New York: Wiley-Interscience, Ch. 5, 2001.

[6] http://www.cs.manchester.ac.uk/apt/projects/, processors/amulet/AMULET1_uP.php.

[7] J. Rabaey, A. Cahndrakasan, and B. Nikolic. *Digital Integrated Circuits—A Design Perspective*. Upper Saddle River, NJ: Pearson Education, p. 511, 2003.

[8] S. Furber and J. Garside. *Amulet3: A High-Performance Self-Timed ARM Microprocessor*. University of Manchester, p. 1.

[9] A. Semenov, A. Koelmans, L. Lloyd, and A. Yakovlev. Designing an asynchronous processor using Petri nets. *IEEE Micro*, vol. 17, pp. 54–64, 1997.

[10] B. Gaide. A high-throughput self-timed FPGA core architecture. Masters report, University of Texas at Austin, 2005.

[11] B. Gaide and L. John. A high-throughput self-timed FPGA core architecture, Digest of *UCAS-2 (Workshop on Unique Chips and Systems*), held in conjunction with *ISPASS 2006*, March 2006.

[12] R. Payne. Self-timed field-programmable gate array architectures. Doctoral dissertation, University of Edinburgh, pp. 29, Ch. 5–7 , 1997.

[13] R. Payne. Asynchronous FPGA architectures. *IEEE Proc.-Comput. Digit. Tech.*, vol. 143, no. 5, September 1996.

[14] K. Maheswaran. Implementing self-timed circuits in field programmable gate arrays. Master's thesis, U.C. Davis, 1995, Ch. 3.1.

[15] J. Teifel and R. Manohar. An asynchronous dataflow FPGA architecture. *IEEE Transactions on Computers*, vol. 53, no. 11, pp. 1379–1387, November 1997.

[16] C. Wong, A. Martin, and P. Thomas. *An Architecture for Asynchronous FPGAs*. Department of Computer Science, CAL-Tech., pp. 172–174.

[17] M. Dean, D. Dill, and M. Horowitz. *Self-Timed Logic Using Current-Sensing Completion Detection (CSCD).* Computer Systems Laboratory, Stanford University, September 1992.

[18] W. C. Elmore. The transient response of damped linear networks with particular regard to wideband amplifiers. *Journal of Applied Physics*, vol. 19, issue 1, pp. 55–63, January 1948.

[19] C. H. Roth. *Fundamentals of Logic Design*. Fourth Edition. Boston: PWS, pp. 681–688.

[20] K. Killpack, E. Mercer, and C. Myers. A Standard-Cell Self-timed Multiplier for Energy and Area Critical Synchronous Systems. University of Utah.

[21] F. Moraes, L. Torres, M. Robert, and D. Auvergne. Estimation of layout densities for CMOS digital circuits. *Patmos '98 International Workshop, on Power and Timing Modeling, Optimization and Simulation*. 1998.
[22] J. Teifel and Rajit Manohar. *Highly Pipelined Asynchronous FPGAs*. Cornell University, pp. 8–9.
[23] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Boston: Pearson Addison Wesley, p. 60, 2005.
[24] T. Sakurai. Simple formulas for two and three dimensional capacitances. *IEEE Transactions on Electron Devices*, p. 183, vol. 30, issue 2, February 1983.
[25] J. Xu and W. Wolf. Wave pipelining for application-specific networks-on-chips. *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, p. 199, 2002.
[26] http://web.archive.org/web/19961230105139/http://www.prep.org.
[27] C. Traver, R. Reese, and M. Thornton. Cell designs for self-timed FPGAs. *14th Annual IEEE International ASIC/SOC Conference*, 2001.
[28] http://www.cs.man.ac.uk/async.
[29] S. Hauck, S. Burns, G. Borriello, and C. Ebeling. An FPGA for implementing asynchronous circuits, *IEEE Design & Test of Computers*, Vol. 11, No. 3, Fall, 1994.
[30] E.A. Walkup, S. Hauck, G. Boriello, and C. Ebeling. Routing-directed placement for the Triptych FPGA, *ACM/SIGDA Workshop on Field-Programmable Gate Arrays*, February, 1992.
[31] C. Ebeling, G. Boriello, S. Hauck, D. Song, and E.A. Walkup. Triptych: A new FPGA architecture, *Oxford Workshop on Field-Programmable Logic and Applications*, September, 1991.
[32] R. Payne. Self-timed FPGA systems. 5th International Workshop on Field Programmable Logic and Applications, 1995.
[33] I. Sutherland. Micropipelines. *Communications of the ACM*, vol. 32, pp. 720–738, June 1989.

6

The Continuation-Based Multithreading Processor: Fuce

Masaaki Izumi, Satoshi Amamiya, Takanori Matsuzaki, and Makoto Amamiya
Kyushu University

CONTENTS

6.1 Introduction

Processor architectures have achieved performance improvements by using instruction-level parallelism in processors. In particular, the performance enhancement of superscalar processors is remarkable. However, the problem in superscalar processors is that they cannot use whole parallelism

because the processors are limited in their ability to exploit instruction-level parallelism from single-process execution or single-thread execution [3]. In contrast, multithreading processors that exploit thread level parallelism are researched. The Simultaneous Multithreading (SMT) processor [7][9] executes two or more processes or threads simultaneously and achieves the improvement of throughput. A typical example of the SMT processor which is made for business is the Pentium 4 supporting hyper-threading technology [9].

Moreover, by the semiconductor technology advancement, the Chip Multiprocessor (CMP) [5][10] equipped with two or more processor cores in the chip has been researched in recent years. Because there are two or more processor cores in the CMP, the CMP can execute two or more processes or threads at the same time. SPARC T1 [14] and the IBM POWER5 [13] are examples of such commodity CMPs. The SPARC T1 has eight fine-grained multithreading processor cores that execute four threads concurrently.

However, in the multithreading processor, the overhead of thread scheduling that OS manages is increased, because the thread scheduling becomes complex unlike a uniprocessor. The overhead exists in thread scheduling which is managed by the OS, and this overhead does not sufficiently bring out the performance of the multithreading processor.

We are developing the Fuce processor [2], which is based on the advanced version of a dataflow computing model. The Fuce processor adopts the programming model based on the continuation-based multithreading model and pursues parallel execution in thread level. The Fuce processor is a CMP equipped with eight thread execution units to perform concurrent multithread execution with the hardware units. The Fuce processor reduces the overhead of thread execution management due to the hardware-level multithread execution control.

6.2 Continuation-Based Multithreading Model

6.2.1 Continuation

The core concept of the Fuce thread execution model is the continuation [1], which is an advanced version of the dataflow computing model. More concretely, in Fuce, continuation is defined by the static dependency analysis among threads, whereas in the dataflow model the continuation is defined by the dependency relation between different computational elements (operations). Here, the thread is defined as a block of sequentially ordered machine instructions, which is executed exclusively without interference. Note that this thread definition differs from the typical definition of a nonblocking thread [4][6][12], or the definition of a block in TRIPS [11]. From the architectural point of view, in Fuce the processor pipeline may be blocked during execution within a thread, even when memory access occurs.

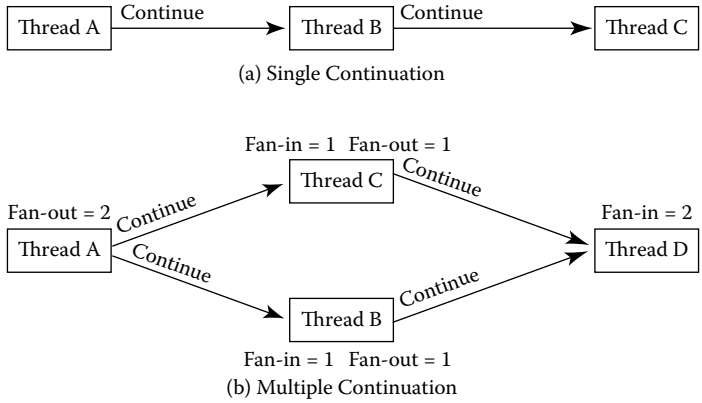


FIGURE 6.1
Thread continuation.

Figure 6.1(a) shows the single-dependency relation among thread A, thread B, and thread C. Thread B requires the computation result of thread A, and thread C requires that of thread B. In order to complete execution of all these threads, thread A has to notify thread B of the result of computation, and thread B has to notify thread C. We define this notification of the result as continuation. In typical RISC manner, the action of continuation can be divided into two actions, namely: transferring the computed result data and sending the continuation signal. Therefore, in Fuce, the data passing and the continuation are clearly separated in program code. The action of continuation is explicitly specified in a thread program, separated from the data passing, by using a special machine instruction.

We introduce two types of threads, called predecessor and successor. The predecessor thread is a thread that notifies the continuation to another thread, and the successor thread is a thread that will be notified by another thread. In Figure 6.1(a), for example, thread A is a predecessor thread of thread B and thread C is a successor thread of thread B. Figure 6.1(b) shows an example of multiple successor threads of a predecessor thread. Thread B and thread C can be executed concurrently, because there are no dependencies between thread B and thread C. Thread D cannot be executed before completing execution of thread B and thread C. Because many threads like thread B and thread C are considered to exist simultaneously in typical program code, effective parallel processing will be much more possible if we design an appropriate processor that supports the exclusive multithread execution model.

We introduce two numbers, called fan-in and fan-out. Fan-in is defined as the number of predecessor threads, and fan-out is defined as the number of successor threads. In Figure 6.1(b), the fan-in of thread D is two, and the fan-out of thread A is two. The order of thread execution is controlled by the continuation. Each thread decreases its fan-in value by one each time it

receives the continuation signal, and when the fan-in reaches zero, the thread becomes ready to be executed. Once the thread execution is triggered, nothing can interfere with its execution until the execution terminates.

6.2.2 Thread and Instance

In order to realize the continuation-based exclusive multithread execution in the Fuce architecture, its programming model is defined in terms of the function and the thread. In general, a function is composed of several threads, and has a function instance when activated. The function instance is used as its execution environment in the thread execution. The function instance has the thread program codes and data area of the activated function. Threads in the same function share its function instance.

Features of the thread are summarized as

- The thread has its synchronization value. Its initial value is set to its fan-in value. When the continuation signal is issued by the predecessor thread, the synchronization value is decreased, and when it reaches zero the thread is ready to execute. The continuation signal is issued when the continuation instruction is executed in the predecessor thread.
- The synchronization with other threads is decided only by the issue of the continuation signal delivered by the threads.
- The thread continues its execution without interruption until the thread termination instruction is executed. No thread has a busy wait state during execution.

6.3 Thread Programming Technique

The exclusive multithread execution model easily achieves a multithread programming technique which would be difficult in the conventional sequential execution model that is convenient for serial programs. The thread programming technique extracts the parallelism that exists inside the program as much as possible, and achieves more efficient multithread execution control. We show concretely the demand-driven concept and the data-driven concept by the continuation-based multithreading control. In addition, we show thread pipelining that extracts the pipeline parallelism.

6.3.1 Data-Driven Execution

For data-driven control, the continuation point is put in the predecessor thread and the predecessor thread continues the execution from this point to the successor thread. Data needed by the successor thread is transferred

from the predecessor thread before the continuation signal is issued. Thus the data-driven computing concept is realized in the continuation-based model. This technique is very effective except for the programs that need mutual exclusion control. We discuss here the mutual exclusion problem in data-driven control and its solution.

For mutual exclusion in the data-driven concept, the thread tries to lock another thread that accesses an exclusive resource, because the thread that accesses the resource has to be executed exclusively. That is, the thread has to be continued selectively from its predecessor threads. In order to control the selective continuation, the test&lock operation is devised in the data-driven method. Figure 6.2 shows an example of mutual exclusion control using the test&lock operation. This example shows the case where two predecessor threads selectively continue to the mutually exclusive thread.

1. The predecessor thread performs the test&lock operation to the successor thread.
2. (a) When the predecessor thread succeeds in locking, the predecessor thread continues to the mutually exclusive thread.

   (b) When the predecessor thread fails to get the lock, the predecessor thread is reactivated and tries to get the lock again.
3. When the successor thread terminates the execution of a critical region, it releases the lock.

We call this technique of using the lock operation for the control of mutual exclusion, the lock operation technique. This lock operation technique is effective in the case where any threads continue to the mutually exclusive thread if they cannot explicitly specify their predecessor threads. This situation will occur when any number of processes will be dynamically created for resource management in the OS.

In continuation-based multithread execution, the thread is executed exclusively without interference. Therefore, when missing resource acquisition,

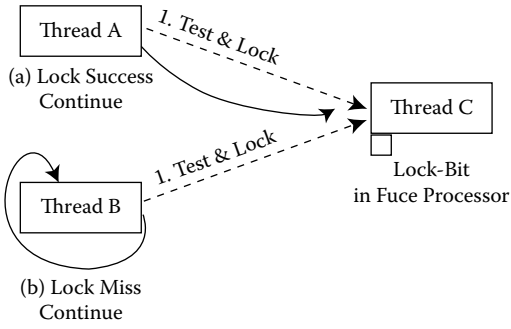


FIGURE 6.2
Lock operation for mutual exclusion.

the thread execution should not be in busy wait but should terminate its execution and reactivate, because deadlock would occur if multiple threads are in busy wait. The problem is that the execution resource is uselessly consumed in repeating the rerunning of the thread that misses for the resource acquisition. Repeated useless reactivation of threads for the test&lock operation would impede other threads in starting execution.

6.3.2 Demand-Driven Execution

In data-driven execution, the continuation point is set in the predecessor thread, and the dependence relation between threads is defined as data-driven continuation. In the demand-driven concept, on the other hand, the continuation point is set in the successor thread, and the continuation relation is defined from the successor thread to the predecessor thread by demand. Figure 6.3 shows the execution of thread A, thread B, and thread C that uses the demand-driven continuation.

Thread B needs the result of thread A. Thread C needs the result of thread B. In order to execute these three threads with the demand-driven continuation, thread C sends the demand for data to thread B, and thread B sends the demand for data to thread A. Afterwards, thread A sends the result data to thread B and continues to thread B. Thread B similarly sends the result data to thread C and continues to thread C.

The demand for the result is called the demand-driven continuation, and the notification of the result data is called the data-driven continuation.

The thread is reactivated when it fails to get the lock of mutual exclusion, and in some cases the reactivation repeats many times in the data-driven concept. In order to exclude the lock, we use the demand-driven method to the mutual exclusion. Figure 6.4 shows an example of mutual exclusion control by demand-driven. This example shows the case where two predecessor threads continue to one successor thread.

1. The successor thread continues to one of its predecessor threads with demand, and terminates.
2. The demanded predecessor thread sends the result data to the successor thread, and continues to the successor thread.
3. The successor thread executes with the result data, then continues to another predecessor thread with demand, and terminates.

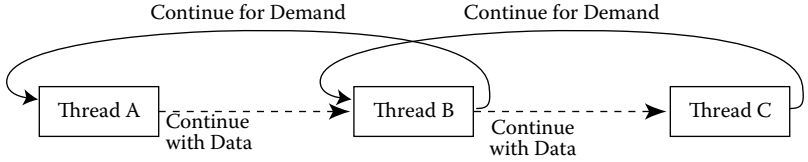


FIGURE 6.3
Continuation for demand.

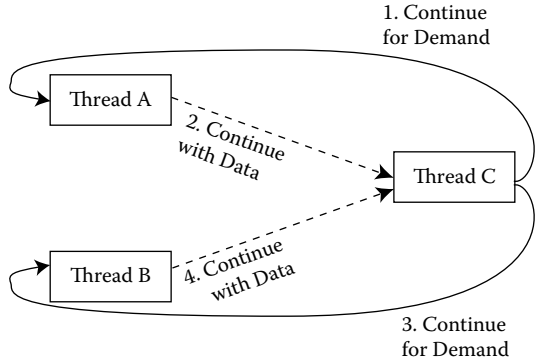


FIGURE 6.4
Continuation for mutual exclusion.

We call this technique to use the demand for control of the exclusive threads activation, the demand-driven method. This method needs no test&lock operations and eliminates the useless reactivation of the thread. We can use this demand-driven technique if the predecessor threads are predetermined and known by the exclusive successor thread as shown by the arrows in Figure 6.4. In a producer and consumer processing scheme like stream processing, use of this technique will achieve efficient mutual exclusion control.

6.3.3 Thread Pipelining

Thread pipelining controls the thread executions so that each thread executes concurrently in pipelined fashion. Figure 6.5 depicts the thread pipelining.

In Figure 6.5, thread A is the predecessor that continues to thread B, and thread B is the successor thread continued from thread A. Thread A passes

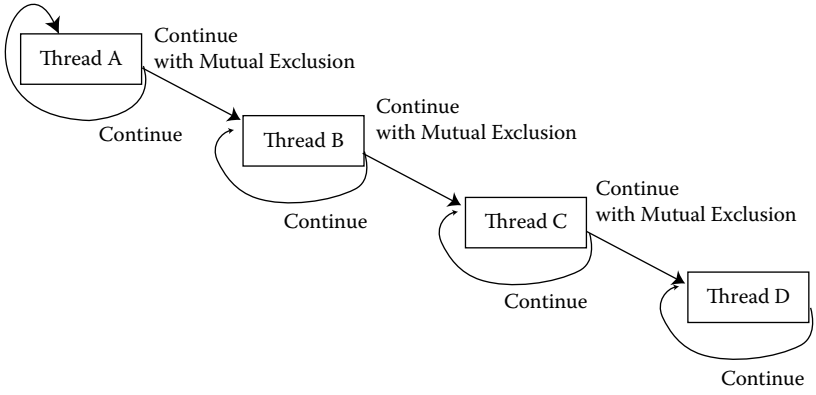


FIGURE 6.5
Thread pipelining.

the result and continues to thread B. Then thread A reactivates itself to generate the next result. At the same time, the continued thread B initiates the computation with the passed data. At this time, thread A and thread B can be executed in parallel. In the same way, thread pipelining is exploited between threads B and C, and between threads C and D. Thus, thread pipelining exploits the thread level parallelism because thread pipelining can execute the predecessor thread and the successor thread in parallel. In this scheme, thread pipelining makes the best use of continuation-based multithread execution without a memory buffer.

6.4 Fuce Processor

The objective in designing the Fuce processor is to fuse the intraprocessor computation and interprocessor communication. Not only user-level programs, but even the OS kernel program including external interrupt handling code and communication processing code are assumed to be composed of sets of threads, and the Fuce processor is designed to execute the multiple threads in parallel and concurrently. Figure 6.6 shows an overview of the Fuce processor. The Fuce processor mainly implements multiple thread execution units, register files, and the thread activation controller. In addition, assuming the advance of semiconductor technology, the Fuce processor has on-chip memory to reduce memory access latency. The processor executes multiple exclusive threads in parallel with the multiple thread execution units.

6.4.1 Thread Execution Unit

The thread execution unit (TEU) executes instructions of a thread. The Fuce processor has multiple TEUs, each of which executes an activated thread

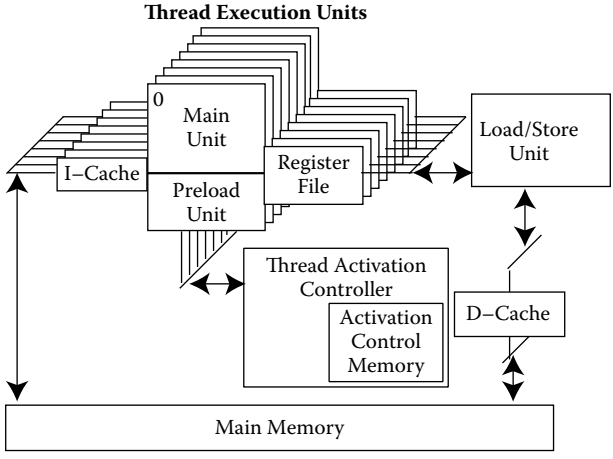


FIGURE 6.6
Fuce processor.

exclusively. The TEU consists of a main unit and a preload unit. The main unit is a very simple RISC processor that can issue the thread control instructions concerning the continuation. The preload unit is a subset of the main unit and mainly executes load instructions to set up the execution context.

6.4.2 Register Files

There is a set of two register files. One is the current register file, which is used by the main unit. The other is the alternate register file, which is used by the preload unit. When the TEU switches thread execution from the current one to the next one, the TEU switches the roles of the current register file and the alternate register file.

The Fuce processor achieves preloading of thread context using the preload unit and register files [8]. This allows the Fuce processor to hide memory access latency and to achieve fast thread-context switching. Figure 6.7 shows an overview of thread context preloading.

While the main unit is executing a thread using the current register file in the foreground, the preload unit starts to execute the load instructions of another thread using the alternate register file and transfers data from the main memory to the alternate register file in the background. Here, we assume that the programmer or compiler has scheduled instructions so that all load instructions are arranged to the forepart of a thread and the other instructions construct the rest of the thread. The execution of the rest is done by the main unit. At this time, the current register file will become the alternate register file when the main unit finishes executing the thread. Also the alternate register file will become the current register file. Therefore, the main unit and the preload unit can start to execute different exclusive threads independently using current register files and

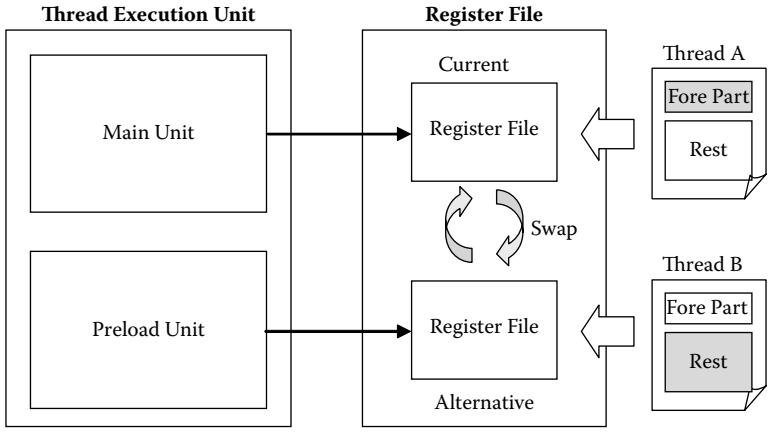


FIGURE 6.7
Preloading of thread context.

alternate register files. The preload unit executes the forepart of a thread, and the main unit executes the rest of the thread.

6.4.3 Thread Activation Controller

The thread activation controller (TAC) is the core component of the Fuce processor. The TAC controls all successor threads and exclusive multithreading.

The TAC has an activation control memory (ACM), which has information about function instances. Figure 6.8 shows an overview of the TAC. The structure of the ACM is similar to the paging system used in virtual memory in a typical OS. Each page in the activation control memory is associated with a function instance, and information about all threads involved in a function is recorded in an ACM page. Sync-count, *fan-in*, code-entry, and lock-bit comprise the necessary thread information for controlling thread execution. The sync-count is the current waiting number for continuation needed to execute its thread. Whenever the thread is continued, the sync-count is reduced by one. The *fan-in* is the *fan-in* value of the thread. The code-entry is the pointer to the thread code. The lock-bit is used for mutual exclusion. Every thread has a thread ID, which consists of a page number and an offset. The page number selects the function instance, and the offset selects thread-entry in a function instance.

A hardware queue, called a ready-queue, is implemented inside the TAC to enable concurrent operation between the TAC and the TEUs. When a TEU

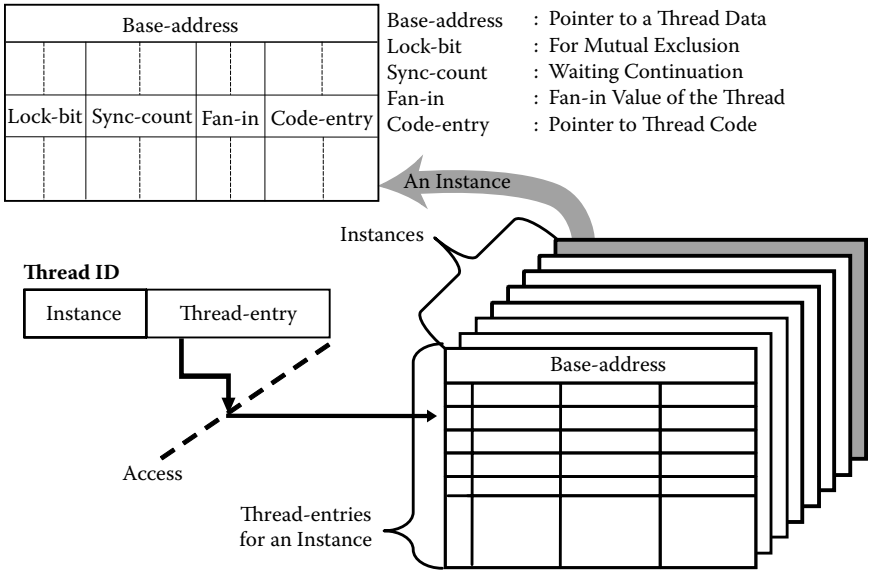


FIGURE 6.8
Activation control memory.

finishes the execution of a thread and its alternate register file becomes available, a thread in the ready-queue is allocated to the TEU to start its execution. At the same time, preloading operation starts with the alternate register file.

The Fuce processor manages events in the TAC. The event-handling threads are preregistered in the ACM, and when an event occurs, the event triggers the corresponding event-handling thread by issuing continuation signal. This is done by making the event-handling device in the Fuce processor issue the continuation signal towards the ACM entry of the corresponding event-handling thread. In this way, the Fuce processor unifies the external event handling and the internal computation as continuation by the TAC.

6.5 Implementation on FPGA

The Fuce processor prototype is implemented on the FPGA board, Accverinos B-1 [15], with eight Xilinx XC2V6000 FPGA chips and 16 SDRAM memory modules. Figure 6.9 depicts the mapping of the Fuce prototype processor on the Accverinos B-1.

The external host machine interacts with the Fuce prototype processor through the PCI bus. The host machine throws the Fuce machine codes and data into the FPGA boards. The memory access control unit distributes them

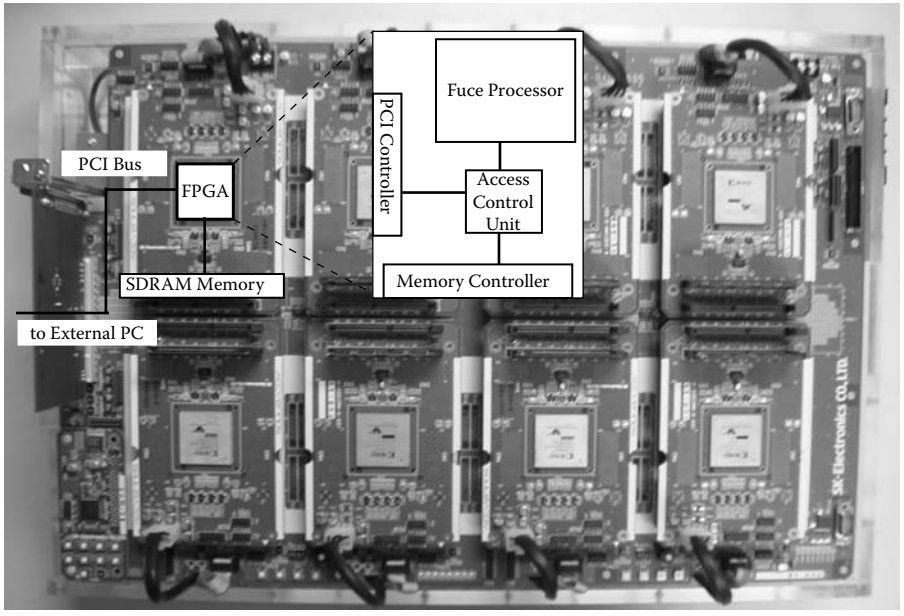


FIGURE 6.9
Accverinos B-1.

TABLE 6.1

Specification of FPGA Board

ACM Size	4 KByte
Frequency of Fuce processor	3 MHz
Frequency of PCI bus	33 MHz
Frequency of SDRAM memory	133 MHz

to the SDRAM memory and the Fuce processor. Table 6.1 shows the specifications of the Accverinos B-1.

The Fuce prototype processor has eight TEUs. The latency of the TAC is one cycle, and memory access latency is variable. Note that the processor in current implementation has one Kbyte instruction cache in each TEU, but it has no data cache. See Tables 6.2 and 6.3.

6.5.1 Hardware Cost of the Fuce Processor

The number of logic gates was calculated to evaluate the cost of the Fuce processor. The Fuce processor was written in VHDL and the FPGA circuit was synthesized using Synplify Pro.

Table 6.4 shows the hardware cost for the Fuce processor. Note that no data cache or memory is included in this calculation. Also, note that the integer multiplier, integer divider, and the floating-point arithmetic unit are not implemented in the current Fuce processor prototype. The hardware cost required for the logic part is 150,000 logic gates. If one FPGA gate is composed of 24 transistors, the Fuce processor requires 3.6 million transistors. The logic circuit of the TAC requires about 360,000 transistors. As this data shows, the circuit size for the thread management is very small. The circuit cost for the thread management unit including the TAC will be lightly affected even if the arithmetic and logic unit gets more complicated.

By the way, the logic part of the Pentium 4 is about 24 million transistors. Compared to the Pentium 4, the circuit scale of the Fuce processor is very small. When the integer multiplier, integer divider, and the floating-point

TABLE 6.2

Specification of Thread Execution Unit

General registers	2 sets * 32 registers * 32 bits
Instruction cache	1 Kbyte
Pipeline structure	5 stages
Instruction issue	2 instructions per clock cycle
Main unit	1 instruction per clock cycle
Preload unit	1 instruction per clock cycle

TABLE 6.3

Specification of Fuce Processor

Number of TEUs	8
Memory size	1 Mbyte
Data cache	None
Instruction cache	8 Kbyte
Memory access latency	20, 60, and 100 clock cycles
TAC's latency	1 clock cycle

unit are implemented in the Fuce processor, the circuit cost will increase. But, the Fuce processor with a small-scale thread management circuit can execute eight threads in parallel, whereas the Pentium 4 can execute only two threads at the same time. In the Fuce processor, every thread execution is triggered by an event, and this makes the processor structure very simple. Therefore, a comparatively small size hardware is required for the TEU.

6.5.2 Simulation Result

Performance of the Fuce processor was evaluated by software simulation. The Fuce processor was described by VHDL and runs on the ModelSim HDL simulator. We evaluated the concurrency performance of the Fuce processor and thread pipelining effect by running several benchmark programs on the simulator. As the Fuce processor on this simulator has only the integer execution circuit, the floating-point execution circuit is imitated by adding the NOP execution cycles. The simulator imitates the multiplication and the division of integer arithmetic with only one cycle.

Quick Sort, Merge Sort, 8-Queen, and Fast Fourier Transform were used as benchmark programs. The Quick Sort program sorts 7000 data, the Merge Sort program sorts 4096 data, and 8-Queen program searches for all solutions. The Fast Fourier Transform program processes 4096 elements. The benchmark programs were written with Fuce assembler language. Quick Sort, Merge

TABLE 6.4

Amount of Gates of Fuce Processor

Module	**FPGA Gates**
TEUs (×8)	124,048
Load/store unit	5,555
TAC	9,103
Etc.	11,733
Overall	150,448

Sort, 8-Queen, and Fast Fourier Transform were chosen because they have high concurrency. These programs are suitable for evaluating the performanceof concurrent execution. Quick Sort, Merge Sort, and 8-Queen programs exploit very high instance-level parallelism, and the Fast Fourier Transform program exploits very high parallelism in both instance-level and data-level. These programs are written by the well-known standard algorithms.

Furthermore, the performance of the thread pipelining was evaluated using Quick Sort and Merge Sort programs. Two kinds of mutual exclusions, the data-driven technique and the demand-driven technique, were applied to the thread pipelining. The data-driven program uses the thread pipelining with the data-driven technique, and the demand-driven program uses the thread pipelining with the demand-driven technique. These programs show that the thread pipelining technique exploits thread-level parallelism more than the programs written in well-known algorithms.

Performance was evaluated for various values of the number of TEUs and memory access latency.

Table 6.5 and Table 6.6 show the performance improvement to the increase in the number of TEUs. In the Quick Sort and Merge Sort programs, the performance is affected by the increase in the memory access latency. In the 8-Queen and the Fast Fourier Transform programs, the speedup is roughly linear to the increase in the number of TEUs. The Tables also show that the speedup ratio is roughly linear to the increase in memory access latency. From this data it is said that the Fuce processor exploits performance enough for concurrent programs.

Table 6.7 and Table 6.8 show the performance improvement to the increase in the number of TEUs in the thread pipelining Quick Sort and Merge Sort programs. Figure 6.10 and Figure 6.11 show the clock cycles to the number of TEUs in the thread pipelining Quick Sort and Merge Sort programs. (In these Figures, the number next to the program name is the memory access latency.) In the thread pipelining Quick Sort and Merge Sort programs, the performance is improved more for the increase of memory access latency.

TABLE 6.5

Speedup Ratio in the Well-Known Quick Sort and Merge Sort Methods (Normalized with Each One TEU)

	Quick Sort			Merge Sort		
	Memory Access Latency					
# of TEUs	20 (Clock Cycles)	60	100	20	60	100
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.79	1.73	1.75	1.73	1.66	1.63
4	2.88	2.68	2.70	2.61	2.39	2.28
6	3.50	3.18	3.21	3.08	2.75	2.58
8	3.86	3.46	3.48	3.37	2.96	2.76

TABLE 6.6

Speedup Ratio in the Well-Known 8-Queen and Fast Fourier Transform Methods (Normalized with Each One TEU)

# of TEUs	8-Queen			Fast Fourier Transform		
	Memory Access Latency					
	20 (Clock Cycles)	60	100	20	60	100
1	1.00	1.00	1.00	1.00	1.00	1.00
2	1.99	1.99	1.99	2.00	2.00	2.00
4	3.96	3.97	3.98	3.95	3.94	3.98
6	5.91	5.94	5.95	5.82	5.82	5.86
8	7.80	7.85	7.89	7.64	7.68	7.74

TABLE 6.7

Speedup Ratio in the Thread Pipelining Quick Sort (Normalized with Each One TEU)

# of TEUs	Standard Method			Data-Driven			Demand-Driven		
	Memory Access Latency								
	20 (Clock Cycles)	60	100	20	60	100	20	60	100
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.79	1.73	1.75	1.96	1.98	1.99	1.98	1.99	1.99
4	2.88	2.68	2.70	3.85	3.93	3.93	3.45	3.18	3.05
6	3.50	3.18	3.21	5.42	5.22	4.86	3.70	3.31	3.15
8	3.86	3.46	3.48	6.00	5.42	4.97	3.89	3.36	3.19

TABLE 6.8

Speedup Ratio in the Thread Pipelining Merge Sort (Normalized with Each One TEU)

# of TEUs	Standard Method			Data-Driven			Demand-Driven		
	Memory Access Latency								
	20 (Clock Cycles)	60	100	20	60	100	20	60	100
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	1.73	1.66	1.63	1.99	1.99	1.99	1.97	1.94	1.91
4	2.61	2.39	2.28	3.80	3.90	3.93	3.13	2.92	2.72
6	3.08	2.75	2.58	5.47	5.68	5.75	3.78	3.40	3.12
8	3.37	2.96	2.76	6.75	7.39	7.57	3.94	3.53	3.22

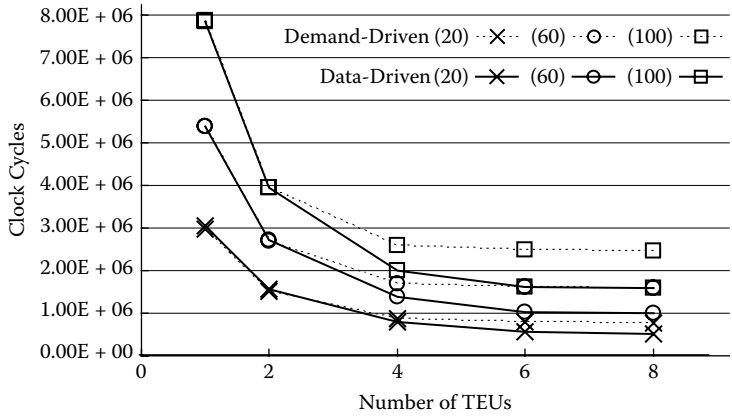


FIGURE 6.10
Clock cycles of thread pipelining quick sort.

From Table 6.7 and Figure 6.10, the data-driven method extracts parallelism more than the demand-driven one in the Quick Sort program. In the data-driven method, the lock operation seldom fails, and the extracted parallelism saliently improves the execution performance. On the other hand, although the demand-driven method excludes all locks, its performance improvement is not as explicit compared with the data-driven method. The performance in the data-driven method improves linearly for one TEU to four TEUs for all of the memory access latency, because the lock-miss decreases with the increase in the memory access latency in the data-driven method. In eight TEUs, the performance improvement in the demand-driven method is higher than

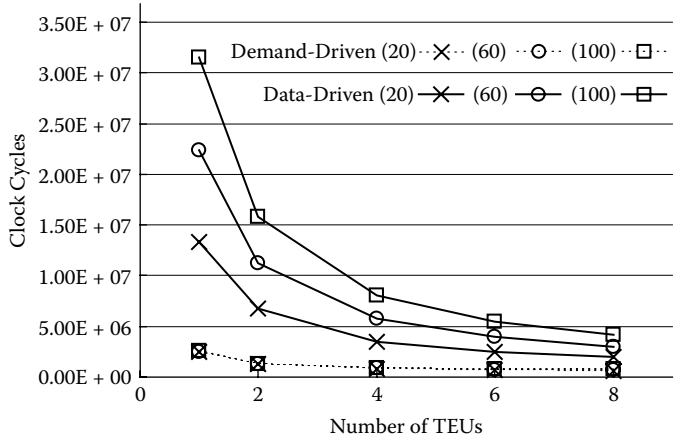


FIGURE 6.11
Clock cycles of thread pipelining merge sort.

the well-known method for one TEU to six TEUs, whereas the performance improvement in the demand-driven method is at the same level as the well-known method. This is because the thread pipelining extracts the parallelism in earlier stages of computation. In the data-driven Quick Sort program, the lock-miss decreases when the memory access latency increases, and the data-driven method effectively exploits the parallelism. From Figure 6.7, the data-driven program extracts more parallelism than the demand-driven program. Thereby, as Figure 6.10 shows, the data-driven method achieves higher performance than the demand-driven one for the increase in memory access latency. In the Quick Sort program, the demand-driven method cannot preload the thread context and therefore cannot use multiple TEUs effectively.

Table 6.8 shows that the data-driven method achieves a linear speedup to the increase in the number of TEUs in the Merge Sort program. However, as Figure 6.11 shows, the data-driven method consumes more execution clock cycles than the demand-driven one. The data-driven method fails 98% of the lock operations and repeats its thread execution to get the lock. Thus, the thread execution repeats uselessly and it consumes many more execution clock cycles. On the other hand, the demand-driven method never uses the test & lock operations and improves the throughput. The demand-driven method exploits the parallelism even though the thread needs two continuations for the demand and the computation result.

Figure 6.11 shows that the data-driven method consumes more execution time than the demand-driven one. This is because the lock-miss is caused by the feature of the Merge Sort program in which most of the lock operations fail to get the lock in data-driven execution. And repeated thread execution for the lock competes with other thread executions to start their execution. Thus, the demand-driven method exploits the parallelism more in the thread pipelining, and it improves the performance.

If programs can exploit parallelism enough, higher speedup will be achieved on the same number of TEUs even if the memory access latency increases. For example, the well-known 8-Queen and Fast Fourier Transform programs and the data-driven Merge Sort program, which have enough parallelism, achieve higher speedup on the same number of TEUs than other methods. The reason is that the memory access latency is hidden by the preloading of thread context.

6.6 Conclusion

This chapter described the processor architecture, named Fuce, which supports thread-level parallel computation. The Fuce architecture is designed to fuse intraprocessor computation and interprocessor communication. The basic programming model of the Fuce architecture is the continuation-based multithreading. Then, the chapter discussed continuation-based thread

programming, Fuce processor construction, and evaluation of the Fuce processor.

This chapter showed that the Fuce processor exploits parallelism in concurrent execution of multiple threads and invents the stream-processing performance extracted by thread pipelining. It was shown that the Fuce processor improves its performance linearly to the increase in the number of TEUs in concurrent processing. The thread pipelining also extracts the parallelism as much as possible from stream processing style programs.

The problem of the Fuce processor, although the problem is common to all parallel processing, is that it is difficult to make use of the locality of data. We have to develop a method for extracting the data locality in parallel processing, and a method of thread allocation and activation to control the effective use of the cache memory.

In the next step, we will implement the OS kernel mechanism of the Fuce processor on the FPGA board and will evaluate processor performance using more practical benchmark programs. For more detailed evaluation of stream processing, benchmark programs such as multimedia processing will be considered.

Acknowledgments

This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Scientific Research (A), No.1520002, 2003.

References

[1] Amamiya, M., "A new parallel graph reduction model and its machine architecture." *Data Flow Computing: Theory and Practice*, Ablex, pp. 445–467 (1991).

[2] Amamiya, M., Taniguchi, H., and Matsuzaki, T., "An architecture of fusing communication and execution for global distributed processing." *Parallel Processing Letters*, Vol. 11, No. 1, pp. 7–24 (2001).

[3] Wall, D. W., "Limits of instruction-level parallelism." *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems*, pp. 176–188 (1991).

[4] Roh, L., and Najjar, W. A., "Analysis of communications and overhead reduction in multithreaded execution." *In Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques* (1995).

[5] Hammond, L., Hubbert, B. A., Siu, M., Prabhu, M. K., Chen, M., and Olukotun, K., "The Stanford Hydra CMP." *IEEE Micro*, Vol. 20, No. 2, pp. 71–84 (2000).

[6] Kavi, K. M., Youn, H. Y., and Hurson, A. R., " PL/PS: A non-blocking multithreaded architecture with decoupled memory and pipelines." *In Proceedings of the Fifth International Conference on Advanced Computing (ADCOMP '97)* (1997).

[7] Lo, J. L., Eggers, S. J., Emer, J. S., Levy, H. M., Stamm, R. L., and Tullsen, D. M., "Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading." *ACM Transactions on Computer Systems*, Vol. 15, No. 3, pp. 322–354 (1997).

[8] Matsuzaki, T., Tomiyasu, H., and Amamiya, M., "Basic mechanisms of thread control for on-chip-memory multithreading processor." *In Proceedings of the Fifth Workshop on Multithreaded Execution, Architecture and Compilation (MTEAC-5)*, pp. 43–50, (2001).

[9] Marr, D. T., Binns, F., Hill, D. L., Hinton, G., Koufaty, D. A., Miller, J. A., and Upton, M., "Hyper-threading technology architecture and microarchitecture." A hypertext history, *Intel Technology Journal*, 6,1 (online journal) (2002).

[10] Nishi, N. et al., "A 1GIPS 1W single-chip tightly-coupled four-way multiprocessor with architecture support for multiple control flow execution." *Proceedings ISSCC2000* (2000).

[11] Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S.W., and Moore, C.R., "Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture." *In Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 422–433 (2003).

[12] Ungerer, T., Robic, B., and Silc, J., " A survey of processors with explicit multithreading." *ACM Computing Surveys 35*, pp. 29–63 (2003).

[13] Sinharoy, B., Kalla, R. N., Tendler, J. M., Eickemeyer, R. J., and Joyner, J. B., "POWER5 system microarchitecture." *j-IBM-JRD*, Vol. 49, No. 4/5, pp. 505–521 (2005).

[14] Kongetira, P., Aingaran, K., and Olukotun, K., "Niagara: A 32-way multithreaded SPARC processor." *IEEE Micro*, Vol. 25, No. 2, pp. 21–29 (2005).

[15] SK-Electronics. Accverinos B-1., http://www.accverinos.jp/english/pro-km1.html.

7

A Study of a Processor with Dual Thread Execution Modes

Rania Mameesh and Manoj Franklin
University of Maryland

CONTENTS

7.1 Introduction

All major high-performance microprocessor vendors have announced or are already selling chips with two to eight cores. Future generations of these processors will undoubtedly include more cores. Multicore architectures exploit the inherent parallelism present in programs, which is the primary means of increasing processor performance, in addition to decreasing the clock period and the memory latency. Two of the techniques that have been recently proposed for exploiting parallelism in nonnumeric programs are *subordinate threading* and *speculative multithreading*. Both of these techniques use multiple processing cores.

In the subordinate threading technique, one of the processing elements (PEs) executes the *main thread*, whereas the others execute *subordinate threads* (a.k.a. *helper threads*). A subordinate thread works on behalf of the main thread, thereby speeding up the main thread's computation. Subordinate threading techniques have been proposed for tasks such as data prefetching [1–5] and branch outcome precomputation [6, 7]. Quite often, the subordinate thread is a redundant copy of the main thread, but pruned in an appropriate manner to achieve the desired goal.

In the speculative multithreading technique, the compiler or hardware extracts speculative threads from a sequential program, and the processor executes multiple threads in parallel, with the help of multiple processing elements. A speculative thread is spawned before control reaches that thread, and before knowing if its execution is required or not. The use of speculative threads allows aggressive exploitation of thread-level parallelism from programs that are inherently sequential. Examples of speculative threading processors are the multiscalar processor [8] and the trace processor [9].

In this chapter we perform a study on subordinate threading and speculative multithreading. We compare one type of subordinate threading technique (decoupled execution) against one type of speculative multithreading (trace processor). Decoupled architectures were studied in [10–12], in which the program is partitioned into two partitions at a fine granularity. They achieve good performance by exploiting the fine-grain parallelism present between the two partitions. Traditional decoupled architectures partition the instruction stream into a memory access stream and a computation execute stream, such that memory accesses can be done well ahead of when the data is needed by the execute stream, thereby hiding memory access latency. Other ways of partitioning are also possible.

In a trace processor, the program is partitioned (at a slightly coarser level) into traces, each of which is a contiguous sequence of dynamic instructions. A trace executes on each processing element. Processing elements are arranged as a circular queue, in which only the head PE is allowed to commit its instructions. All other processing elements cannot commit instructions until they become the head.

We perform our quantitative comparison of decoupled execution and trace processing using 2-PE processors. In our comparison, we identify characteristics of code regions that are better run using each type of processor (trace processor or decoupled processor). Finally we investigate a technique that exploits the variance within an application by switching between trace processing and decoupled processing in a single processor. Our experimental results show that switching between them provides an average performance improvement of 17% higher than that of decoupled execution and trace processing.

The outline of this chapter is as follows. Section 7.2 discusses the motivation of our work, which is variance in program behavior within an application. We also discuss program characteristics that favor the trace processor or decoupled processor. Section 7.3 discusses a 2-PE hybrid processor that can switch between the trace processing and decoupled processing modes. We present our experimental results in Section 7.4. Section 7.5 discusses related work. We conclude in Section 7.6.

7.2 Motivation: Performance Variance within an Application

We first briefly describe the trace processor and the decoupled processor that we used, followed by a performance study of one of the SPEC2000 benchmarks, *bzip*, using a single-PE processor, a 2-PE trace processor, and a 2-PE decoupled processor. We then compare the trace processor execution against the decoupled processor. The metric we use in our comparison is based on how much each technique manages to overlap computations on each processing element.

7.2.1 Our Implementation of the Trace Processor and Decoupled Processor

The decoupled architecture requires two architecture contexts. The trace processor requires at least two architecture contexts. We only use two architecture contexts for the trace processor because the comparison of trace processor against decoupled processor would be unfair if more than two architecture contexts are used for the trace processor.

7.2.1.1 Trace Processor

Trace processors are organized around traces [9][13]. Traces have fixed sizes, and are formed dynamically by the hardware as the program executes. A trace predictor is used to predict the next trace to execute on the next empty PE. Traces are committed in program order; thus the head PE has the first trace,

and the next trace in program order is fetched by the following PE which in turn will become the head when the head PE commits all its instructions. The trace processor we use is very similar to the one in [9, 14] except for a minor modification. We allow the head PE to fetch up to two traces if it finished fetching the first trace but has not yet finished with committing all its instructions. This is done to reduce the amount of time the fetch unit is idle in the head PE. Hence, our trace processor may contain up to three traces at any time numbered 1, 2, and 3 according to the sequential order of the program. The first trace is executed by the head PE, the second trace is executed by the following PE, and the third trace is fetched by the head PE. The third trace, however, does not modify the state of the head PE while there is a trace ahead of it. When the head PE commits all the instructions of the first trace, it may then allow the third trace to modify its state. The second trace is handled by the second PE (nonhead PE). In our simulations, we used a trace composed of four blocks. The maximum size of a block is seven instructions.

7.2.1.2 Decoupled Processor

The simulated decoupled processor dynamically partitions the active portion of the program into two partitions. One partition—the main thread—is composed of highly predictable branch instructions and computations leading to these branches. The second partition—the subordinate thread—is composed of all other instructions. Computations that lead to a highly predictable branch execute on the main thread and not on the subordinate thread if they are identified as unreferenced writes as in [15]. The main thread executes all store instructions so as to maintain a correct data memory state (the subordinate thread does not write its stores to the second-level data cache). This is not a major overhead, as most of the data cache misses incurred by the subordinate thread are not incurred again by the main thread. The subordinate thread passes all its outcomes and control information to the main thread. The main thread is sped up, by receiving almost perfect branch predictions from the subordinate thread as well as fewer data cache misses (stores and loads executed by the subordinate thread bring the required pages into the dL2 cache). The main thread does not execute instructions that are correctly executed by the subordinate thread (except stores). Also, the main thread does not fetch or decode instructions, because each instruction is fetched, decoded, and executed by the subordinate thread and all this information is sent to the main thread. (Notice that instructions that are not executed by the subordinate thread are still fetched and decoded by the subordinate thread.) If the subordinate thread goes on the wrong path, the main thread squashes it and restarts it.

7.2.2 Analysis of the Performance Variance of Benchmark *bzip*

Figure 7.1 shows how the performance of the SPEC2000 benchmark program *bzip* keeps changing with respect to time when executed on three different execution models: a trace processor, a decoupled processor, and a single-PE

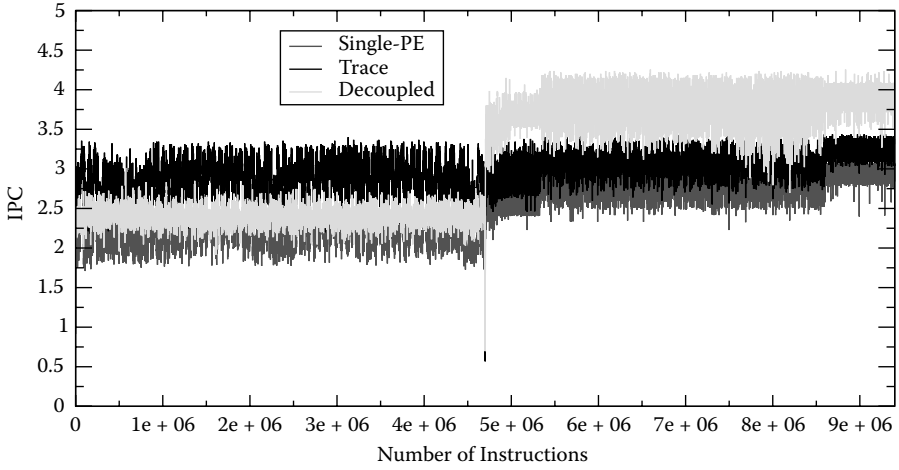


FIGURE 7.1
Performance variance for benchmark *bzip*, when using trace processor, decoupled processor, and single-PE processor.

processor. The performance is measured in terms of the IPC (instructions per cycle), and is plotted for every 1000 dynamic instructions. After skipping the first 1 billion instructions, 9.6 million instructions were simulated. The *x*-axis indicates the number of instructions from 0 to 9.6 million. The *y*-axis indicates the IPC. The first 4.8 million instructions make the first half of Figure 7.1 and the second half is from 4.8 million to 9.6 million instructions. It is clear from the figure that the behavior of the trace processor and the decoupled processor are quite different for *bzip*. More importantly, the behavior in the first half and second half are quite different. In the first half, the trace processor has higher performance, but in the second half the decoupled processor has higher performance. This alternation in performance indicates that within the benchmark *bzip*, thread-level parallelism or fine-grain parallelism alone does not give the highest performance. Rather, the highest performance alternates between both thread-level parallelism, and fine-grain parallelism.

Table 7.1 shows some statistics that explain the above behavior. From the table, we can see that the average number of dynamic branches increases significantly (almost doubles) from the first half of *bzip* to the second half. However, the branch prediction accuracy increases slightly as well. The average number of memory references decreases from the first half to the second half. The dL1 miss rate also decreases from the first half to the second half. These are changes in program characteristics that favor an overall increase in performance.

7.2.3 Analysis of Decoupled Execution versus Trace Execution

For the decoupled processor as well as the trace processor, the IPC in the second half is higher than that in the first half. However, the increase is

TABLE 7.1

Performance Variance for *bzip* Benchmark

	First Half of BZIP			Second Half of BZIP		
	Trace Processor	Decoupled Processor	Single-PE Processor	Trace Processor	Decoupled Processor	Single-PE Processor
IPC	2.8096	2.4107	2.1610	3.0811	3.7527	2.7787
% branch instrs	8.13	8.13	8.13	30.70	30.70	30.70
br. pred. accuracy (%)	94.69	95.50	94.66	97.92	98.46	97.91
% memory instrs	35.73	35.73	35.73	19.57	19.57	19.57
dL1 miss rate	0.0063	0.0037	0.0038	0.0026	0.0013	0.0013
% instrs. executed correctly by subordinate thread	—	76.95	—	—	44.35	—
% instrs. executed correctly by main thread	—	23.04	—	—	55.64	—
% instrs. executed by nonhead PE	33.00	—	—	38.06	—	—
% instrs. executed by head PE	66.99	—	—	61.93	—	—

much more dramatic for the decoupled processor. Let us take a closer look at the reasons for this.

7.2.3.1 Decoupled Processor

In the decoupled processor, the subordinate thread does not execute highly predictable branches and computation leading up to such branches. The main thread executes the instructions not executed by the subordinate thread, in addition to a few classes of instructions that are executed by both. In the first half of *bzip*, the subordinate thread executes about 76.95% of the instructions, and the main thread executes about 23.04% of the instructions. Because of this imbalance, the decoupled processor could not deliver good performance in the first half.[1] In the second half, the subordinate thread executes about 44.36% of the instructions, and the main thread executes about 55.64% of the instructions. Because of this balance among the two PEs, much more performance is obtained in the second half.

7.2.3.2 Trace Processor

The same argument applies to the trace processor also. The number of instructions ready to commit when a processing element (PE) becomes the

[1] A good balance of the workload among the two PEs is important for performance. Even more important is the extent to which the critical instructions; however, that measure is very difficult to quantify.

head PE increases from the first half of *bzip* to the second half. In the first half of *bzip*, the nonhead PE executes 33.00% of the instructions and the head PE executes 66.99%. In the second half, the nonhead PE executes 38.06% and the head PE executes 61.93%. Thus, the work is more equally divided among the PEs in the second half of *bzip* than in the first half, and therefore the IPC increases from the first half of *bzip* to the second half.

7.2.3.3 Trace versus Decoupled

In the first half of *bzip*, the trace processor has a higher IPC than the decoupled processor, because it is more successful in partitioning the work equally among the PEs. For the decoupled processor, the number of instructions executed correctly by the PEs is split 76.95%–23.04%, whereas for the trace processor it is 66.99%–33.00%. The opposite is true in the second half.

7.2.4 Code Region Characteristics

Both the trace processor and the decoupled processor overlap useful computations by partitioning the dynamic code among the two PEs. The way each processor overlaps computations, as we saw, is different. The trace processor divides the program into traces that follow each other in program order, whereas the decoupled processor partitions each block of instructions into two (hence, each trace gets partitioned into two, one of which executes on one PE and the other executes in parallel on the other PE). In this section, we take a closer look at these differences. The examples in Figure 7.2 illustrate how the trace processor and the decoupled processor overlap computations. Through these examples we show which characteristics of code regions make them best suited for the trace processor and which characteristics make them best suited for the decoupled processor.

Figure 7.2a shows three consecutive traces from the first half of *bzip*. Figure 7.2b shows two consecutive traces from the second half of *bzip*. The instructions enclosed in a dark grey box in trace 2 are those that can execute without stalls on the second PE in a trace processor. The instructions enclosed in a white box are those that do not execute on the subordinate thread, so they would execute on the main thread in a decoupled processor. Instructions enclosed in a light gray box can execute in parallel in both the trace processor and the decoupled processor. Note that, in the decoupled processor, instructions can be removed from both trace 1 and trace 2, for execution on the main thread. In the first example (Figure 7.2a) there is a third trace. Trace 3 writes into register 2 and register 2 is not referenced after the last branch of trace 2, therefore it exposes instruction "slt" in trace 2 to be removed.

We can conclude that two characteristics have a detrimental effect on the relative performance of trace processors compared to decoupled processors:[2] large number of unpredictable branches and long data dependency chains.

[2] The discussion presented here is applicable to the form of decoupled processors we have analyzed.

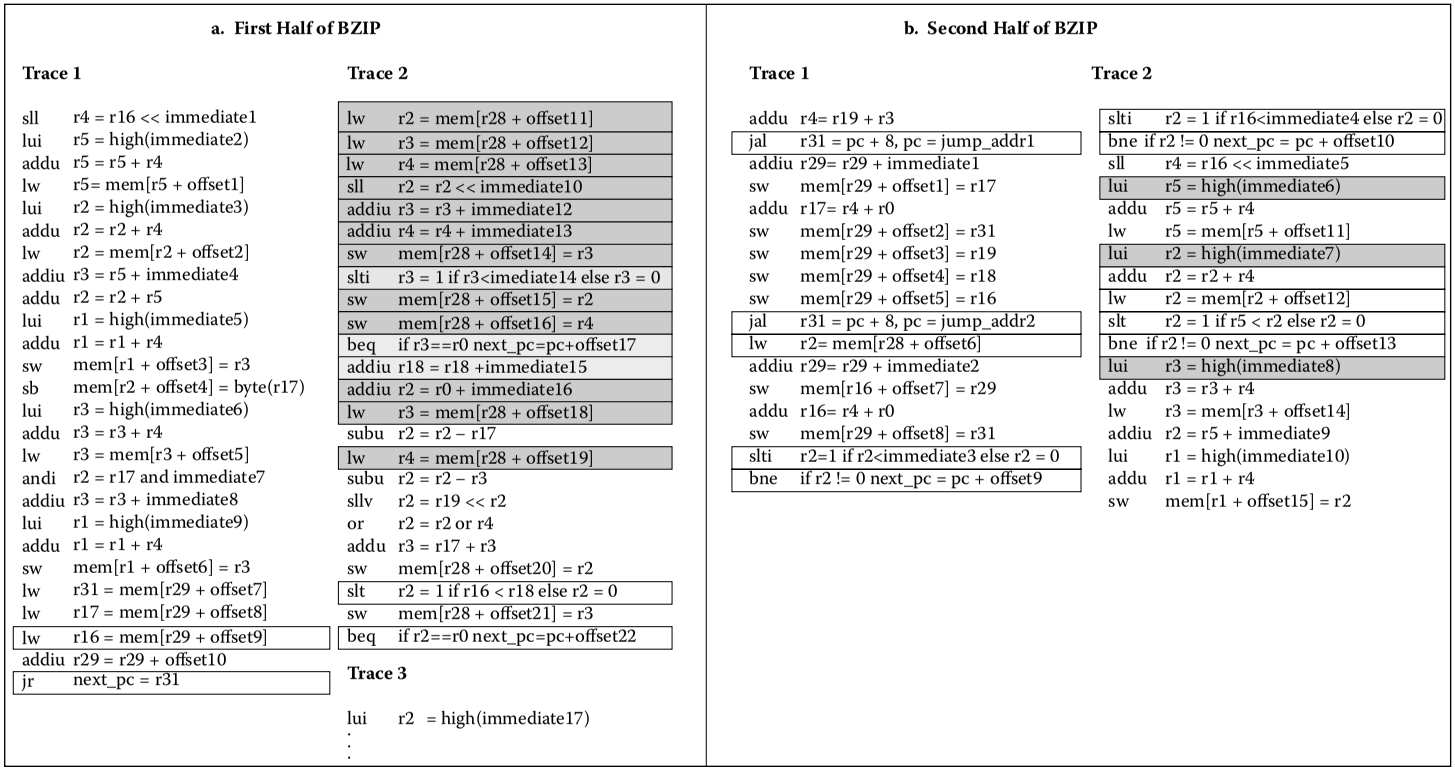


FIGURE 7.2
Examples from benchmark ***bzip***: (a) Three traces from the first half of *bzip*; (b) Two traces from the second half of *bzip*.

7.2.4.1 Shorter Data Dependency Chains and Unpredictable Branches

The example in Figure 7.2a shows that with fewer highly predictable branches, the subordinate thread will end up executing more instructions. With fewer register dependencies among instructions, there will be more parallelism to exploit. In this case, the trace processor does better than the decoupled processor. The number of instructions not executed by the subordinate thread in 7.2a is 7 (unshaded boxes and light grey boxes). The number of instructions that can execute without delays on a second PE in a trace processor is 15 (dark and light grey boxes).

7.2.4.2 Long Data Dependency Chains and Predictable Branches

The example in Figure 7.2b shows that when there are long data dependency chains, the trace processor performs worse than the decoupled processor. The number of instructions not executed by the subordinate thread in this example is 11 (white boxes). The number of instructions in trace 2 that can execute without delays on the second PE of a trace processor is 4 (dark gray boxes).

From these two findings, we can see that although both decoupled processor and trace processor are similar in their goal of overlapping computations to enhance performance, they do it differently. Code regions that are characterized with shorter data dependency chains and predictable branches are best executed by a trace processor. Code regions that are characterized by long data dependency chains and unpredictable branches are best executed by a decoupled processor. Therefore, a hybrid processor that identifies these characteristics dynamically can switch between the decoupled mode and the trace mode when appropriate.

7.3 A Hybrid Processor

7.3.1 Basic Idea

In order to exploit the variance posed within an application during different phases of its execution, we propose a hybrid processor that incorporates both types of threads. For part of the time the processing elements act as a trace processor. For the remaining part, they act as a decoupled processor. Figure 7.3 shows the hardware components of such a hybrid processor. In the trace processor mode, the PEs are called *head* PE and *nonhead* PE. In the decoupled processor mode, the PEs are called *main* PE and *subordinate* PE. The control unit serves the trace processor by deciding the next trace to be fetched in the next PE. For implementing the hybrid function, it also contains a switch that switches between the decoupled and trace processing modes.

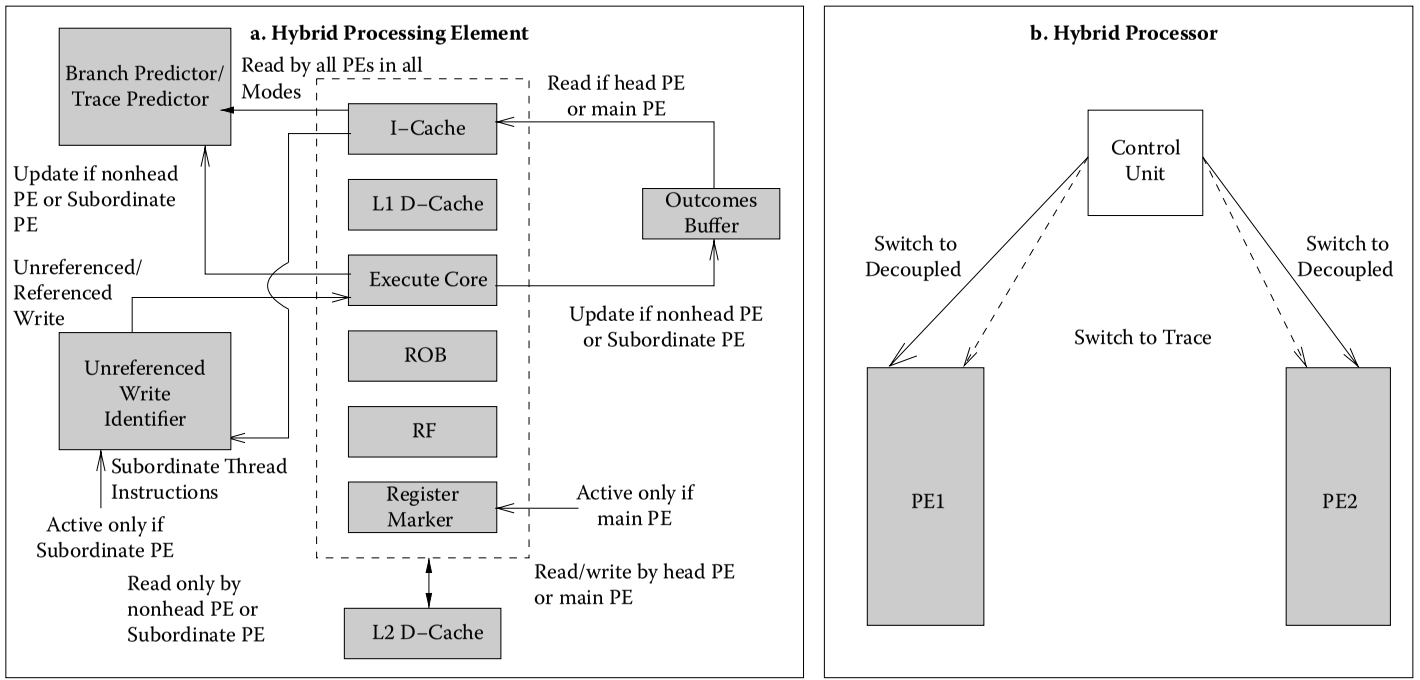


FIGURE 7.3
(a) One processing element of a hybrid processor; (b) Hybrid processor composed of two processing elements and a control unit that switches mode between trace processing and decoupled processing.

While in the decoupled processor mode, the trace predictor continues to be updated for every trace. This is done in order to have an accurate trace predictor, so that upon switching to the trace processor mode, there is no performance loss due to incorrect trace predictions.

The *register marker* and the *unreferenced write identifier* are used while in the decoupled mode to aid in forming the main thread and subordinate thread partitions. The *outcomes buffer* is used to communicate values from the head PE to the nonhead PE in the trace processing mode. It is also used in the decoupled mode to pass the outcomes and decoded information of all instructions executed by the subordinate thread to the main thread.

7.3.2 Minimal Hardware Overhead

The proposed hybrid processor includes minimal hardware requirements above what is required for the trace processor and the decoupled processor. It requires the switch in the control unit as mentioned before. Each hybrid processor PE may have one of four roles at any point of time, a head PE or nonhead PE (as in the trace processor) and a main PE or subordinate PE (as in the decoupled processor). Switching between the roles for any PE is handled by the control unit switch.

7.3.3 Switching Options

We investigate two mechanisms for switching from one execution mode to the other. One of them incurs a lot of penalty (*blunt switching*) and the other has no penalty (*careful switching*). We explain these two in detail.

7.3.3.1 Blunt Switching

In blunt switching the switching is done instantly, potentially throwing away useful work. That is, when the control unit determines that switching modes could lead to better performance, it immediately switches modes. Whatever work done by the nonhead PE (or subordinate PE) is lost in that switching, because the thread being executed in that PE is squashed.

7.3.3.2 Careful Switching

The performance of the hybrid processor under the blunt switching strategy was not very promising, as shown later. After a careful analysis we realized that the amount of work lost during switching was huge. Careful switching was our means to save the lost work. When switching from trace processing to decoupled processing, if there is a trace in the nonhead PE, then it becomes the subordinate thread. When the trace in the head PE commits, it becomes the main PE. There are no penalties incurred when switching from the decoupled mode to the trace processor mode as well. When in the decoupled mode, the subordinate thread sometimes goes on the wrong path.

TABLE 7.2

Microarchitectural Simulation Parameters

Block size	7 instructions
Trace size	4 blocks
Instruction cache	Size/assoc/repl = 16KB/1-way/LRU Line size = 32 instructions Miss penalty = 30 cycles
Data cache	Size/assoc/repl = 16KB/4-way/LRU Line size = 32 instructions Miss penalty = 30 cycles
Dispatch/issue/retire bandwidth	4-way
Trace predictor/branch predictor	Size = 8192 Number of paths = 16 Confidence counters = 16
Subordinate thread recovery delay	5 cycles to startup recovery, 4 register restores per Cycle (total of 64 registers), invalidate all first Level data cache entries of, Total latency = 21 cycles
Switching delays	0 cycles (for blunt switching) Variable (for blunt switching)

That requires recovery with an associated delay (21 cycles in our experiments as shown in Table 7.2). In the careful switching scheme, the actual switching is done only at times of recovery, which will incur 21 cycles anyway even if no switching occurs. The hardware associated with the switching may incur gate delays that cannot be accounted for in our simulation model, as our model is based on a cycle-level simulator.

7.4 Experimental Results

In order to study the potential for a hybrid processor that switches between decoupled processing and trace processing, we developed three cycle-accurate simulators based on the simplescalar toolset [16]. One simulator models the trace processor, another models the decoupled processor, and the third models the hybrid processor that switches between the two execution models. All three simulators include two processing elements. Each PE may issue up to 4 instructions per cycle and may hold up to 32 instructions in its reorder buffer. The level 1 data cache of the subordinate thread is invalidated on recovery from the wrong path.

The microarchitectural parameters we used for the study are shown in Table 7.2. We used the SPEC_INT2000 benchmarks. To get to the interesting portions of the benchmarks, we skipped the first 1 billion instructions for each

benchmark, except for *parser* and *twolf*, for which we skipped the first 500 million instructions. We executed 500 million instructions per benchmark.

We performed two experiments to study the potential of the hybrid technique. In the first experiment we ran the trace processor and the decoupled processor once and gathered the IPC data for every 500 instructions executed and placed the data on files. We then ran the hybrid processor with the gathered data as input files. The hybrid processor simulator checks the IPC values in both files every 500 instructions. If one IPC is higher and doesn't belong to the processor currently run by the hybrid, then the hybrid performs a switch. In the second experiment, we did the same but for every 1000 instructions.

7.4.1 Experiment 1 (Every 500 Instructions)

Table 7.3 shows some statistics for the hybrid execution. We run each benchmark for 500 million instructions, so the maximum number of switching is 1 million. Table 7.3 shows the percentage of switching for each benchmark, the percentage of execution time spent in each processing mode (trace and decoupled), and the number of instructions executed by each processing mode. The average number of times switching occurred over all the benchmarks is 32.32%. The average amount of execution time spent in the decoupled processor mode over all the benchmarks is 50.14%. The average performance of the hybrid processor (blunt switching), hybrid processor (careful switching), the decoupled processor, and the trace processor are plotted against that of the single-PE processor in Figure 7.4. The hybrid with blunt switching has an average performance improvement of 5% higher than that of the decoupled and 6% higher than that of the trace processor. It is clear from the figure that the performance of the

TABLE 7.3

Hybrid Processor Switching Statistics (Every 500 Instructions)

	Hybrid Processor Checks Performance Every 500 Instructions				
	%Times Switched	%Cycles in Decoupled	%Cycles in Trace	%Instructions Done in Decoupled Mode	%Instructions Done in Trace Mode
gzip	31.17	68.19	31.81	69.24	30.76
gcc	30.35	51.54	48.46	56.12	43.88
bzip	23.80	66.85	33.15	66.14	33.86
mcf	27.56	41.07	58.93	41.08	58.92
twolf	47.31	57.63	42.37	58.77	41.23
vortex	27.22	51.09	48.91	56.05	43.95
parser	32.62	52.24	47.76	58.00	42.00
perl	23.15	12.74	87.26	13.23	86.77
vpr	47.70	49.91	50.09	54.77	45.23
Average	32.32	50.14	49.86	52.60	47.40

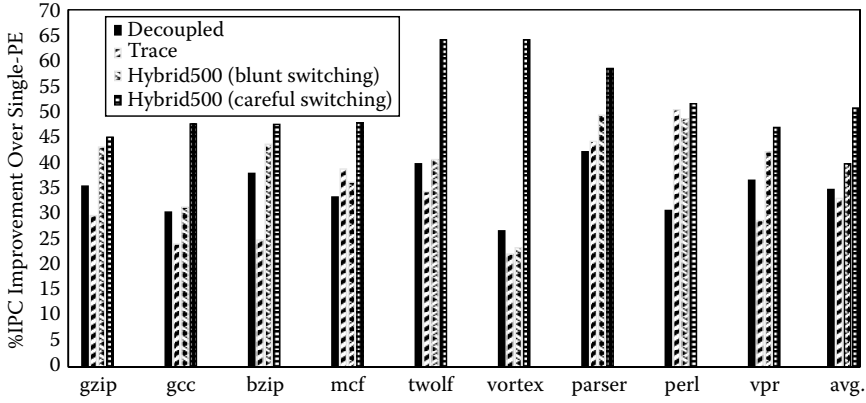


FIGURE 7.4
Percentage of IPC performance for the trace processor, decoupled processor, and the hybrid processor (blunt and careful switching) over the single-PE processor. (Hybrid processor checks IPC every 500 instructions.)

hybrid (careful switching) is far better. Its average performance improvement is higher than that of the trace by 17% and higher than that of the decoupled by 16%. Note that its percentage of IPC improvement is 50% higher than that of both the decoupled and trace processor for benchmark vortex. This is because vortex is one of those benchmarks in which the IPC alternates between highs and lows for the trace processor and the decoupled processor. The highs of the trace processor overlap with the lows of the decoupled processor, as shown in Figure 7.5. The lows (or highs) of the trace processor sometimes overlap with the highs (or lows) of the decoupled processor with a difference of more than 150%, as shown in Figure 7.5.

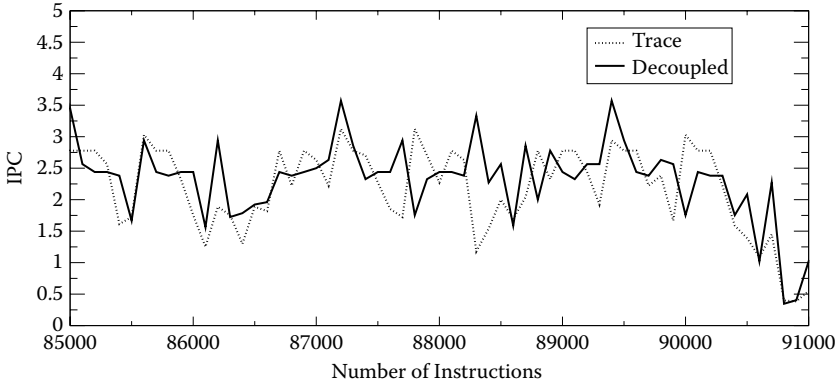


FIGURE 7.5
Overlap of the IPC highs and lows of trace processor and decoupled processor for benchmark vortex.

TABLE 7.4

Hybrid Processor Switching Statistics (Every 1000 Instructions)

	Hybrid Processor Checks Performance Every 1000 Instructions				
	% Times Switched	**% Cycles in Decoupled**	**% Cycles in Trace**	**% Instructions Done in Decoupled Mode**	**% Instructions Done in Trace Mode**
gzip	13.12	74.76	25.24	74.60	25.40
gcc	16.57	54.46	45.54	58.06	41.94
bzip	9.25	67.46	32.54	66.92	33.08
mcf	14.94	37.39	62.61	42.88	57.12
twolf	25.22	60.99	39.01	61.73	38.27
vortex	18.44	49.30	50.70	55.33	44.67
parser	15.11	51.14	48.86	57.28	42.72
perl	5.07	4.90	95.10	5.31	94.69
vpr	23.94	53.85	46.15	57.01	42.99
Average	15.74	50.47	49.53	53.24	46.76

7.4.2 Experiment 2 (Every 1000 Instructions)

Table 7.4 shows the statistics for the hybrid execution, when switching is potentially done after every 1000 instructions. The table shows the percentage of times switching actually occurred for each benchmark, the percentage of execution time spent in each processing mode (trace and decoupled), and the percentage of instructions executed in each processing mode. The average number of times switching occurred over all the benchmarks is 15.74%. The average amount of execution time spent in the decoupled processor mode over all the benchmarks is 50.47%. The average performance of the hybrid processor (blunt switching), hybrid processor (careful switching), the decoupled processor, and the trace processor is plotted against the single-PE processor in Figure 7.6. The hybrid processor with blunt switching has an average performance improvement of 6% higher than that of the decoupled processor and 7% higher than that of the trace processor. It is clear from the figure that the performance of hybrid (careful switching) is again higher than that of blunt switching. Its average performance improvement is higher than that of the trace by 14% and higher than that of the decoupled by 13%. Note that its percentage of IPC improvement is 50% higher than both the decoupled and the trace processor for benchmark vortex for the same reasons as in the first experiment.

7.4.3 Comparison between Experiment 1 and Experiment 2

The difference between experiment 1 and experiment 2 is the granularity at which the hybrid processor may switch between the two execution modes. From the results shown in Figure 7.4 and Figure 7.6, as the granularity

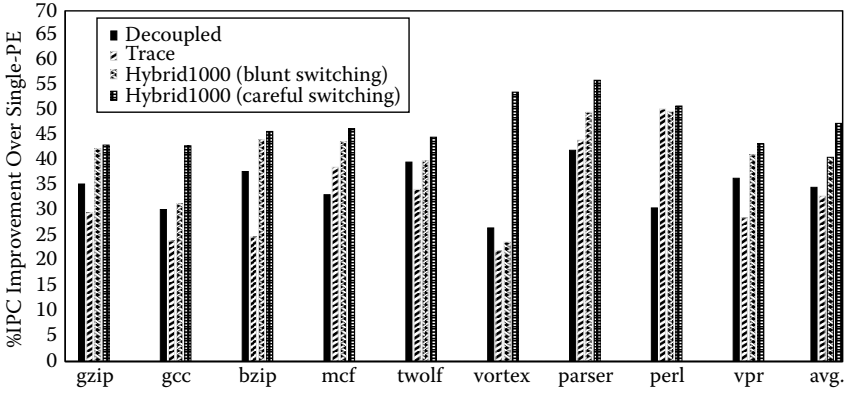


FIGURE 7.6
Percentage of IPC performance for the trace processor, decoupled processor, and the hybrid processor (blunt and careful switching) over the single-PE processor. (Hybrid processor checks IPC every 1000 instructions.)

decreases the performance of hybrid (careful switching) increases. It is also evident from Table 7.3 and Table 7.4 that the percentage of switching is higher for experiment 1 than for experiment 2. These two findings indicate that with smaller granularities, more overlapping of high performance and low performance regions of trace and decoupled is likely to be exploited by the hybrid. Figure 7.7 shows the comparison between the hybrid processor that checks the IPC every 500 instructions versus 1000 instructions. For all the benchmarks, the hybrid (careful switching) does better in experiment 1 than in experiment 2. The hybrid with blunt switching does worse for experiment 1, because of the increased penalties due to increased switching. Note that

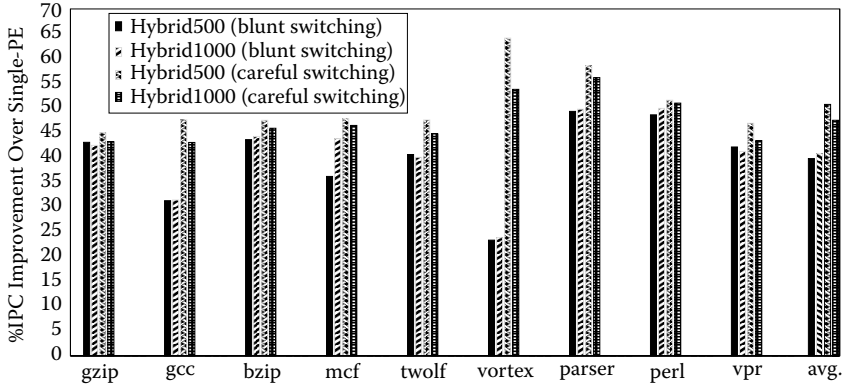


FIGURE 7.7
IPC performance for hybrid processor when checking performance every 500 instructions and every 1000 instructions.

blunt switching may not have a very bad effect if at the time of switching not much work is lost, as evident from some benchmarks such as *vpr* (47% switching).

7.5 Related Work

In [6], a technique is introduced in which the subordinate thread is shortened to be as small as possible. Their technique is called *pruning* and is based on the predictability of values and addresses. The subordinate thread is shortened by pruning computations that are predictable.

The varying behavior of programs was studied in [17] and [18]. In [18] the behavior of programs was classified over their course of execution correlating the behavior among IPC, branch prediction, value prediction, address prediction, cache performance, and reorder buffer occupancy. A program phase was defined in [17] as a set of intervals within a program's execution that have similar behavior, regardless of temporal adjacency.

Techniques to exploit program phases (behavior) were presented in [19] and [20]. In [19] the microarchitectural resources were dynamically tuned to match the program's requirements with regard to power consumption. Program phases were identified dynamically and smaller hardware configurations were used to save power consumption during phases of fewer hardware requirements. In [20] an architecture that can provide significantly higher performance in the same die area than a conventional chip multiprocessor was introduced. It does that by matching the various jobs of a diverse workload to the various cores providing high single-thread performance when thread-level parallelism is low and high throughput when thread-level parallelism is high.

7.6 Conclusion

We performed a comparative study of trace processors and decoupled processors. In our study we identified characteristics of codes that would make a decoupled processor perform better than a trace processor with similar hardware configuration. We also identified code characteristics that would make a trace processor perform better than a decoupled processor. The differences in the code characteristics were evident in some benchmarks, which proved that within an application, different code regions require a different architecture to provide the best performance. We introduced a hybrid processor that exploits the variance within an application such that it executes part of the application using a decoupled processor mode and the remaining

part using a trace processor mode. It does that with the goal of maximizing performance. Our simulations show that our scheme has great potential. It achieves an average performance improvement of 17% higher than what is possible with the decoupled processor.

We plan to extend our work to identify more code region characteristics and use multiple architectures to run them. Dynamic switching between different architectures is also a research topic that we plan to investigate.

References

[1] A. Roth and G. S. Sohi, "Speculative data-driven multithreading," in *Proceedings HPCA-7*, 2001.

[2] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic speculative precomputation," in *Proceedings 34th International Symposium on Microarchitecture*, December 2001.

[3] C. Zilles and G. S. Sohi, "Execution-based-prediction using speculative slices," in *Proceedings ISCA-28*, June 2001.

[4] M. Annavaram, J. Patel, and E. Davidson, "Data prefetching by dependence graph precomputation," in *Proceedings ISCA-28*, June 2001.

[5] J. Dundas and T. Mudge, "Improving data cache performance by pre-executing instructions under a cache miss," in *Proceedings International Conference on Supercomputing*, pp. 68–75, July 1997.

[6] R. Chappell, F. Tseng, A. Yoaz, and Y. Patt, "Difficult-path branch prediction using subordinate microthreads," in *Proceedings 29th International Symposium on Computer Architecture*, May 2002.

[7] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y. Patt, "Simultaneous subordinate microthreading (ssmt)," in *Proceedings ISCA-26*, May 1999.

[8] M. Franklin, *Multiscalar Processors*. Kluwer Academic, 2002.

[9] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, "Trace processors," in *Proceedings 30th Annual Symposium on Microarchitecture (Micro-30)*, pp. 24–34, 1997.

[10] L. Kurian, P. T. Hulina, and L. D. Coraor, "Memory latency effects in decoupled architectures with a single data memory module," in *Proceedings ISCA-19*, pp. 236–245, 1992.

[11] J. E. Smith, S. Weiss, and N. Y. Pang, "A simulation study of decoupled architecture computers," in *IEEE Transactions on Computers*, August 1986.

[12] J.-M. Parcerisa and A. Gonzalez, "Improving latency tolerance of multithreading through decoupling," in *IEEE Transactions on Computers*, 1999.

[13] S. Vajapeyam and T. Mitra, "Improving superscalar instruction dispatch and issue by exploiting dynamic code sequences," in *Proceedings 24th International Symposium on Computer Architecture*, 1997.

[14] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-based next trace prediction," in *Proceedings 30th International Symposium on Microarchitecture*, 1997.

[15] K. Sundaramoorthy, Z. Purser, and E. Rotenburg, "Slipstream processors: Improving both performance and fault tolerance," in *Proceedings ASPLOS-IX*, pp. 257–268, 2000.

[16] D. Burger, T. M. Austin, and S. Bennett, "Evaluating future microprocessors: The simplescalar tool set," Tech. Rep. CS TR-1308, University of Wisconsin Madison, July 1996.

[17] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," in *IEEE Computer*, pp. 84–93, 2003.

[18] T. Sherwood and B. Calder, "Time varying behavior of programs," in *Tech. Rep. No. CS99-630, Dept. of Computer Science and Eng.*, UCSD, August 1999.

[19] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-ISA heterogeneous multi-core architectures for multithreaded workload performance," in *Proceedings 31st International Symposium on Computer Architecture*, June 2004.

[20] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Proceedings 17th International Symposium on Computer Architecture*, 2002.

8

Measurement-Based Power Phase Analysis

W. Lloyd Bircher and Lizy Kurian John
University of Texas at Austin

CONTENTS

8.1 Introduction

Power consumption is a problem for designers at all levels from transistor architects to system programmers. For many it is a first-order design constraint. In recent years, designers have made progress in addressing this problem. At the transistor and microarchitecture levels, these improvements

are applied without knowledge of what type of workloads are present. For example, at the transistor level, the power supply voltage can be reduced when the feature size is reduced. At the microarchitecture level the clock signal can be gated off for idle functional units. However, at the instruction set level or higher, techniques for reducing power consumption are applied dynamically in response to workload variation. Typically, this involves identifying performance-independent phases within a program. These phases allow for the exchange of performance for power savings without a reduction in perceived performance. In the commercial server environment, significant opportunity exists for reducing power consumption through the application of dynamic power management.

Although there has been much research on adaptive power-aware architectures at various granularities, there has not been sufficient study on the actual phase granularities of real programs. Many past studies used simulations. Some used performance counter-based data to extrapolate power in order to identify phases. In this chapter, we study phases based on actual measurement. We measure power samples for several periodicities to study the granularities of phases that exist in commercial servers.

Knowing how power consumption of the whole system is varying is often insufficient to perform adaptations effectively. If one can know how power consumed by a certain resource varies under different conditions, it can be helpful for performing adaptations. In this chapter, we study the power of each subsystem and the variations within it. We use the coefficient of variation (CoV) of power samples for CPU, chipset, memory, I/O, and disk to see the variations in power consumed by each. Homogeneity of power samples from each of these subsystems is presented.

Using 10 KHz instrumentation, we illustrate power phases beyond typical coarse-grain phases used in servers running commercial workloads [1]. By quantifying how much power is typically consumed in a subsystem and for how long the power consumption is stable enough to justify application of power adaptations, effective adaptations can be selected for a workload. This chapter considers power phase durations of 1 ms, 10 ms, 100 ms, and 1000 ms. Phases in this range are applicable to more fine-grain adaptations such as dynamic voltage scaling, throttling, or other microarchitectural approaches.

This chapter makes three primary contributions. The first is a measurement framework for the fine-grain study of subsystem power consumption. By simultaneously measuring the power consumption of multiple subsystems, it is possible to observe complex interactions between subsystems without the need for simulation. Second, using this framework we demonstrate the variation in power consumption at the subsystem level for SPEC CPU, SPECjbb, and dbt-2 workloads. Finally, we characterize power phase behavior of a commercial workload. Unlike previous studies, the characterization includes available power phase duration and amplitude distribution which can be used to predict the amount of detectable phase behavior in a workload.

8.2 Related Work

Existing measurement-based workload power studies of computing systems have been performed at the various levels: microarchitecture [2]–[4], subsystem [5]–[7], or complete system [8]. This study targets the subsystem level and extends previous studies by considering a larger number of subsystems. Also, unlike existing subsystem studies that analyze power on desktop and mobile uniprocessor systems, we consider an enterprise class multiprocessor system running a typical commercial workload.

Past studies performed at the microarchitecture level [2][3] utilize performance-monitoring counters (PMCs) to estimate the contribution to microprocessor power consumption due to the various functional units. These studies only consider uniprocessor power consumption and use scientific workloads rather than commercial. Furthermore, because power is measured through a proxy it is not as accurate as direct measurement. Also, Natarajan [4] performs simulation to analyze power consumption of scientific workloads at the functional unit level.

At the subsystem level, [5]–[7] consider power consumption in three different hardware environments. Bohrer [5] considers the following subsystems in a uniprocessor personal computer: CPU, hard disk, and combined memory and I/O. The workloads represent typical web server functions such as http, financial, and proxy servicing. Our study adds multiprocessors, and considers chipset, memory, and I/O separately. Mahesri and Vardhan [6] perform a subsystem-level power study of a Pentium M laptop. They present average power results for productivity workloads. In contrast, we consider a server-class SMP running commercial and scientific workloads. Feng et al. [7] perform a study on a large clustered system running a scientific workload. As part of a proposed resource management architecture, Chase et al. [8] present power behavior at the system level. To the best of our knowledge, our study is the first to present a power characterization that includes phase duration.

8.2.1 Dynamic Adaptation

Dynamic adaptation is a valuable tool for improving the energy efficiency (instructions/joule) and reliability of computing systems. Unlike static techniques that may limit peak performance in order to reduce average power consumption and increase energy efficiency, dynamic adaptation offers high performance and high efficiency. Dynamic techniques take advantage of a critical feature of modern computing: within an application or a group of simultaneously executing applications, the demand for computing performance is typically variable. During certain phases of execution, an application can reduce its execution time through increased processing performance. In other phases, increases in processing performance will have negligible impact. During these performance-independent phases, it is possible to save power

without reducing the perceived performance of the processor. This chapter seeks to improve the utilization of dynamic adaptations by quantifying the availability of power phases in commercial and scientific workloads. Knowing which workloads offer the greatest availability of distinct phases can assist designers in choosing the best candidates for applying dynamic adaptations.

In addition to increasing efficiency, dynamic adaptations may also be used to increase reliability. Adapting for efficiency usually increases reliability because the component consumes less power and therefore operates at a lower average temperature. However, in some cases adaptations must be applied without concern for performance or efficiency. These adaptations are used to guarantee safe operating conditions at levels from a particular component through large groups of systems. Considering individual components, most current generation microprocessors contain facilities to reduce performance (clock rate) when thermal emergencies occur. A thermal emergency is an elevated die temperature caused by excessive utilization, cooling equipment failure, or high ambient temperature.

At the other extreme, the reliable operation of an entire server rack or computing center may be jeopardized by a highly utilized component. Due to the demand to increase performance in computing centers, many centers are now being designed with systems capable of exceeding thermal and power constraints of the building or room in which they are housed. This is typically not a problem, unless the rare case occurs in which many of the systems are being highly utilized at once. By knowing which workloads have sustained, high levels of utilization, designers can allocate computing resources in a manner that limits or prevents the likelihood of these emergencies.

8.3 Methodology

In this section, we describe our experimental approach composed of power sampling, workload selection, and phase classification.

8.3.1 Power Sampling

For this study, we utilize an existing measurement framework from a previous processor power study [9] and extend it to provide additional functionality required for subsystem level study. The most significant difference between the studies of processor level versus subsystem level is the requirement for simultaneously sampling of multiple power domains. To meet this requirement we chose the IBM x440 server, described in Table 8.1.

By choosing this server, instrumentation is greatly simplified due to the presence of current-sensing resistors on the major subsystem power domains. The current-sensing resistors are included in the server to prevent

TABLE 8.1

IBM x440 SMP Server Parameters

Four Pentium 4 Xeon 2.0 GHz, 512 KB L2 Cache, 2 MB L3 Cache, 400 MHz FSB
32 MB DDR L4 Cache
8 GB PC133 SDRAM Main Memory
Two 32 GB Adaptec Ultra160 10 K SCSI Disks
Fedora Core Linux, kernel 2.6.11, PostgreSQL 8.1.0

over-current conditions in the server's various power domains. This reduces the probability of a short circuit becoming a fire. Although these resistors are used to detect the case where power supply current is greater than a fixed limit, they can be adapted to provide more fine-grain information. This allows study of power phase behavior in the server.

Using the current-sensing resistors, five power domains are considered: CPU, chipset, memory, I/O, and disk. The components of each subsystem are listed in Table 8.2. Due to the complex power requirements of the server chipset used in the x440, it is not possible to directly measure the power consumption of all of the five subsystems. By considering the consistent nature of power consumption in some components, it is possible to infer how much power each subsystem is using. For example, a particular power domain may supply current to two subsystems. If all current delivered to one of those components is effectively constant, that current can be subtracted out to allow observation of the more dynamic remaining components. In addition to the components listed in Table 8.2, additional support circuitry for those subsystems is included in the power measurement such as decoupling capacitors, strapping resistors, and clock generation circuits.

Power consumption for each subsystem (CPU, memory, etc.) can be calculated by measuring the voltage drop across that subsystem's current-sensing

TABLE 8.2

Subsystem Components

Subsystem	Components
CPU	Four Pentium 4 Xeons
Chipset	Memory controllers and processor interface chips
Memory	System memory and L4 cache
I/O	I/O bus chips, SCSI, NIC
Disk	Two 10 K rpm 32 G disks

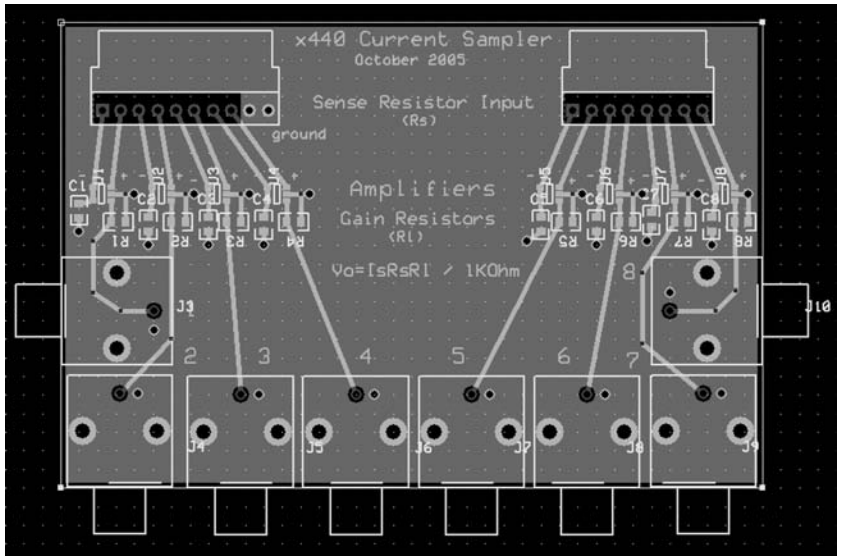


FIGURE 8.1
Current sense amplification PCB.

resistor. In order to limit the loss of power in the sense resistors and to prevent excessive drops in regulated supply voltage, the system designer used a particularly small resistance. Even at maximum power consumption, the corresponding voltage drop is in the tens of millivolts. In order to improve noise immunity and sampling resolution we designed a custom circuit board to amplify the observed signals to levels more appropriate for our measurement environment. The printed circuit board is shown in Figure 8.1. This board provides amplification for eight current measurement channels using the Texas Instruments INA168 current shunt monitor pictured in Figure 8.2. This integrated circuit provides a difference amplifier intended for power instrumentation of portable systems. It provides an output voltage that is directly proportional to the voltage across a current-sensing resistor. The gain of the amplifier is set using a user-selectable resistor. In our case, we chose a gain of 20X. This provides a reasonable voltage level for our data acquisition equipment and allows sampling of signals that vary at rates in excess of 10 KHz. The board also provides BNC-type connecters to allow direct connection to the data acquisition component. The board can be seen as part of the entire measurement environment in Figure 8.3.

This measurement environment is similar to that used in a previous study of uniprocessor power consumption [9]. The main components of the environment are subsystem power sensing, amplification (custom board), data acquisition, and logging. Subsystem power sensing is provided by resistors on board the x440 server. The voltage drop across the resistors is amplified by the custom circuit board. The amplified signals are captured by the National

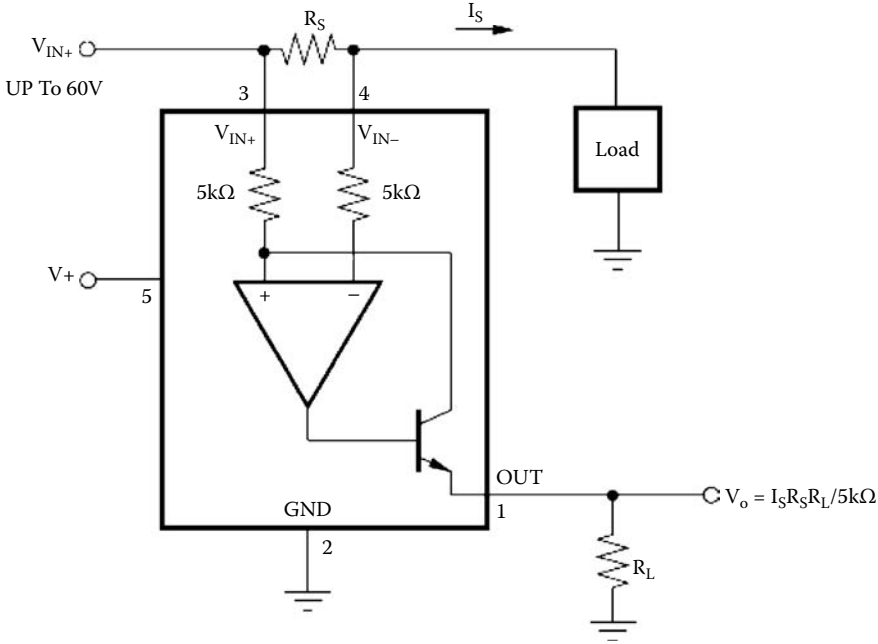


FIGURE 8.2
TI current shunt monitor INA168 [10].

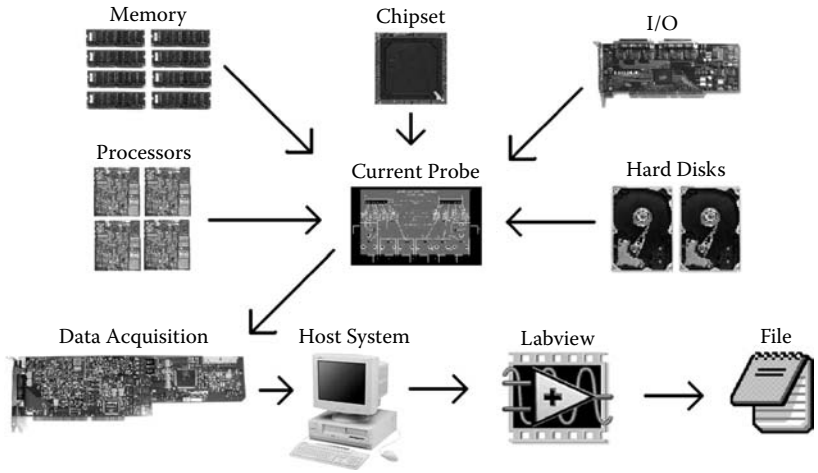


FIGURE 8.3
Power measurement environment.

Instruments AT-MIO-16E-2 data acquisition card. Finally, the host system, running LabVIEW, logs the captured data to a file for offline processing.

Earlier we mentioned the capability of the current-sensing amplifiers to measure signals faster than 10 KHz. This decision was dictated by peak sampling rate of our data acquisition system. Although the AT-MIO-16E-2 data acquisition card is capable of 500 K samples/second, the effective limit is approximately 10 KHz. Two factors contribute to the reduced sampling rate. First, the need to measure eight channels simultaneously reduces the rate by 8X. Second, the host system for the sampling card has insufficient performance to sustain full-speed sampling.

The final component of our measurement environment is the offline processing of log files. Processing is made up of two major parts: amplitude analysis and phase classification. The details of these parts are described in Section 8.3.3.

8.3.2 Workloads

In this section we describe the various benchmarks that are used as workloads used in the study. These benchmarks are intended to be representative of typical server workloads. For commercial workloads, we consider dbt-2 in Section 8.3.2.1 and SPECjbb in Section 8.3.2.2. Section 8.3.2.3 covers the SPEC CPU benchmark which represents scientific workloads. Finally, Section 8.3.2.4 describes the baseline idle workload which is common in server environments.

8.3.2.1 Transaction Processing

For the majority of our analysis we utilize the dbt-2 transaction processing workload from Open Source Development Labs [11] as a representative commercial workload. This workload imitates the TPC-C benchmark. It represents a wholesale parts supplier accepting orders and distributing parts to various sales districts. Results from this workload are presented in terms of new order transactions per minute (NOTPM). They are not intended to be directly comparable to TPC-C results, but they do scale similarly. Dbt-2 dictates the warehouse/client configuration to maintain similarity to TPC-C. Within these requirements it is found that disk space is the primary bottleneck of our x440. The 28 Gbytes of available space on a dedicated disk yielded a 160-warehouse workload. Although higher throughput is possible with more disk space, we were able to obtain 234 NOTPM.

8.3.2.2 SPECjbb 2000

Because our server is disk-bound with respect to server workloads, we include the disk-independent SPECjbb 2000 workload. This workload emulates a three-tiered server-side Java application. The benchmark scales the amount of work to fully utilize the available processing resources.

8.3.2.3 SPEC CPU 2000

Eight SPEC CPU 2000 workloads are analyzed for average power consumption. These do not generate significant I/O traffic, however, they do utilize the CPU and memory subsystems intensively. For each workload, eight instances are run simultaneously. This allows full utilization of the eight available hardware threads (four physical processors with two-way hyperthreading). The power measurements are made after all workloads have passed the initialization phase (reading dataset from disk).

8.3.2.4 Baseline

The baseline workload is the minimum processing required by the operating system when the server is idle. This workload is especially important because it demonstrates the high levels of idle power required to operate a server. For this workload and for all others the following software levels are used: Linux kernel version 2.6.11, Fedora Core 4, gcc version 4.0.0, and Intel FORTRAN compiler 9.1.036.

8.3.3 Phase Classification

Classification of power phases is presented in two ways: amplitude distribution and duration strata. For our purposes, amplitude distribution is defined as a probability distribution of sampling power at a particular amplitude (watts). Duration is the length of a phase in milliseconds.

Amplitude results are presented as probability distributions of all power samples. The samples are stratified into groups with a range equal to one twentieth of the difference between maximum and minimum sampled value. The shape of the distribution can be used to direct power management policies. For example, multimodal power amplitude distributions suggest multiple distinct power phases. In contrast, narrowly distributed power consumption suggests a simpler phase behavior. For the purpose of dynamic phase detection and adaptation, the widely distributed (large standard deviation) or multimodal distributed (multiple peaks) offer the best opportunities, due to the presence of multiple distinct behaviors. Very narrowly distributed power behavior indicates highly homogeneous power consumption and consequently, little opportunity to detect power phases. Finally, the location of the distribution center provides a single, representative power consumption value for the subsystem. Figure 8.4 provides an example of two power amplitude distributions. The high homogeneity distribution indicates that the vast majority of samples are within 5 watts of the 21-watt average. In contrast, the low homogeneity distribution has a much larger variation of nearly 15 watts. Also, two dominant amplitudes are present at 32 watts and 35 watts. This suggests the presence of at least two distinct phases with respect to power amplitude.

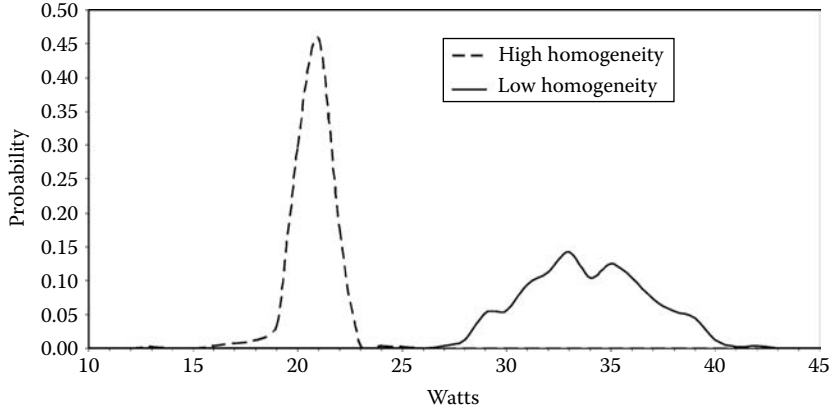


FIGURE 8.4
Amplitude distribution examples.

Presented power phase duration results do not have as fine a granularity (4 levels) as the amplitude results (20 levels). Rather than performing an exhaustive search of all possible phase durations, we instead selected four groups: 1 ms, 10 ms, 100 ms, and 1000 ms. These groups are intended to cover the range of useful but underutilized durations. Phase durations much greater than 1 s are fairly well known and utilized in the server environment [1][12]. No phases less than 1 ms are considered due to the overhead of current software-directed adaptation mechanisms.

Our mechanism for defining a phase is similar to a phase comparison metric used by Lau et al. [13]. In order for a series of samples to be considered a phase, they must have a coefficient of variation less than the limit specified for the experiment. Our results show that a CoV of 0.05 yields representative phases that differ from the sampled data by 3.2% on average.

Phase groupings by duration should be inclusive of all phases greater than or equal to their duration size, yet smaller than the next larger duration group. For example, all phases with durations from 10 ms to 99 ms are placed in the 10-ms group. Also, phases are mutually exclusive. Grouping in the largest possible duration is preferred. For example, although a 100-ms phase is composed of ten 10-ms phases, the 10-ms phases are not placed in the 10-ms group. This approach favors identifying the maximum number of long duration phases, because long phases give the best opportunity for amortizing the cost of identification and power adaptation. We also present results in which samples are allowed to exist within multiple groups. Using the previous example of a 100-ms phase, the phase would be placed in the 10-ms group (ten instances) as well as the 100-ms group.

8.4 Power Analysis

This section presents subsystem power analysis in three forms. First in Section 8.4.1, power traces are presented at varying resolutions to illustrate the need for fine-grain sampling and phase detection. Next, Section 8.4.2 presents probability distributions of power samples. By considering the distribution characteristics, selection of power management strategies is improved. Finally, Section 8.4.3 provides phase duration results based on varying levels of intraphase homogeneity.

8.4.1 Power Traces

In this section power traces of the dbt-2 workload are presented at various sample rates to justify the need for fine-grain sampling and adaptation. Traditional coarse-grain power phases have easily observable behavior. An example of a coarse-grain power phase can be seen in Figure 8.5. For all figures in this section the legend ordering reflects the graph ordering. For example, the top subsystem in the legend is the CPU. Therefore, the top (highest power) subsystem in the graph is the CPU. Similarly, the bottom subsystem in the legend and graph is the chipset. This case demonstrates a server transitioning from being very heavily loaded (0–2000 seconds), servicing a large number of warehouse transactions, to the idle state (2000–5000 seconds), servicing only periodic operating system traffic. These phases are easily detected due the large difference in power consumption. Also, the long phase length reduces the need for frequent sampling. Typically, commercial workload power savings is accomplished by aggressive sleep modes such as standby or hibernation, during the long-term, low-utilization phases [13][1].

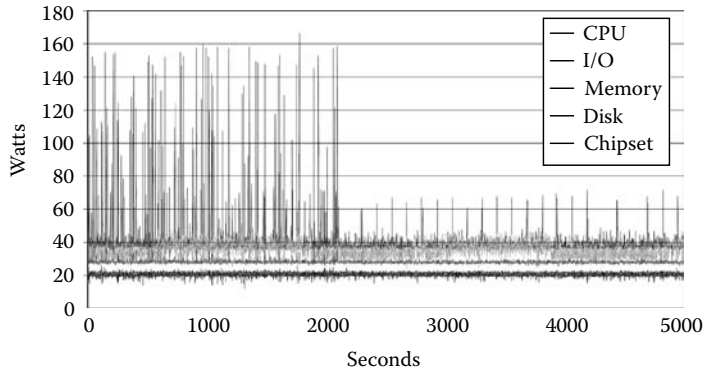


FIGURE 8.5
Power trace, 1-second sampling using dbt-2.

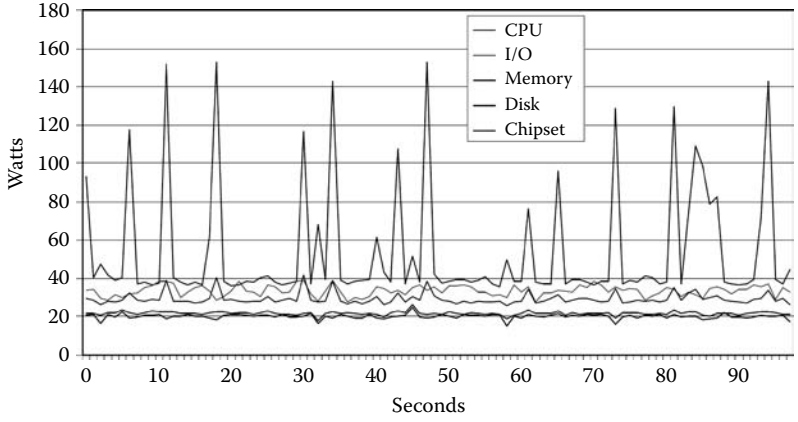


FIGURE 8.6
Power trace, 20-millisecond sampling using dbt-2.

To further reduce power consumption, the shorter, less distinct power phases must be utilized. Figure 8.6 illustrates the presence of numerous distinct power phases once the granularity of sampling is increased. At this resolution it becomes clear that significant fluctuations in power use are occurring. CPU power varies by more than 3X whereas most other subsystems vary from 30–50%. At this level more responsive techniques such as DVFS are appropriate.

Utilizing the extent of our sampling environment, Figure 8.7 shows the presence of very fine-grain power phases when the sampling resolution is increased to 10 KHz. At this level, the large phase magnitude changes are present, but duration appears shorter. Most discernable phases are on the order of milliseconds, with the exception of the two 10-ms phases at 55 and

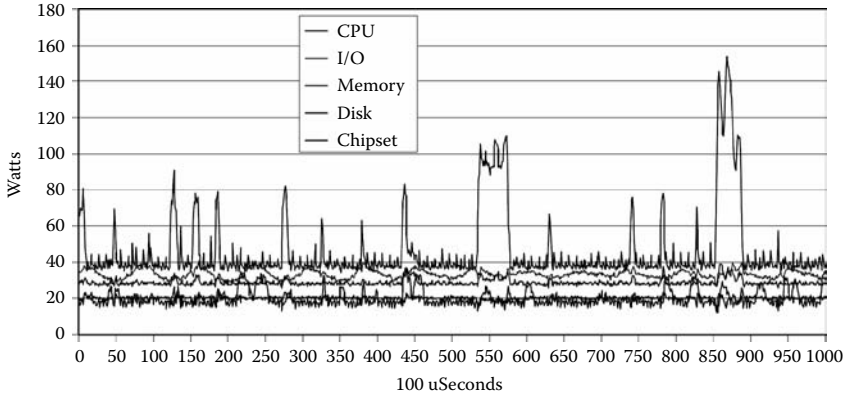


FIGURE 8.7
Power trace, 100-microsecond sampling using dbt-2.

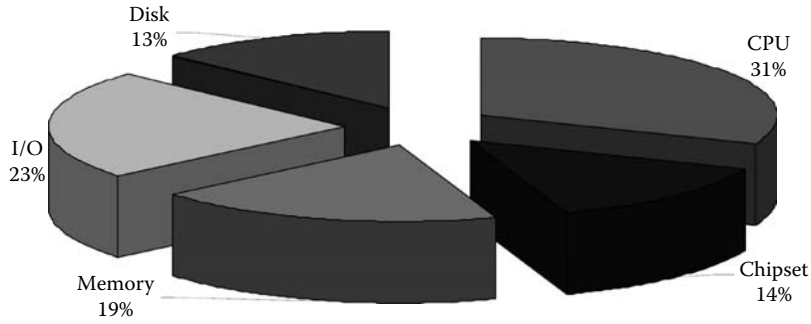


FIGURE 8.8
Server average power consumption using dbt-2.

85 ms. For such short duration changes, it is not reasonable to use DVFS due to the long time required for voltage [14] and frequency transition. A more appropriate technique would be explicit clock gating [15] or ISA directed powerdown of microarchitectural features.

8.4.2 Amplitude

8.4.2.1 Dbt-2

Figure 8.8 shows the average power breakdown for the various subsystems under the dbt-2 workload. Not surprisingly, the CPU subsystem is the dominant power user. However, unlike distributed, scientific [7], and mobile, productivity workloads such as [6], I/O and disk power consumption are significant. Although differences in average subsystem power are large at 138% for disk compared to CPU, the variations within an individual subsystem are even greater. A comparison of subsystem power amplitude distributions is made in Figure 8.9. Note that the CPU distribution is truncated at

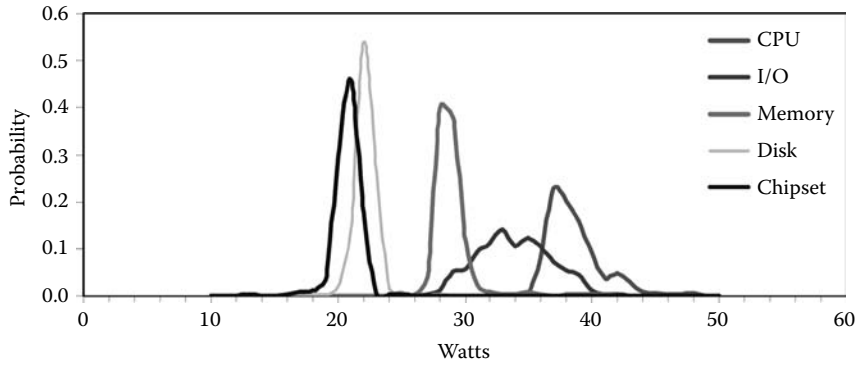


FIGURE 8.9
Subsystem amplitude distributions using dbt-2.

60 watts to prevent obscuring results from the other subsystems. A small number of phases (6.5%) exist above 60 watts and extending to 163 watts. As in Figures 8.5 through 8.7, the legend is ordered with the highest power at the top and lowest at the bottom.

These distributions suggest that there are significant opportunities for phase-based power savings for CPU, I/O, and disk. These subsystems have wider or multimodal distributions. The larger variations in power consumption provide greater opportunity to use runtime detection techniques such as [16][17]. In contrast, chipset and memory have very homogeneous behavior suggesting nearly constant power consumption and less opportunity for phase detection using this workload.

8.4.2.2 SPEC

For the case of SPEC CPU and SPECjbb workloads the results are somewhat different. In Table 8.3 a comparison of average subsystem power consumption is given for all workloads. Compared to the disk-bound dbt-2, these memory-bound and CPU-bound applications show significantly higher CPU and memory power consumption. Although dbt-2 only increases average CPU power by 26% compared to idle, all of these workloads increase average CPU power by more than 250%. For memory, the top three consumers are all floating point workloads. This supports the intuitive conclusion that memory power consumption is correlated to utilization.

The remaining subsystems had little variation from workload to workload. For the disk subsystem this can be explained by two factors. First, most workloads used in this study contain little disk access with the exception of dbt-2. For most others, the initial loading of the working set is the majority of the disk access. Using synthetic workloads targeted at increasing disk

TABLE 8.3

Average Power Consumption (Watts)

	CPU	Chipset	Memory	I/O	Disk
idle	38.4	19.9	28.1	32.9	21.6
gcc	162	20.0	34.2	32.9	21.8
mcf	167	20.0	39.6	32.9	21.9
vortex	175	17.3	35.0	32.9	21.9
art	159	18.7	35.8	33.5	21.9
lucas	135	19.5	46.4	33.5	22.1
mesa	165	16.8	33.9	33.0	21.8
mgrid	146	19.0	45.1	32.9	22.1
wupwise	167	18.8	45.2	33.5	22.1
dbt-2	48.3	19.8	29.0	33.2	21.6
SPECjbb	112	18.7	37.8	32.9	21.9

utilization, we were only able to achieve less than 3% average increase of disk power compared to idle. This is due to the second factor which is a lack of disk power management. Modern hard disks often have the ability to save power during low utilization through low power states and variable speed spindles. However, the disks used in this study do not make use of these power-saving modes. Therefore, disk power consumption is dominated by the power required for platter rotation which can account for almost 80% of max power [18]. For I/O and chipset subsystems, little workload-to-workload variation was observed. In the case of the chipset, offset errors due to aliasing were introduced that affected average power results. As we show in the next section greater variation was found within each workload.

8.4.2.3 Intraworkload Variation

To quantify the extent of available phases within a workload we use the metric coefficient of variation. This metric uses standard deviation to quantify variation in a dataset, and also normalizes the variation to account for differences in average data. Because the subsystems in this study have average power values that differ by nearly an order of magnitude, this metric is most appropriate. Table 8.4 provides a summary of CoV for all workloads.

Compared to the variation in average power among workloads on a given subsystem, the variation within a particular workload is less consistent. Subsystem–workload pairs such as CPU–gcc and memory–SPECjbb have a very large variety of power levels. In contrast disk–art and chipset–mcf have as much as 300X less variation.

The cause for this difference can be attributed to the presence or lack of power management in the various subsystems. The most variable subsystem, the CPU, makes use of explicit clock gating through the instruction set.

TABLE 8.4

Power Consumption Coefficient of Variation

	CPU	Chipset	Memory	I/O	Disk
idle	8.86E-03	4.61E-03	1.17E-03	3.86E-03	1.25E-03
gcc	5.16E-02	1.13E-02	6.90E-02	4.05E-03	2.44E-03
mcf	3.37E-02	8.53E-03	3.60E-02	3.81E-03	1.50E-03
vortex	6.99E-03	4.12E-03	2.06E-02	3.11E-03	7.82E-04
art	2.47E-03	3.66E-03	5.31E-03	3.12E-03	2.51E-04
lucas	1.21E-02	6.34E-03	5.73E-03	3.09E-03	3.25E-04
mesa	6.05E-03	3.49E-03	8.81E-03	3.86E-03	3.85E-04
mgrid	3.58E-03	2.46E-03	3.36E-03	3.06E-03	2.37E-04
wupwise	1.56E-02	6.96E-03	9.45E-03	3.12E-03	4.95E-04
dbt-2	1.70E-01	6.73E-03	2.37E-02	4.35E-03	1.61E-03
SPECjbb	2.34E-01	1.75E-02	7.61E-02	1.70E-03	3.34E-03

Whenever the operating system is unable to find a schedulable process, it issues the "halt" instruction. This puts the processor in a low power mode in which the clock signal is gated off in many parts of the chip. This mode reduces power consumption in the processor to less than 25% of typical. Because the memory subsystem does not make use of significant power management modes, its variation is due only to varying levels of utilization. These workloads exhibit large variations in memory utilization, therefore this has a significant impact.

In contrast, the chipset and I/O subsystems have little variation in utilization. Because these subsystems also do not make use of power-saving modes, their total variation is very low. In the case of I/O, the observed workloads make little or no use of disk and network resources. For the chipset subsystem, the causes are not as clear and require further study. As mentioned in the previous section the lack of disk power management causes little variation in disk power consumption. If these subsystems are to benefit from dynamic adaptation, workloads with larger variation in utilization would be needed.

In order to justify the use of CoV for identifying workloads with distinct phases we consider probability distributions for some of the extreme cases. In order for a subsystem–workload pair to be a strong candidate for optimization, it must have distinct program/power phases. If a workload exhibits constant power consumption it is difficult to identify distinct phases. Furthermore, if the difference in phases is very small, it may be difficult to distinguish a phase in the presence of sampling noise. Therefore, we propose that a strong candidate should have multiple distinct phases. This can be observed in the power amplitude distributions in Figures 8.10 and 8.11.

In Figure 8.10 we see the gcc workload running on the CPU subsystem. Because this workload has significant variation in instructions per cycle

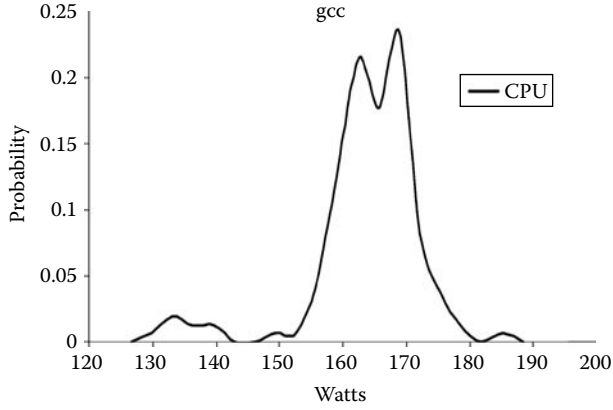


FIGURE 8.10
CPU power distribution, gcc: CoV = 51.6×10^{-3}.

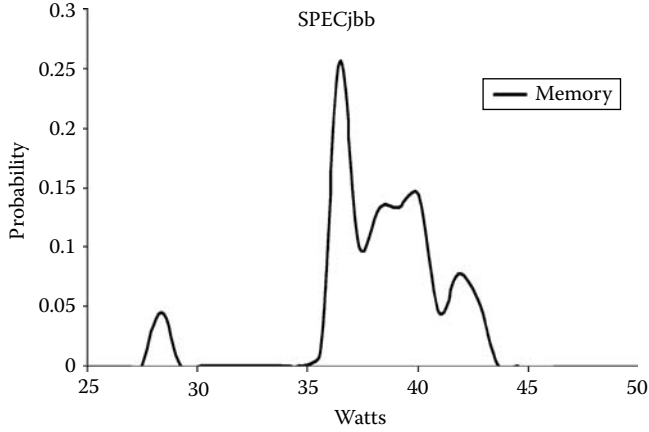


FIGURE 8.11
Memory power distribution, SPECjbb: CoV = 76.1×10^{-3}.

(IPC) and IPC has been shown to be strongly correlated with power [9], the resultant power variation is also significant. From this graph three local maximums are apparent at: 133 W, 162 W, and 169 W. Applying Bircher's models, these correspond to IPCs of ~0, 1.11, and 1.52. Therefore, approximately 5% of the time the processor is stalled waiting for memory or a pipeline fill (IPC = 0). This can be found by taking the sum of the probabilities under the first local maximum near 133 W. The remainder of the workload has varying degrees of utilization, but typically has IPC greater than 1. Therefore, dynamic adaptations for this subsystem–workload pair would likely need to make use of the high-IPC cases which are very common. The low IPC phases are too rare for this combination.

Similarly, the memory subsystem coupled with SPECjbb exhibits a large range of variation. In Figure 8.11, four distinct local maximums are visible. This lowest, which is near 28 W, corresponds to idle power. Therefore, about 4% of this workload makes no access to memory. The other three maximums at 37 W, 39 W, and 42 W are strong candidates for adaptation because they are significantly different from adjacent maximums.

At the other extreme of variation we consider two floating-point workloads: art and mgrid running on disk and chipset subsystems, respectively. Unlike dbt-2, these workloads are memory-bound and make little use of the disk subsystem. The resultant distribution for art can be seen in Figure 8.12. Although two local maximums are apparent, their difference is very small. The entire range of observed power consumptions varies from only 21.865–21.9 W, a difference of only 35 mW. Because no direct access of the disk is made from within the application, the only disk access is caused by period operating system traffic. It is possible that the two maximums are caused by the idle case and a rare seek/read/write cycle. Because the seek/read/write cycles would have to be very rare to produce such a small difference, it is

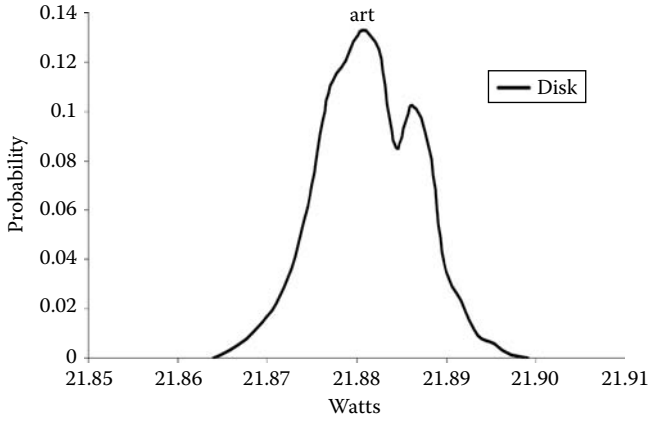


FIGURE 8.12
Disk power distribution, art: CoV = 0.251×10^{-3}.

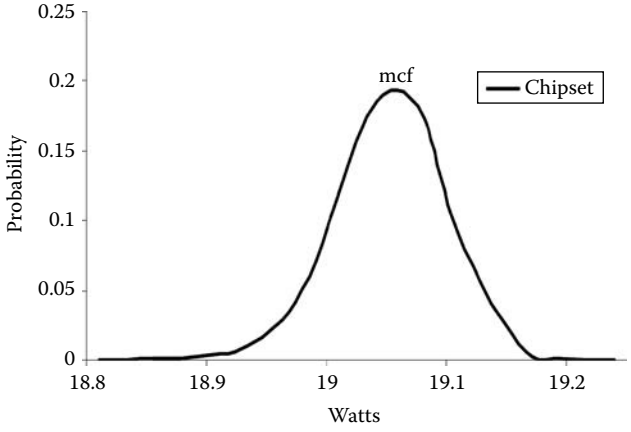


FIGURE 8.13
Chipset power distribution, mcf: CoV = 8.53×10^{-3}.

difficult to distinguish them from noise in the measurement. An even simpler distribution exists for the chipset subsystem running mgrid in Figure 8.13. In this case only one maximum exists at 19.05 W. The total variation is approximately 300 mW. For both cases it is quite difficult to identify multiple distinct phases. Therefore, these subsystem–workloads are not strong candidates for dynamic adaptation.

8.4.3 Duration

The presence of power variation is not sufficient to motivate the application of power adaptation. Due to the overhead of detection and transition,

TABLE 8.5

Percentage of Classifiable Samples Using dbt-2

Duration(ms)	CPU	Chipset	Memory	I/O	Disk
CoV = 0.25					
1	98.5	100	100	99.5	100
10	90.8	100	100	87.6	100
100	70.0	100	100	85.3	100
1000	36.0	100	100	96.3	100
Error %	8.78	3.70	3.47	15.2	6.31
CoV = 0.10					
1	91.7	100	100	81.1	100
10	66.0	100	98.6	35.7	88.6
100	43.1	100	94.4	21.0	95.6
1000	9.30	100	93.1	0.00	95.0
Error %	4.60	3.70	3.47	6.63	6.31
CoV = 0.05					
1	61.6	88.3	97.7	22.4	98.4
10	25.5	78.0	91.2	1.70	32.1
100	6.00	63.2	78.6	0.00	18.5
1000	0.00	64.4	50.0	0.00	0.00
Error %	3.38	3.46	2.68	3.67	2.93

adapting for short duration phases may not be worthwhile. Table 8.5 presents the percentage of samples that are classifiable as phases with durations of 1 ms, 10 ms, 100 ms, and 1000 ms under the dbt-2 workload. These results assume a group of samples can be defined as phases of multiple durations. As described in Section 8.3.3, a 100-ms phase would be made up of ten 10-ms phases. Results for coefficient of variation of 0.25, 0.1, and 0.05 are presented. At CoVs of 0.25 and 0.1 excessive error exists especially in I/O subsystem phase classifications. A probable cause of the error is the greater sample-to-sample variability of the I/O power trace. The disk subsystem, which has higher than average error, also has a wider than average distribution. For the following discussion, a CoV of 0.05 is utilized.

The effect of narrow chipset and memory distributions is evident in their high rates of classification. For both, at least half of all samples can be classified as 1000-ms phases. In contrast, CPU, I/O, and disk have no 1000-ms phases and considerably fewer phases classified at finer granularities. These results can be used to plan power management strategies for a particular workload–subsystem combination. For example, by noting that the I/O subsystem has almost no phases longer than 1 ms, the designer would be required to use very low latency adaptations. In contrast, the disk subsystem has 18.5% of samples definable as 100-ms phases, thus providing greater

TABLE 8.6

Example Workload Phase Classification

	High Power (%)	Med Power (%)	Low Power (%)
High Duration	5	10	20
Med Duration	0	15	5
Low Duration	10	35	0

opportunity to amortize adaptation costs. Although chipset and memory subsystems have a large percentage of classifiable samples, they may not be viable candidates for adaptation. By also considering that most of the chipset and memory samples are very close to the average standard deviations of 0.9 W and 1.4 W, respectively, there may be insufficient variation for runtime phase detection.

From these results, it is clear that distinct phases are detectable at granularities ranging from seconds to milliseconds. The next step in utilizing the phases is to combine the amplitude and duration results to direct power management strategies. An example classification is given in Table 8.6.

This classification can be used to direct selection of power-saving techniques. The phase duration selects the power management type based on similar transition times. The power level and frequency work in opposition to each as a policy control. For example, although a particular phase may occur 5% of the time, because it is such a high-power case it would be valuable to reduce its power. This is similar to the case of the CPU presented in Figure 8.9. At the other extreme, a phase may consume very low power, but because it occurs very frequently it would be valuable to address.

8.5 Conclusion

In this chapter we have presented a framework for measuring power at a fine grain. Using this framework we show that for scientific workloads, the CPU and memory subsystem exhibit the greatest variation in power consumption. The large variation is shown to be due to the presence of power management facilities or varying levels of utilization. Other subsystems such as chipset, I/O, and disk contain much less variation due to a lack of power management facilities and low utilization. We also illustrate distinct power phases in the dbt-2 commercial server workload ranging in duration from milliseconds to seconds and amplitude variations from 30 to 300%. Furthermore, we suggest that for this workbad CPU, I/O, and disk subsystems have a greater potential for phase detection.

References

[1] Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam, Managing server energy and operational costs in hosting centers. *ACM SIGMETRICS Performance Evaluation Review*, pp. 303–314, June 2005.

[2] Canturk Isci and M. Margaret Martonosi, Runtime power monitoring in high-end processors: Methodology and empirical data. *International Symposium on Microarchitecture*, pp. 93–105, December 2003.

[3] Frank Bellosa, The benefits of event-driven energy accounting in power-sensitive systems. *Proceedings of 9th ACM SIGOPS European Workshop*, pp. 37–42, September 2000.

[4] Karthik Natarajan, Heather Hanson, Steve Keckler, Charles Moore, and Doug Burger, Microprocessor pipeline energy analysis. *IEEE International Symposium on Low Power Electronics and Design*, pp. 282–287, August 2003.

[5] Pat Bohrer, Elmootazbellah N. Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony, *The Case For Power Management in Web Servers*. IBM Research, Austin, TX, www.research.ibm.com/arl.

[6] Aqeel Mahesri and Vibhore Vardhan, Power consumption breakdown on a modern laptop, workshop on power aware computing systems. *37th International Symposium on Microarchitecture*, December 2004.

[7] Xizhou Feng, Rong Ge, and Kirk W. Cameron, Power and energy profiling of scientific applications on distributed systems. *International Parallel & Distributed Processing Symposium*, pp. 34–50, April 2005.

[8] Jeffrey Chase, Darrell Anderson, Prachi Thakar, and Amin Vahdat. Managing energy and server resources in hosting centers. *18th ACM Symposium on Operating System Principles*, pp. 103–116, October 2001.

[9] W. Lloyd Bircher, Madhavi Valluri, Jason Law, and Lizy K. John. runtime identification of microprocessor energy saving opportunities. *International Symposium on Low Power Electronics and Design*, pp. 275–280, August 2005.

[10] Texas Instruments. INA168 High-Side Measurement Current Shunt Monitor. ti.com, May 2006.

[11] Open Source Development Lab, Database Test 2. www.osdl.org/lab_activities/kernel_testing/osdl_database_test_suite/osdl_dbt-2/, February 2006.

[12] Karthick Rajamani and Charles Lefurgy. On evaluating request-distribution schemes for saving energy in server clusters. *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 111–122, March 2003.

[13] Jeremy Lau, Stefan Schoenmackers, and Brad Calder. Structures for phase classification. *IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 57–67, March 2004.

[14] Advanced Micro Devices. *BIOS and Kernel Developer's Guide for AMD Athlon® 64 and AMD Opteron® Processors*, November 2005.

[15] Intel software network. Thermal protection and monitoring features: A software perspective. www.intel.com/cd/ids/developer/asmona/eng/newsletter, February 2006.

[16] Canturk Isci and Margaret Martonosi, Phase characterization for power: Evaluating control-flow-based and event-counter-based techniques. *12th International Symposium on High-Performance Computer Architecture*, pp. 122–133, February 2006.

[17] Ashutosh Dhodapkar and James Smith. Comparing program phase detection techniques. *36th International Symposium on Microarchitecture,* pp. 217–228, December 2003.

[18] John Zedlewski, Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Arvind Krishnamurthy, and Randolph Wang. Modeling hard-disk power consumption. *File and Storage Technologies*, pp. 217–230, March 2003.

9

Visualization by Subdivision: Two Applications for Future Graphics Platforms

Chand T. John
Stanford University

CONTENTS

9.1 Introduction

Obtaining compact geometric representations of complex shapes is critical to performing efficient operations on geometric data in computer graphics and computational geometry. Operations commonly performed on geometric data include compression, animation, and rendering. One effective method for compactly representing geometric data is to subdivide the data into parts that can themselves be represented efficiently. It is often desirable to divide a

shape into meaningful parts because this simplifies any subsequent processing of the shape. Methods for subdividing complex shapes into meaningful parts have a strong impact on a variety of fields. For example, potentially cancerous polyps in a human colon can be automatically detected as geometric anomalies if the colon's geometry is subdivided into polyp and nonpolyp regions [11]. The distribution of temperature in the earth's oceans can be compared to a typical distribution to decide automatically whether a new El Niño is developing, if regions of significantly varying temperature are identified separately from each other. If we are animating an airplane flying over part of North America, we can divide up the geometry of the terrain into meaningful regions, compress the geometric data describing each region, and uncompress a region's data only when the airplane flies over that particular region, to greatly improve efficiency of the animation [10]. Radiosity is a popular method for rendering scenes, but unfortunately it tends be slow in its most general forms. However, if a scene is divided into parts, then computing the illumination for each part of a scene becomes an order of magnitude faster, making radiosity a practical method for visualization of that scene [7]. For radiosity, it may be less critical to have a subdivision that is meaningful, but any other geometric operations to be performed on the data, such as compression for storing the data, will depend on having a subdivision that is meaningful. It would be far less efficient to have to generate different subdivisions of a shape for each type of operation that is performed, when one meaningful subdivision can reveal the overall structure of the shape in a way that makes all operations efficient. In general, meaningful subdivisions greatly enhance the efficiency of common operations performed on complex geometric objects.

In this chapter we introduce two visualization applications in which the subdivision of a geometric object into meaningful parts is central to having an effective description of the shape. In our first application, described in Section 9.2, we introduce a mathematical relationship [12] between a class of smooth curves known as *quadratic Bézier curves* (QBCs), and a class of function sets called *iterated function systems* (IFSs), which can be used to generate complex 2D shapes known as *self-affine sets*. This relationship allows us to construct complex 2D shapes that are represented with no loss of information by a small number of QBCs. Thus, this relationship facilitates compression of these complex 2D shapes. The relationship also enables animation of a continuously changing 2D shape simply by continuously deforming the curves that represent it. The results can be extended to 3D shapes. Although we have not developed a good algorithm for representing an arbitrary 2D or 3D shape by a set of curves in this fashion, we are able to create a variety of shapes using these curves. Essentially each curve represents one "part" or "feature" of the overall shape. IFSs alone have been used in the past for generating movies of complex shapes such as clouds [2]. However, it is not easy to manipulate an IFS in a way that creates realistic shapes and deformations. Using QBCs, however, not only gives us a greater level of intuitive control over the complex shape represented by an IFS, but also provides more control

over smooth deformations of the shape than is offered by simply tweaking the IFS's coefficients. We give three examples of the use of QBCs in modeling and deforming 2D cloudlike shapes based on intuitive ideas about how real clouds form in the 3D world.

Visualization, simulation, and animation of clouds is needed in weather prediction and in realistic scene construction for computer games. In meteorology, clouds are modeled as the result of thermodynamic interactions in water droplet populations. In computer games and computer-generated movies, it is common to use some physics of light movement to maintain a certain level of realism in rendering clouds, but approximations are made in order to avoid significant computational expense. Flight simulators are an example of games in which realistic cloud rendering is needed [9] [24]. As mentioned before, IFSs have also been used for rendering movies of clouds [2], although this technique is not often used in modern cloud modeling.

In Section 9.2, we introduce an IFS-based cloud modeling technique, but focus more on the shape representation aspect of visualization rather than the rendering. Barnsley [2] illustrates that there are ways to make realistic renderings of objects as long as we can compactly represent the geometry of a complex set. In this chapter we demonstrate that we can use QBCs that represent IFS-based clouds, and that the QBCs can be manipulated to also mimic the shape changes that real clouds undergo as they develop.

Real clouds can be classified into three main classes: stratiform, cumuliform, and cirriform. Stratiform clouds are formed by gentle mixing of air masses of different temperatures with minimal vertical movement of air: these clouds are commonly associated with steady drizzles and fog. Cumuliform clouds include the puffy "fair weather" clouds typically seen on partly cloudy days, as well as the large cumulus congestus and cumulonimbus clouds that generate thunderstorms. Cirriform clouds are wispy high-level clouds seen at the top of the troposphere, at the highest altitudes where clouds can form. We demonstrate that we can control the formation of 2D self-affine clouds that mimic the geometry and formation of stratus clouds, cumulonimbus clouds, and lenticular altocumulus (lens-shaped middle altitude) clouds.

In our second application, in Section 9.4, we describe four algorithms for segmentation of geometric shapes represented as 3D triangle meshes, and apply these algorithms to human tooth data. Researchers who study statistics and abnormalities of human tooth shapes are interested in producing such segmentations in order to classify and compare teeth in large dental databases. Those who research dental morphology to study genetic factors in bone structures of people in various populations are also interested in automatic shape analysis of teeth. Visualization of developing and changing teeth is itself useful for those who study human teeth and prescribe stage-by-stage dental procedures, and a meaningful decomposition of the shape of a tooth is useful in constructing simulations and measurements of changes in different areas of the tooth. The central problem is to produce a meaningful segmentation of the geometry of a tooth.

Some 3D shape segmentation algorithms are extensions of 2D *image segmentation* algorithms, which have received decades of attention in computer vision. For example, one of the first recent papers on 3D mesh segmentation [18] extends an image segmentation approach based on the concept of "watersheds." Others use approaches based on deformable models for shapes from medical imaging [1], electrical charge distributions [25], implicit surfaces [3], stereoscopic image pairs [14], cutting along edges of high curvature [8], differential geometry [22], and simple surface patch extraction [19]. Some approaches treat a whole surface as a single segment and repeatedly divide it into smaller segments [13], whereas others treat each point or face of a surface as a single segment and merge neighboring segments together to form larger segments [7]. Some techniques are results of applying ideas from human vision theory [21] to computer vision. Some apply classical approaches from data clustering [15] and others use new approaches based on the topological characteristics of a shape [4]. We apply versions of four of the above algorithms to human teeth and assess which of them is the best algorithm both for our application and for general geometric data.

9.2 Controlling Self-Affine Clouds Using QBCs

First we introduce some basic mathematics of QBCs and IFSs. Then we prove the QBCIFS theorem, which relates QBCs and IFSs. Finally we use the theorem to generate animations of 2D self-affine clouds that bear an overall geometric morphology similar to real-world 3D clouds.

9.2.1 Fundamentals of Curves and IFSs

9.2.1.1 Quadratic Bézier Curves

Throughout this chapter, $\mathbb{E}^2$ denotes the set of points in the Euclidean plane, and $\mathbb{R}^2$ denotes the set of vectors in the plane. Let, $P_0 \; P_1, \ldots, P_n \in \mathbb{E}^2$. Let α_0, $\alpha_1, \ldots, \alpha_n \in [0,1]$ such that

$$\alpha_0 + \alpha_1 + \ldots + \alpha_n = 1.$$

Then the *barycenter* of the points $\{P_i\}_{i=0}^n$ with weights $\{\alpha_i\}_{i=0}^n$ is the point

$$P = \sum_{i=0}^{n} \alpha_i P_i. \tag{9.1}$$

Simply put, P is the center of mass ("barycenter") of the points $\{P_i\}_{i=0}^n$ with weights $\{\alpha_i\}_{i=0}^n$. The process of computing a barycenter using Equation (9.1) is called a *barycentric combination*. Note that although addition and scalar

multiplication are not defined over $\mathbb{E}^2$, equation 1 is still valid when it is written as

$$P = P_0 + \sum_{i=1}^{n} \alpha_i (P_i - P_0),$$

because each $P_i - P_0$ is a vector in $\mathbb{R}^2$, where addition and scalar multiplication are valid operations, and the addition of a point to a vector is also a valid operation.

Suppose we are given three distinct, noncollinear points P_0, P_1, and P_2, in the plane. Suppose we are also given a real number $t \in [0,1]$. The *de Casteljau algorithm* proceeds as follows. First compute two barycentric combinations to obtain two intermediate points:

$$P_0^1(t) = (1-t)P_0 + tP_1 \tag{9.2}$$

$$P_1^1(t) = (1-t)P_1 + tP_2 \tag{9.3}$$

Then compute a similar barycentric combination over these intermediate points:

$$P_0^2(t) = (1-t)P_0^1(t) + tP_1^1(t) \tag{9.4}$$

$$= (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 \tag{9.5}$$

The set of points $\{P_0^2(t) : t \in [0,1]\}$ is the quadratic Bézier curve with *control points* P_0, P_1, and P_2. The triangle $\Delta P_0 P_1 P_2$ is called the *control polygon* of the curve. For more on the theory of Bézier curves and surfaces, see [6].

9.2.1.2 Iterated Function Systems

An *affine map* is a transformation w: $\mathbb{E}^2 \to \mathbb{E}^2$ such that

$$w(x,y) = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} e \\ f \end{bmatrix}, \tag{9.6}$$

where a, b, c, d, e, and f are real numbers. We may abbreviate Equation (9.6) with the notation $w(X) = AX + T$, where A is the 2×2 matrix above, $X = [x\ y]^T$, and $T = [e\ f]^T$.

An important fact is that barycentric combinations are invariant with respect to affine maps. That is, if w is an affine map and P is a barycenter defined as in Equation (9.1), then

$$w(P) = w\left(\sum_{i=0}^{n} \alpha_i P_i\right) = \sum_{i=0}^{n} \alpha_i w(P_i), \tag{9.7}$$

that is, $w(P)$ is still the barycenter of the points $\{w(P_i)\}$ with weights $\{\alpha_i\}$. In fact, affine maps are precisely those maps that preserve barycentric combinations. The proof of this fact is straightforward; see page 18 in [6].

An *iterated function system* (IFS) is a set of N affine maps $w_1,\ldots,w_N$. Here we only focus on IFSs with $N = 2$, so we denote an IFS as a pair $\{w_1, w_2\}$ of affine maps. Let $H(\mathbb{E}^2)$ denote the set of nonempty compact subsets of $\mathbb{E}^2$. Associated with each IFS is a function w: $H(\mathbb{E}^2) \to H(\mathbb{E}^2)$ such that $W(K) = w_1(K) \cup w_2(K)$, for every $K \in H(\mathbb{E}^2)$. Let $W^{on}(K)$ denote the repeated application of the map W to the set K a total of n times.

A common restriction is to assume that w_1 and w_2 are *contractive* maps: that, for each $i = 1, 2$,

$$\| w_i(X) - w_i(Y) \| \leq s_i \cdot \| X - Y \|, \forall X, Y \in E^2, \tag{9.8}$$

where $0 \leq s_i < 1$ is the *contractivity factor* of w_i. Assuming that w_1 and w_2 are contractive, then we know from the *contraction mapping theorem* [2] that w_1 and w_2 have unique fixed points X_1 and X_2, respectively, and furthermore, for any $X \in \mathbb{E}^2$,

$$\lim_{n\to\infty} w_1^{on}(X) = X_1 \text{ and } \lim_{n\to\infty} w_2^{on}(X) = X_2. \tag{9.9}$$

Convergence for w_1 and w_2 are with respect to some measure of distance over $\mathbb{E}^2$, such as the Euclidean metric. Not only do w_1 and w_2 push every point toward their own fixed points, but also W maps every $K \in H(\mathbb{E}^2)$ to its own unique fixed point $L = \lim_{n\to\infty} W^{on}(K)$, called the *attractor* of the IFS $\{w_1, w_2\}$. Note that we can start with any nonempty compact set K and end up with the same attractor L, for a fixed IFS. Convergence in $H(\mathbb{E}^2)$ is with respect to the Hausdorff metric. Formally, an IFS consisting only of contractive maps is called a *hyperbolic* IFS. The term "IFS" can be used to refer to an arbitrary collection of maps with no condition imposed on the maps. For our purposes, we always require an IFS to be composed of affine maps, but they need not be contractive unless explicitly stated. We show below that it is not necessary for w_1 and w_2 to be contraction mappings in order for W to converge to the attractor of its IFS, but simply that w_1 and w_2 must mimic the general behavior of an IFS made up of contraction mappings. See [2] for a thorough treatment of IFSs.

9.2.2 An IFS with a QBC Attractor

We now describe a connection between the two seemingly unrelated mathematical objects introduced above: QBCs and IFSs. Consider the QBC defined by $P_0 = (0,0), P_1 = (1/2, 0)$, and $P_2 = (1,1)$ (see Figure 9.1). It is easy to verify that the image of the function $P_0^2(t)$ for $t \in [0,1]$ is the graph of $y = x^2$ for $x \in [0,1]$.

Now suppose we were to use the de Casteljau algorithm to compute $P_0^2(u)$, where $0 < u < 1$ is some fixed real number. We would compute the points

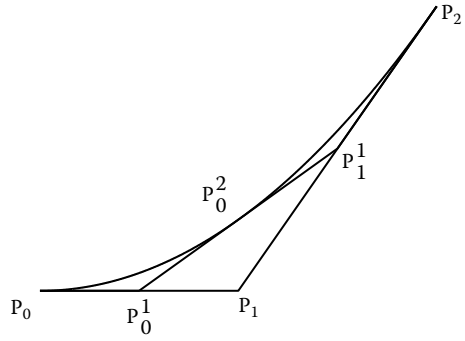


FIGURE 9.1
The de Casteljau algorithm is applied to a quadratic Bézier curve with control points $P_0 = (0,0)$, $P_1 = (1/2,0)$, and $P_2 = (1,1)$. This curve is the graph of $y = x^2$ for $x \in [0,1]$.

$P_0^1(u), P_1^1(u),$ and $P_0^2(u)$. Now define w_1 and w_2 to be the unique affine transformations satisfying

$$w_1(P_0) = P_0 \quad w_1(P_1) = P_0^1(u) \quad w_1(P_2) = P_0^2(u) \tag{9.10}$$

$$w_2(P_0) = P_0^2(u) \quad w_2(P_1) = P_1^1(u) \quad w_2(P_2) = P_2. \tag{9.11}$$

So w_1 maps the original control polygon $\Delta P_0 P_1 P_2$ to the polygon $T_1 = \Delta P_0 P_0^1(u) P_0^2(u)$ and w_2 maps the original control polygon to the polygon $T_2 = \Delta P_0^2(u) P_1^1(u) P_2$. Let S_1 denote the QBC whose control polygon is T_1 and let S_2 be the QBC whose control polygon is T_2. It is easy to verify algebraically that S_1 and S_2 are, respectively, the graphs of $y = x^2$ for $x \in [0,1/2]$ and $x \in [1/2,1]$, respectively. In other words, the maps w_1 and w_2 subdivide the original curve into two subcurves that intersect in exactly one point: $P_0^2(u) = (u,u^2)$. The functions also map the original control polygon to the control polygons that correspond to each of the two subcurves.

We can compute w_1 and w_2 by solving a system of linear equations directly from their definition. This yields

$$w_1 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} u & 0 \\ 0 & u^2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \tag{9.12}$$

and

$$w_2 \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1-u & 0 \\ 2u(1-u) & (1-u)^2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} u \\ u^2 \end{bmatrix}. \tag{9.13}$$

Barnsley [2] describes a way to construct an IFS whose attractor is the graph of a function interpolating a set of points in $\mathbb{E}^2$. Formally, given data points $(x_0, y_0), (x_1, y_1), \ldots, (x_N, y_N)$ where $x_0 < x_1 < \ldots < x_N$, for some $N > 1$, define an IFS $\{w_1, \ldots, w_N\}$ satisfying the following conditions:

$$a_n = \frac{x_n - x_{n-1}}{x_N - x_0}, \tag{9.14}$$

$$e_n = \frac{x_N x_{n-1} - x_0 x_n}{x_N - x_0}, \tag{9.15}$$

$$c_n = \frac{y_n - y_{n-1} - d_n(y_N - y_0)}{x_N - x_0}, \tag{9.16}$$

$$f_n = \frac{x_N y_{n-1} - x_0 y_n - d_n(x_N y_0 - x_0 y_N)}{x_N - x_0}, \tag{9.17}$$

$b_n = 0$, and $0 \le d_n < 1$ for each $n \in \{1, \ldots, N\}$, where the variables are the coefficients of each $w_n : w_n(x, y) = (a_n x + b_n y + e_n, c_n x + d_n y + f_n)$. Then two facts from [2] hold:

1. There is a metric d on $\mathbb{E}^2$ equivalent to the Euclidean metric, such that the IFS is hyperbolic with respect to d. There is a unique nonempty compact set $S \in \mathbb{E}^2$ such that

$$S = \bigcup_{n=1}^{N} w_n(S).$$

2. Moreover, s is the attractor of this IFS, and S is the graph of a continuous function $f : [x_0, x_N] \to \mathbb{R}$ interpolating the original $N + 1$ data points. f is called a *fractal interpolation function*.

We set $N = 2$ and have the data points (0,0), (u, u^2), and (1, 1), and set $d_1 = u^2$ and $d_2 = (1 - u)^2$. Note the resulting IFS is $\{w_1, w_2\}$ where w_1 and w_2 are defined as in Equations (9.12) and (9.13). Then if we define W: $H(\mathbb{E}^2) \to H(\mathbb{E}^2)$ such that $W(B) = w_1(B) \cup w_2(B)$ for all $B \in H(\mathbb{E}^2)$, we have from the above facts and the IFS definitions that $\{W^{on}(B)\}$ converges to the QBC above (call it S) with respect to the metric d, and that S is the unique fixed point of W. In summary, we have shown how to construct a whole family of hyperbolic IFSs (parameterized by $0 < u < 1$) whose attractor is a particular QBC: the graph of $y = x^2$ for $x \in [0, 1]$.

9.2.2.1 IFSs with QBC Attractors

Suppose we are given three points $Q_0, Q_1, Q_2 \in \mathbb{E}^2$ that are distinct and noncollinear. Let T be a QBC with control points Q_0, Q_1, and Q_2. Let $0 < u < 1$ be

an arbitrary real number. If we let $Q_0^2(t)$ denote the point on T with parameter value $t \in [0, 1]$, then define $T_1 = \{Q_0^2(t) : t \in [0, u]\}$ and $T_2 = \{Q_0^2(t) : t \in [u, 1]\}$. Define an affine map ω: $\mathbb{E}^2 \to \mathbb{E}^2$ such that $\omega(P_i) = Q_i$ for $i = 0, 1, 2$, where each P_i is a control point for the graph of $y = x^2$, as defined in the previous section. Clearly this map is unique and invertible. Moreover, because affine maps preserve barycentric combinations, $\omega(S) = T$, $\omega(S_1) = T_1$, and $\omega(S_2) = T_2$,. Let v_1 and v_2 be the unique affine maps mapping T to T_1 and T_2, respectively. It is easy to see that

$$v_1 = \omega \circ \omega_1 \circ \omega^{-1} \text{ and} \tag{9.18}$$

$$v_2 = \omega \circ \omega_2 \circ \omega^{-1}. \tag{9.19}$$

Define V: $H(\mathbb{E}^2) \to H(\mathbb{E}^2)$ such that $V(B) = v_1(B) \cup v_2(B)$ for all $B \in H(\mathbb{E}^2)$. Clearly $V(T) = v_1(T) \cup v_2(T) = T_1 \cup T_2 = T$, so T is a fixed point of V. It is easy to see that $V = \omega \circ W \circ \omega^{-1}$, which implies that $V^{on} = \omega \circ W^{on} \circ \omega^{-1}$. Now for any $B \in H(\mathbb{E}^2)$, we know $\omega^{-1}(B) \in H(\mathbb{E}^2)$, so $W^{on}(\omega^{-1}(B)) \to S$ as $n \to \infty$. But because ω is continuous, $V^{on}(B) = \omega(W^{on}(\omega^{-1}(B))) \to \omega(S) = T$ as $n \to \infty$. Furthermore, if A_1 and A_2 are both fixed points of V, then $V(A_1) = A_1$ and $V(A_2) = A_2$, so $\omega(W(\omega^{-1}(A_1))) = A_1$ and $\omega(W(\omega^{-1}(A_2))) = A_2$, so $W(\omega^{-1}(A_1)) = \omega^{-1}(A_1)$ and $W(\omega^{-1}(A_2)) = \omega^{-1}(A_2)$, but because W has S as its unique fixed point, it follows that $S = \omega^{-1}(A_1) = \omega^{-1}(A_2)$, so $A_1 = A_2 = T$, so V does have a unique fixed point T to which every sequence $\{V^{on}(B)\}$ converges.

Here we have proven that, even if v_1 and v_2 are not contraction mappings in a conventional sense, they still mimic the behavior of w_1 and w_2, and therefore the IFS $\{v_1, v_2\}$ still converges to its attractor, the QBC with control points Q_0, Q_1, and Q_2. Thus we have given a constructive proof of the following result. See Figure 9.2.

THEOREM 1

Any quadratic Bézier curve with distinct noncollinear control points P_0, P_1, and P_2 *is the attractor of some family of iterated function systems* $\{w_1, w_2\}$, *where the family is parameterized by a real number* $0 < u < 1$.

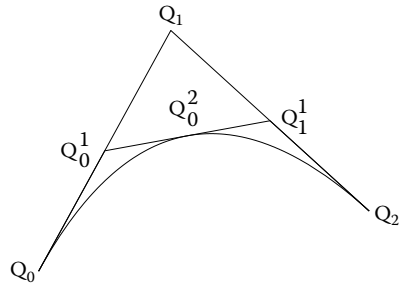


FIGURE 9.2

The de Casteljau algorithm is applied to a quadratic Bézier curve with control points Q_0, Q_1, and Q_2. This curve is the graph of $y = x^2$ for $x \in [0, 1]$. The behavior of the affine maps v_1 and v_2 is analogous to the behavior of w_1 and w_2 on $y = x^2$.

9.2.2.2 Controlling IFS Clouds with QBCs

We have developed a method for finding an IFS whose attractor is a given QBC. Suppose we have several such QBCs $S_1, S_2, \ldots, S_n$. Suppose that we have constructed an IFS $I_i = \{w_{2i-1}, w_{2i}\}$ whose attractor is the curve S_i, for $i = 1,2,\ldots,n$. Then we can combine all the N into one IFS $I = \{w_1, w_2, \ldots, w_{2n}\}$ whose attractor is a dusty cloudlike fractal. We can move control points of the curves S_i in a smooth fashion and recalculate the attractor of the aggregate IFS I to produce an animation that shows the original fractal being deformed smoothly. We would also have chosen arbitrary parameters u_i for each IFS I_i, and these values can also be varied to smoothly deform the attractor of I. If the curves S_i are chosen and deformed appropriately, we can create animations of two-dimensional clouds that form and grow as do real clouds. Although the choice of these curves and their deformations is not brought down to a science by this technique, we have made a considerable improvement over the existing technique of arbitrarily continuously varying the IFS I itself, because our method offers more intuition over the geometric changes in the attractor.

Three examples show how the earlier results can be used to generate animations of growing two-dimensional cloudlike structures. To aid in choosing appropriate Bézier curves, we use some basic (not rigorous) knowledge of the actual physics underlying cloud formation, as well as the shapes of the curves themselves. For more on the basic physics of cloud formation, see [5]. For an explanation of the different types of clouds that form, see [16].

Note that the pictures of the 2D clouds in this chapter are not beautifully shaded. Although it is not hard to modify the picture generation process to produce more nicely shaded images, here we present only completely white points of each cloud over a completely black background, so that the direct result of pooling the IFSs of several QBCs is presented. The pictures are generated using the random iteration algorithm [2]. One way to create shaded images would be to plot pixels with a dim gray intensity, and each time a pixel is hit an additional time by the algorithm, increase the intensity of that pixel. Thus the "denser" areas of a 2D cloud would appear brighter, just as the denser parts of a real cloud would reflect more light and appear brighter than less dense areas.

Example 9.1: Stratus. Stratus clouds are formed and exist in environments where overlying warm air in a stable atmosphere mixes benignly with underlying cool air to form relatively flat clouds at the border of the two air layers. The cloud generally forms from top to bottom. We mimic this cloud formation by starting with one QBC with control points (–10, 0), (0, 0), and (10, 0). This curve is a line segment that lies on the x-axis between $x = -10$ and $x = 10$. Now we make three extra copies of this curve, so that we have four copies of the curve in all. We transform the first copy into a new curve with control points (–10, 0), (0, 1), and (10, 0). All that was done in this transformation is

that the middle control point, (0, 0), was moved to (0, 1). Next, we transform the second copy of the original curve into a curve with control points (–10, –3), (0, –4), and (10, –3) by vertical shifting of the original control points. We transform the third copy of the original curve into a new curve with control points (10, –0), (0, –1.5), and (10, –3). Finally, the fourth copy is transformed to a curve with control points (–10, –3), (0, –1.5), and (10, 0).

As shown in the previous section, the IFSs associated with multiple curves can be pooled into a single aggregate IFS whose attractor is an irregular fractal. However, this fractal can be transformed continuously simply by continuously transforming the control points of the Bézier curves that represent this aggregate IFS. In the previous paragraph, four identical curves make up the initial set of curves, whose IFSs are pooled into one IFS whose attractor ends up being the initial curve. But as these curves are transformed into the new curves described above, the pooled IFS associated with these curves has an attractor that is continuously transformed from a line segment (the initial fractal) into a stratiform cloud (the final fractal). So, the continuous transformation of Bézier curves as described above can be used to graphically illustrate the formation of a stratus cloud. The transformations used on the curves have some relationships to the physical mixing of air layers associated with the formation of such a cloud.

Figure 9.3 illustrates the graphical modeling of stratus cloud formation as described above.

Example 9.2: Cumulonimbus. Updrafts in moist unstable air cause small, puffy cumulus clouds to form above the altitude at which water vapor condenses. If updrafts continue, and sufficient moisture exists in the air, then the small cumulus clouds will become towering cumulus congestus clouds. If updrafts push the top of the cloud up to the tropopause, then vertical growth is halted and an anvil shape appears at the top of the cumulus cloud, forming a cumulonimbus cloud, or thundercloud.

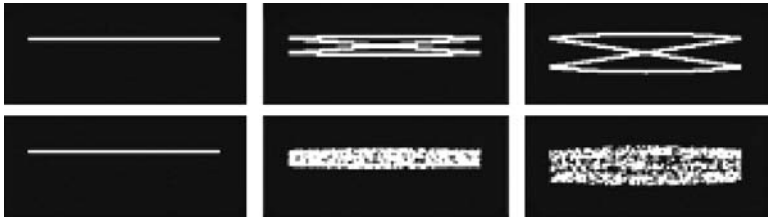


FIGURE 9.3
Graphical illustration of stratus cloud formation by representation of the cloud as the attractor of an IFS that is created by combining the IFSs associated with a set of QBCs. The pictures in the top row are the QBCs that were used to generate the corresponding pictures in the bottom row.

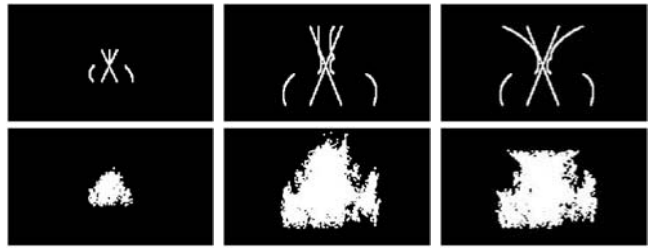


FIGURE 9.4
The growth of a cumulonimbus cloud is illustrated above using QBCs by application of the QBCIFS theorem. The pictures in the top row are the QBCs that were used to generate the corresponding pictures in the bottom row. The first cloud is a 2D representation of a fair-weather cumulus cloud. The second picture represents a towering cumulus congestus cloud. Finally, the third picture represents a cumulonimbus cloud, with an anvil shape on top.

We illustrate such a process by starting with eight identical flat curves (instead of the four curves used in the stratus cloud example) and continuously transforming them upward to form a cumulus cloud, which then grows further into a cumulus congestus cloud. Finally, two curves in the congestus cloud are elongated horizontally to form an anvil-headed cumulonimbus. See Figure 9.4.

Example 9.3: Lenticular cloud. A lenticular cloud is a lens-shaped altocumulus cloud. Lenses are parabolic in shape. So, the relationship between quadratic curves and fractals is quite appropriate for the graphical illustration of a lenticular cloud. The lenticular cloud is modeled by two curves that are nearly identical. See Figure 9.5.

In all space-filling clouds (Examples 9.1 and 9.2), two space-filling curves that formed an X-shape were used. Around these space-filling curves, there were other curves that defined the external shapes of the clouds. Together, the space-filling and external curves generated the two-dimensional fractal clouds that they were meant to represent.

FIGURE 9.5
A lenticular cloud, or lens-shape altocumulus cloud, is modeled above by two nearly identical QBCs. These QBCs are shown in the first picture. The resulting fractal lenticular cloud is shown in the second picture.

9.3 Tooth-Shape Segmentation

We apply four segmentation algorithms to human teeth, represented as 3D triangle meshes. We then discuss which of these algorithms is the best for our application and in general. Finally we speculate on the future of mesh segmentation algorithms, in particular for graphics and medical data processing applications. Throughout this section, we use the following pairs of terms interchangeably: (a) faces and triangles, and (b) segments and clusters.

9.3.1 Bottom-Up Clustering

Garland et al. [7] describe a bottom-up algorithm for segmenting polygonal meshes in order to speed up ray tracing, collision detection, and radiosity computations. Here we describe and apply a simple variation of their algorithm. The input to the algorithm is a *closed triangle mesh*. A closed mesh is one that is topologically equivalent to a sphere; every edge is part of exactly two triangles. The steps of the algorithm are as follows.

1. Number the triangles in the mesh from 1 to N. This ordering usually already exists in the mesh data structure. Let M denote the number of pairs of adjacent (sharing an edge) triangles.
2. Initially, each triangle is in its own segment. We can store this information in an array S of N integers, where the value in the ith slot of the array is the number of the segment containing triangle i. Initially, triangle 1 is in segment 1, triangle 2 is in segment 2, and so on, so $S[i] = i$ for each $i = 1,\ldots,N$.
3. Create a matrix A with M rows and 2 columns. Fill the first column with pairs (i, j) of triangle indices. Fill the second column with "scores" $s(i, j)$ that we assign to each pair (i, j) of adjacent triangles. We can choose any score function s that we wish, as long as the score measures the "flatness" of each pair of triangles. Garland et al. use a quadric metric to compute scores. We use a simpler measure. First, for every triangle in the mesh, we compute its oriented-outward normal vector. Then to any pair (i, j) of adjacent triangles with normal vectors $\mathbf{n}_i$ and $\mathbf{n}_j$, we assign the score $s(i, j) = 1 - \mathbf{n}_i \cdot \mathbf{n}_j$.
4. Sort the rows of the above matrix A in decreasing order of scores.
5. Choose a *threshold score* s_{min} that lies in the range of scores contained in A.
6. Repeat for each pair (i, j) of adjacent triangles, in order of the sorted score list: "merge" the clusters containing the two triangles. That is, look at the segment numbers $S[i]$ and $S[j]$ of i and j. If $S[i] = S[j]$, then i and j are already in the same segment, so we can skip this pair and go on to the next pair in the matrix A. If $S[i] \neq S[j]$, then

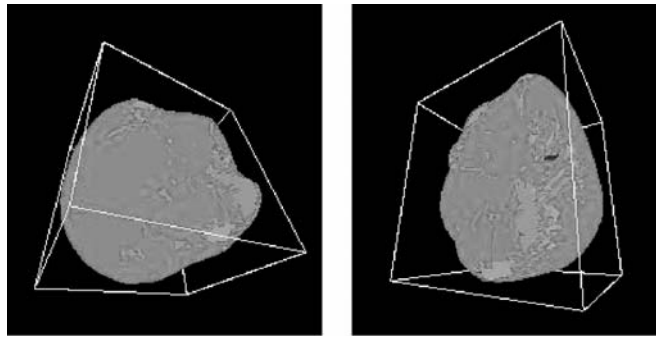


FIGURE 9.6
Decomposition of a tooth using the pairwise merging algorithm based on the bottom-up clustering algorithm of Garland et al. [7]. The two pictures show two views of the same tooth. The blue area is one of the segments constructed by the algorithm. Each shade of pink denotes a different segment. Note how the segments tend to be very small and localized, indicating that the local structure of the mesh is a poor guide to its overall structure.

we want to take all of the triangles in segment $S[j]$ (j's segment) and put them into the segment $S[i]$ (i's segment). This is easy to do: simply scan through S and replace all occurrences of $S[j]$ with $S[i]$.

7. Stop merging when the next score in the list is lower than S_{min}. The result is a segmentation of the original mesh.

Results: See Figure 9.6. The segments tend to be small, and fail to capture the main bumps on a tooth surface. This is because small local fluctuations in the bumpiness of the mesh cause the growth of a segment to suddenly halt before it reaches a reasonable size.

Note: Each pair of adjacent triangles corresponds to a unique edge of the mesh. Thus, if for each triangle from 1 through N, we count its three edges, then at the end we will have counted $3N$ edges total. But because each edge belongs to exactly two triangles, we will have counted every edge twice. Thus the number of edges, or number of pairs of adjacent triangles, is $M = 3N / 2$. Thus the array A we created in the above algorithm is of linear size. Note also that M must be an integer, implying that N must be even: it is impossible to make a closed triangle mesh with an odd number of triangles. The simplest closed triangle mesh is a tetrahedron, which is made up of four triangles.

9.3.2 Top-Down Clustering

Instead of merging small clusters to form larger ones, we can take the opposite route and treat the whole mesh as one huge cluster, and then divide it up into smaller clusters. Katz and Tal [13] use such an approach in order to speed up animations of segmentable objects. We use a variation of their algorithm, which has the following steps. The input is not just a triangle mesh, but also a positive integer k that represents the number of segments into which the mesh will be decomposed.

1. Choose a face with the minimum sum of distances to all other faces in the mesh. Let this be the first *representative face*, f_1.
2. For $i = 2, 3, \ldots, k$ choose the ith representative face f_i to be the face with the maximum possible minimum distance from all previously chosen representative faces $f_1, \ldots, f_{i-1}$.
3. For each nonrepresentative face f in the mesh, compute its distances $d_1, d_2, \ldots, d_k$ to each of the representative faces $f_1, f_2, \ldots, f_k$. Assign f to the segment represented by f_i, a representative face closest to f.

There are many ways to measure the distances in step 3 of the algorithm. We use two methods:

1. *Geometric distance*: The distance between faces f and g is the Euclidean distance between the centroids of f and g in three-dimensional space.
2. *Geodesic distance*: The distance between faces f and g is the distance between the vertices representing f and g when representing the mesh as a graph whose vertices are the centroids of each face and the edges have weights equal to the distances between the centroids of adjacent faces in three-dimensional space. Geodesic distance takes much longer to compute than geometric distance.

Results: See Figure 9.7. The segmentation is much better than in the bottom-up approach: the segments conform more to the general hill and valley shapes on a tooth. However, it is still not clear that this segmentation really captures the essential features of a tooth that characterize what type of tooth it is.

9.3.3 Watershed Segmentation

Watersheds were one of the earliest approaches used in 3D mesh segmentation [18]. The basic idea is to view a 3D shape as a piece of land with hills and valleys, similar to the earth. Page et al. extend these ideas by combining watersheds and fast marching methods [21]. The algorithm we used is based on their work. The steps in our algorithm are as follows.

1. Compute principal curvatures and directions at each vertex of the mesh.
2. Threshold the regions of positive curvature to get an initial marker set of vertices.
3. Apply mathematical morphology to clean up the marker set.
4. Grow each "catchment basin" in the marker set so that every vertex is assigned to some segment. This yields the final segmentation.

Results: See Figure 9.8. Essentially the curvature computation [23] yields nonsensical values. This is because our local information is bad, since our data is a reduced and meshed version of the original data, implying that a

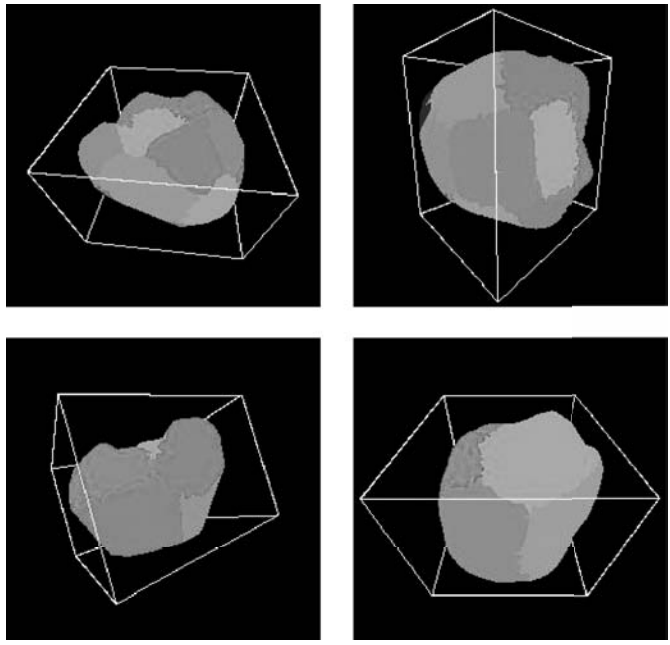


FIGURE 9.7
Top-down decomposition of teeth. The top left image shows the result of dividing a tooth into $k = 20$ segments using geometric distance measurements between faces. The top right image also shows a tooth with $k = 20$ segments but using geodesic distance. The bottom left image shows a tooth divided into $k = 10$ segments using geometric distance. The bottom right image shows a tooth divided into $k = 10$ segments using geodesic distance.

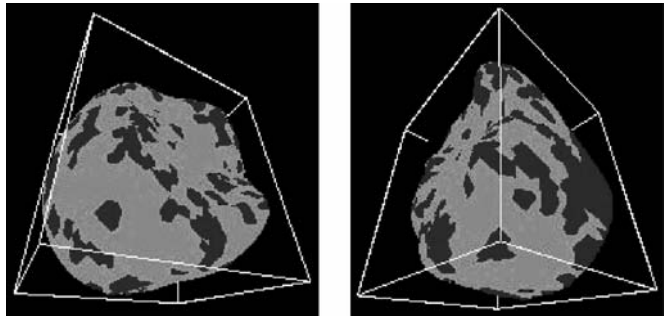


FIGURE 9.8
Initial catchment basins on a tooth using curvature information as in the fast marching watersheds algorithm of Page et al. [21].

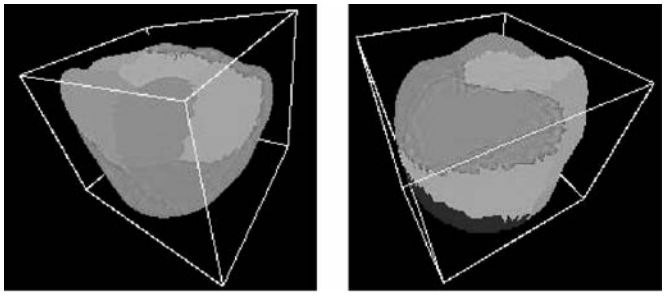


FIGURE 9.9
(Left) Segmentation of a tooth into $k = 10$ segments using Lloyd's algorithm for k-means clustering with $N = 1$ extra iteration. (Right) Segmentation of a tooth into $k = 10$ segments using Lloyd's algorithm with $N = 4$ extra iterations.

lot of important local information was thrown out during processing. The catchment basins in Figure 9.8 yield no useful information about the true hills and valleys on the surface of the tooth.

9.3.4 Lloyd's Algorithm

Lloyd's algorithm is a popular variation of k-means clustering that works as follows. The input is not only a triangle mesh, but also integers $k \geq 2$ and $N \geq 0$.

1. Randomly pick k faces $f_1,\ldots, f_k$ on the mesh.
2. Construct k clusters $C_1,\ldots, C_k$ as follows. Initially, for each $i = 1, 2,\ldots, k$, $C_i = \{f_i\}$. Then for every other face f in the mesh, assign f to the cluster C_j where f_j is the closest of the initial k faces to f.
3. Repeat N times: compute the centroid of each of the k clusters, and then recompute the corresponding clusters, just as before, but using these new cluster centroids.

Results: See Figure 9.9. The segments are similar in nature to those of the earlier top-down clustering. They still do not give a complete description of the features of a tooth, but the segment shapes do conform to some extent to the main cusps on a tooth.

9.4 Conclusion

We have shown that 2D clouds can be depicted using the idea that all quadratic Bézier curves are attractors of IFSs. The formation and change of stratiform and cumuliform clouds can be illustrated by continuously transforming

a set of curves in ways that relate to the graphical and physical nature of the formation processes. Because a fractal cloud generated from Bézier curves shows the shapes of the curves in its own shape, this technique of drawing fractal clouds enables us to illustrate the formation of stratus clouds from stable air mixtures, the growth of cumulonimbus clouds from unstable air to small cumulus to cumulus congestus to anvil-headed thunderclouds, and the shapes of lenticular altocumulus clouds.

The techniques of this chapter can be extended to three dimensions using an analogous relationship between IFSs and Bézier surfaces. The addition of color may require a six-dimensional IFS, where the three spatial coordinates are combined with the three color coordinates (redness, greenness, blueness). Quite possibly, the physics used in choosing curves can be extended to a more rigorous level. The physics that is used in picking Bézier curves or surfaces may become more complicated in higher dimensions, as wind shear and convective cells would induce a nontrivial amount of lateral motion, and the addition of color would present further complications.

Many improvements remain to be made. However, Section 9.2 of this chapter does present a new application of a simple idea that relates two different types of geometrical objects. This method for generating images and animations of real scenes combines the advantages of both Bézier curves and iterated function systems. Extensions of these ideas may prove to be beneficial to some areas of visualization and geometric modeling.

We also applied four segmentation algorithms to human teeth, represented as 3D triangle meshes. In general, methods that relied on local information to form segments (bottom-up clustering and watershed segmentation) performed poorly for our data, whereas the methods that segment based on the global structure of the shape performed much better (top-down clustering and Lloyd's algorithm). The best algorithm in general for noisy data or data whose local information has been tampered with or removed (such as our data), is Lloyd's algorithm. It is a top-down method but also repeatedly refines its own segmentation until it converges close to a good final segmentation; in spirit, this is just like an author starting with a rough draft of a conference paper and repeatedly proofreading it until it has become a polished final draft. Clearly the final result is better than the first attempt. However, for a specific application, people tend to make a special segmentation algorithm that is based on an existing algorithm: for instance, region-growing segmentations in medical imaging originated from the basic idea of bottom-up segmentation. Lloyd's algorithm is more complicated than the first two algorithms we introduced, and so the basic bottom-up and top-down algorithms will continue to coexist with Lloyd's algorithm in the future as a starting point for development of more sophisticated segmentation techniques. Watershed segmentation is another fairly simple and popular segmentation algorithm, which will likely be in use in the future; currently it is still used in medical imaging applications.

For our specific application, new segmentation algorithms that better capture the features of a tooth may need to incorporate information about

how real teeth grow and form their shapes. We are working on constructing a segmentation algorithm based on a growth model of the hormonal mechanism by which cusps on teeth grow in stages. This may highlight a more general need for greater use of scientific principles in the application of computer vision and graphics techniques, rather than purely geometric and statistical approaches.

In general, the concept of representing a complex shape by subdividing it into meaningful parts has proven repeatedly to be a useful method of enhancing visualization, manipulation, and geometric processing of data. The techniques we discussed in this chapter illustrate some of the potential of this concept to improve the state of the art in computer graphics.

Acknowledgments

The work on teeth was supported and supervised by Dr. Leonidas Guibas in the Computer Science Department at Stanford University. The tooth mesh data was provided by Align Technology.

References

[1] Z. Bao, L. Zhukov, I. Guskov, J. Wood, D. Breen. Dynamic Deformable Models for 3D MRI Heart Segmentation. *Proceedings of the International Society for Optical Engineering*, 4684: 398–405, 2002.

[2] M. Barnsley. *Fractals Everywhere*, Academic Press: San Diego, CA, 1988.

[3] R. Benlamri, Y. Al-Marzooqi. 3-D Surface Segmentation of Free-Form Objects Using Implicit Algebraic Surfaces. *Proceedings of the VIIth Digital Image Computing: Techniques and Applications*, C. Sun, H. Talbot, S. Ourselin, T. Adriaansen (Eds.), 2003.

[4] T. Dey, J. Giesen, S. Goswami. Shape Segmentation and Matching with Flow Discretization. *Proceedings of the Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science 2748, F. Dehne, J.-R. Sack, M. Smid (Eds.), 25–36, 2003.

[5] J. A. Dutton. *Dynamics of Atmospheric Motion*, Dover: Mineola, NY, 1995.

[6] G. Farin. *Curves and Surfaces for Computer Aided Geometric Design*, Academic Press: San Diego, CA, 1993.

[7] M. Garland, A. Willmott, P. Heckbert. Hierarchical Face Clustering on Polygonal Surfaces. *ACM Symposium on Interactive 3D Graphics*, 2001.

[8] L. Guillaume, D. Florent, B. Atilla. Constant Curvature Region Decomposition of 3D-Meshes by a Mixed Approach Vertex-Triangle. *Journal of International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 12(2): 245–252, 2004.

[9] M. J. Harris, A. Lastra. Real-Time Cloud Rendering. *Computer Graphics Forum*, Blackwell, Cambridge, MA, vol. 20, 76–84, 2001.

[10] H. Hoppe. Smooth View-Dependent Level-of-Detail Control and Its Application to Terrain Rendering. *IEEE Visualization 1998*, 35–42, 1998.

[11] A. Huang, R. M. Summers, A. K. Hara. Surface Curvature Estimation for Automatic Colonic Polyp Detection. In *Medical Imaging 2005: Physiology, Function, and Structure from Medical Images*. A. A. Amini, A. Manduca (Eds.), *Proceedings of the International Society for Optical Engineering (SPIE)*, 5746: 393–402, 2005.

[12] C. T. John. All Bézier Curves are Attractors of Iterated Function Systems. *New York Journal of Mathematics*, 13(7): 107–115, 2007.

[13] S. Katz, A. Tal. Hierarchical Mesh Decomposition Using Fuzzy Clustering and Cuts. *ACM Transactions on Graphics*, 22(3): 954–961, 2003.

[14] R. Koch. Surface Segmentation and Modeling of 3-D Polygonal Objects from Stereoscopic Image Pairs. *International Conference on Pattern Recognition* (*ICPR*), 233–237, 1996.

[15] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, A. Wu. An Efficient *k*-Means Clustering Algorithm: Analysis and Implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7): 881–892, 2002.

[16] D. M. Ludlum. *The Audubon Society Field Guide to North American Weather*, Alfred A. Knopf: New York, 1991.

[17] B. B. Mandelbrot. *The Fractal Geometry of Nature*, W. H. Freeman: New York, 1983.

[18] A. Mangan, R. Whitaker. Partitioning 3D Surface Meshes Using Watershed Segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 5(4): 308–321, 1999.

[19] A. McIvor, D. Penman, P. Waltenberg. Simple Surface Segmentation. *Digital Image Computing - Techniques and Applications/Image and Vision Computing New Zealand (DICTA/IVCZN)*, 141–146, 1997.

[20] J. R. Munkres. *Topology: A First Course*, Prentice-Hall: New Delhi, India, 1987.

[21] D. Page, A. Koschan, M. Abidi. Perception-Based 3D Triangle Mesh Segmentation Using Fast Marching Watersheds. *Proceedings of the International Conference on Computer Vision and Pattern Recognition*, Vol. II: 27–32, 2003.

[22] T. Srinark, C. Kambhamettu. A Novel Method for 3D Surface Mesh Segmentation. *Proceedings of the 6th IASTED International Conference on Computers, Graphics, and Imaging*, 212–217, 2003.

[23] G. Taubin. Estimating the Tensor of Curvature of a Surface from a Polyhedral Approximation. *Proceedings of the 5th International Conference on Computer Vision*, 902–908, 1995.

[24] N. Wang. Realistic and Fast Cloud Rendering in Computer Games. *Proceedings of the SIGGRAPH 2003 Conference on Sketches and Applications*, Session: Simulating Nature, 2003.

[25] K. Wu, M. Levine. 3D Part Segmentation Using Simulated Electrical Charge Distribution. *Proceedings of the 1996 IEEE International Conference on Pattern Recognition (ICPR)*, 1: 14–18, 1996.

10

A Performance Analysis of Two-Level Heterogeneous Processing Systems on Wavefront Algorithms

Darren J. Kerbyson and Adolfy Hoisie
Los Alamos National Laboratory

CONTENTS

10.1 Introduction

There has recently been renewed interest in the use of acceleration hardware to improve the performance of computationally intensive application components. Field programmable gate arrays (FPGAs), graphic processing units (GPUs), and SIMD arrays are examples of current technologies that can act as accelerators and include the ClearSpeed CSX600 SIMD processor (ClearSpeed, 2005) and the IBM-Toshiba-Sony Cell processor (Kahle et al., 2005). Several large-scale systems using accelerators have been announced

including the roadrunner system at Los Alamos National Laboratory which is expected to have a peak performance of over 1 PF utilizing AMD Opteron processors and IBM Cell processors. However, the utility of these systems will only occur if they can achieve a higher level of application performance than when using a conventional processing system.

A heterogeneous two-level processing configuration is considered here that results from the use of accelerators. The first level consists of conventional cluster processing nodes that are interconnected using a high-speed network. The second level consists of the acceleration hardware which is placed within each of the first-level nodes. Thus the compute nodes of the conventional cluster act as host to the accelerators. There is no connectivity between the second-level acceleration hardware on different nodes, rather the first-level communication network is used for internode data transfer.

In this work we analyze the use of acceleration hardware, which we refer to generically as ADs (or acceleration devices), on a class of applications that use wavefront algorithms. These algorithms are characterized by a dependency in their processing flow that results in a specific order in which individual spatial cells are processed for a given wavefront direction. One application of interest to Los Alamos is Sweep3D. This application performs a deterministic S_n transport calculation that uses a wavefront algorithm as its main computational kernel. It has been estimated that applications like Sweep3D use a high percentage of cycles on the large-scale ASC (Accelerated Strategic Computing) machines (Hoisie, Lubeck, and Wasserman, 2000).

The analysis that we undertake is twofold:

1. Generic ADs on a generic single-direction wavefront calculation
2. A case study with the ClearSpeed CSX600 SIMD AD on a multi-direction wavefront Sweep3D calculation

In the general analysis we characterize an AD as a set of individual processing elements (PEs) arranged in a logical 2-D array that are capable of transferring data with logical neighbors at a certain latency and bandwidth. Additionally we consider each AD PE to achieve a fraction (in the range 1/100 to 1) of the conventional processor performance. The potential performance advantage of an AD is dependent on these parameters.

In the case study using the ClearSpeed CSX600 we consider the use of either 96 or 192 SIMD PEs per PCI-x card. The CSX600 has demonstrated a high level of performance on several applications including DGEMM (ClearSpeed, 2005).

We utilize a performance model in this analysis that has been previously validated on a wide-wide of systems including all ASC systems to date (Hoisie, Lubeck, and Wasserman, 2000). The performance model enables the exploration potential performance, analyzing ADs with widely different characteristics prior to their procurement/deployment within a large-scale system.

Previous work on the analysis of wavefront algorithms includes: the characterization of their computational performance in the absence of

communication costs (Koch, Baker, and Alcouffe 1992); the development of a detailed performance model of Sweep3D for large-scale systems that has been applied to both Massively Parallel Processors (MPPs; Hoisie, Lubeck, and Wasserman, 2000) and to clusters of symmetric multiprocessors (Hoisie, Lubeck, and Wasserman, 1999); and in the use of irregular meshes (Mathis and Kerbyson, 2005). The performance of wavefront algorithms has also been explored on heterogeneous systems in which each node has a different processing capability (Almeida et al., 2005). The contribution of this work is in the analysis of wavefront algorithms on a heterogeneous two-level processing system which can be realistically implemented today in many configurations. It extends work initially presented at the Second Workshop on Unique Chips and Systems (Kerbyson and Hoisie, 2006).

In Section 10.2 we provide an overview of wavefront algorithms while also detailing the earlier performance model. In Section 10.3 we consider the potential performance improvement of using ADs for a range of configurations. In Section 10.4 we detail the case study using the ClearSpeed CSX600. Conclusions drawn from this work are contained in Section 10.5.

10.2 Wavefront Algorithms

Wavefront algorithms are characterized by a dependency in the processing order of cells within a spatial domain. Each cell in a multidimensional spatial grid can only be processed when previous cells in the direction of processing flow have been processed. Examples are shown in Figure 10.1 for one-dimensional, two-dimensional, and three-dimensional regular spatial grids. In each case, five steps of wavefront propagation are shown. For each step, the cell(s) that can be processed are shown in black, and previously

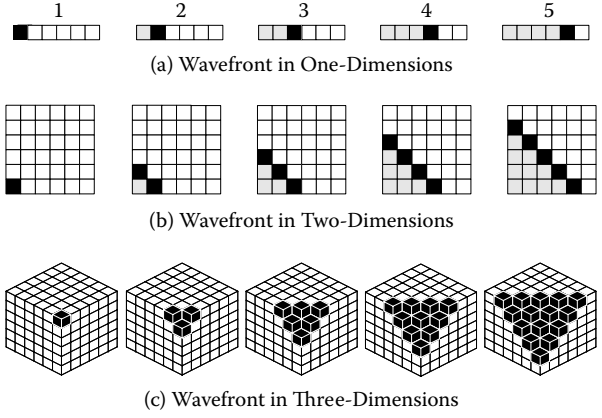


FIGURE 10.1
Example wavefront propagation showing available parallelism.

processed cells are shown shaded (for the 1-D and 2-D cases). The direction of the wavefront is from left to right (1-D), from lower-left to upper-right (2-D), and from the nearest upper corner into the page (3-D). The so-called wavefront thus moves across the spatial grid in the direction of travel, entering at one corner point and exiting after passing through all cells.

The direction of wavefront travel may vary from one calculation phase to another. It has been noted that the available parallelism, that is, the number of spatial cells that can be processed simultaneously, is a function of the dimensionality of the spatial grid minus one. For instance, as shown in Figure 10.1, the available parallelism in the 1-D case is simply one (only one cell can be processed at any time), for the 2-D case a diagonal line of cells can be processed at any time (whose maximum size is equal to the minimum of the two dimensions), and for the 3-D case a diagonal plane of cells can be processed simultaneously (whose maximum size is equal to the minimum of the product of any of the two dimensions).

In the following we consider only a regular 3-D spatial grid with $I \times J \times K$ cells. On a single processor the processing flow can proceed as shown in Figure 10.1c, that is, with a diagonal plane that gradually increases in size from 1 cell (the entry corner) to a maximum and then back to 1. Equally the processing flow may proceed in the *I*, *J*, *K* order using three nested loops. That is, the first complete row of cells can be processed, in the order indicated by the direction of the wavefront, followed by subsequent rows in the same plane and then subsequent planes. This ordering simplifies implementation while also satisfying the wavefront dependency relationships in the direction of travel.

The available parallelism is limited and thus the potential performance gain is also limited. In order to achieve high processor efficiency, a 3-D grid is typically partitioned along two of its dimensions, for example, the *I* and *J* dimensions, leaving the entire *K* dimension local to a processor. In this case, each processor is assigned a subgrid of size $I_c \times J_c \times K$ cells where $I_c = I/P_x$ and $J_c = J/P_y$ on a logical 2-D processing array of $P = P_x \times P_y$ processors. Note that when using a weak-scaling mode the global domain increases in size in proportion with *P* and the subgrid per processor remains a constant.

In a parallel processing flow, the same dependencies in the direction of the wavefront travel have to be observed. In the 3-D grid case, partitioned on a 2-D logical processor array, only one processor is active in the first step (first diagonal), three processors active in the second step (first + second diagonal), six in the third step, and so on. An example 4×4 logical processor array is shown in Figure 10.2. Wavefront n is on the major diagonal of processors, and earlier wavefronts ($n - 1$, etc.) are in front, and later wavefronts ($n + 1$, etc.) are behind. The maximum number of wavefronts that originate from a corner processor is equal to *K*.

The total number of steps in a wavefront operation is equal to the number of wavefronts plus the number of steps required for the wavefront to propagate across the processors (commonly referred to as the pipeline length).

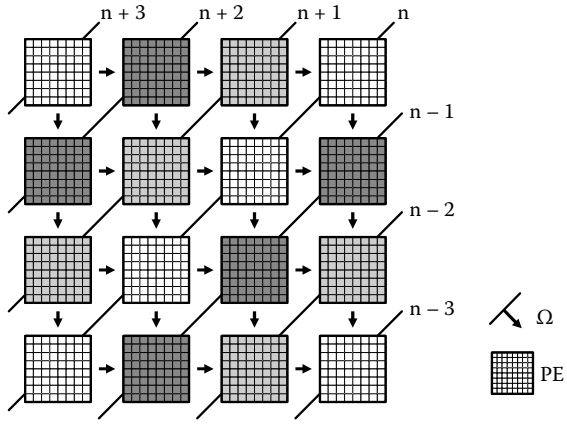


FIGURE 10.2
Example wavefront processing on a 4 × 4 logical 2-D processor array.

By considering the number of wavefronts to be the number of cells in the *K* dimension grouped into blocks of height *B*, the number of steps is simply:

$$steps = \frac{K}{B} + (P_x + P_y - 2) \tag{10.1}$$

where $I_c \times J_c \times B$ cells are processed in each step (in a weak-scaling mode). It has been shown that the parallel computational efficiency (PCE) of wavefront algorithms is given by

$$PCE = \frac{K/B}{K/B + (P_x + P_y - 2)} \tag{10.2}$$

in the absence of communication costs (Koch, Baker, and Alcouffe, 1992). This is the number of wavefronts *K/B* originating from a corner divided by the total number of steps as given by (10.1). The maximum PCE occurs when $B = 1$, and represents an upper bound on the parallel efficiency when communication costs are not negligible.

A more accurate performance model of wavefront algorithms was developed by Hoisie, Lubeck, and Wasserman (2000). This model takes into account additional costs required on a parallel system in terms of communication latencies and bandwidths. It has been applied to the case of MPPs as well as clusters of SMPs (Hoisie, Lubeck, and Wasserman 1999). The model has been validated on many large-scale systems including all ASC machines. It uses as an example the Sweep3D application that is representative of part of the ASC workload. Sweep3D uses a 3-D spatial grid that is partitioned in two dimensions. It is normally executed in weak-scaling mode where the global

problem-size grows in proportion to the processor count and the size of a subgrid remains constant.

A simplified form of the model that gives the processing time for one direction of wavefront travel in Sweep3D is given by Hoisie, Lubeck, and Wasserman (2000):

$$T_{cycle} = \frac{K}{B}.(B.T_c + 4.T_{msg}(B)) + (P_x + P_y - 2).(B.T_c + 2.T_{msg}(B)) \quad (10.3)$$

where the first term represents the number of wavefronts originating from one corner, and the second term is the pipeline length. The computational time for a single wavefront on a single processor is T_c which on a general purpose processor in weak-scaling can be approximated by the single cell processing time multiplied by the number of cells in a wavefront step. The time to communicate one message for a given block size is $T_{msg}(B)$. It can be seen that the total number of steps is the same as in (10.1).

In order to achieve a high parallel efficiency the effect of the pipeline, $P_x + P_y - 2$, and the messaging, T_{msg}, should be minimized. It can be seen from (10.3) that the pipeline is minimized when $B = 1$, and the messaging is minimized when $B = K$. Clearly there is a trade-off between the number of wavefronts and the size of the system. In general, the number of blocks increases (B decreases) with the level of parallelism.

10.3 Large-Scale Two-Level Processing Systems

A two-level processing system is considered here consisting of a conventional arrangement of compute nodes, a high-performance network, and additional acceleration hardware. Compute nodes typically contain between two and eight processors, and the interconnection network is typically a multistage switching fabric such as a fat-tree. Such an arrangement is shown in Figure 10.3.

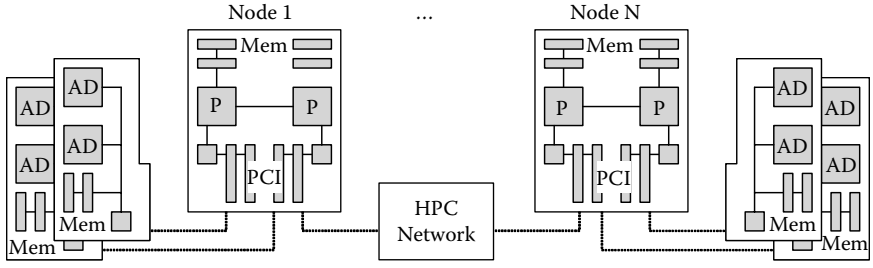


FIGURE 10.3
Example two-level system consisting of: conventional compute nodes, a high-performance network, and additional acceleration hardware.

TABLE 10.1

Compute Node Characteristics

		2.6 GHz Opteron
Processor count	Per node	2
	Large-scale system	*O*(10,000)
Compute time	Per direction/cell (ns)	70
	% of CPU peak	11
MPI communication	Bandwidth (GB/s)	1.6
	Latency (μs)	4

As an example, we consider two-way AMD Opteron processing nodes interconnected using Infiniband. The characteristics of this compute node are listed in Table 10.1. Also listed is the assumed computation performance per wavefront direction per cell based on 40 flops per cell and 11% of a single CPU peak being achieved.

One or more acceleration devices can be added to each compute node. Each AD can be implemented on a plug-in PCI card or connected directly to the processor memory bus. Each can contain one of more PEs. For simplicity we assume that there is one AD per compute node processor.

The exact AD configuration is not a concern to us in the general analysis, but the ClearSpeed CSX600 is used in the case study in Section 10.4. We assume that an AD can be characterized by the following:

1. A number of distinct processors (PEs)
2. Inter-PE communication cost: for PEs arranged in a logical 2-D array
3. Single PE performance: an AD PE is assumed to operate at a multiple of the compute node processor performance (in the range [1/100 … 1])

For the wavefront processing, we assume that the ADs are used to accelerate only the computation associated with an individual block. This is denoted as $B.T_c$ in (10.3). The compute node high-performance network is still used for internode data transfers and is used by appropriate calls to the MPI message library by a compute node processor. It may also be possible to optimize this communication on certain AD implementations.

For simplicity we also assume that if an AD has its own local memory that data transfer to or from this is small in comparison to internode communication.

We first analyze the impact on the PCE in Section 10.3.1 followed by the impact on the cycle time assuming realistic communication costs in Section 10.3.2.

10.3.1 Impact on the PCE

The ADs significantly affect the PCE due to their internal parallelism. When only using the compute nodes, the number of steps in a wavefront operation

is given by (10.1). However, each step now uses the additional parallelism of the AD. The number of substeps required on the AD will be the equal of the block size (B) plus the pipeline length of the AD:

$$steps(AD) = B + (P_x' + P_y' - 2) \tag{10.4}$$

where $P' = P_x' \times P_y'$ is the logical 2-D array of AD PEs. The total number of steps is the product of (10.1) and (10.4):

$$steps(total) = (K/B + (P_x + P_y - 2))(B + (P_x' + P_y' - 2)) \tag{10.5}$$

It can be seen from (10.5) that to reduce the effect of the pipeline across compute nodes B should be small, but to reduce the effect of the pipeline on the AD, B should be large. It can be shown that a minimum number of steps occur when

$$B = \sqrt{\frac{(P_x' + P_y' - 2)}{(P_x + P_y - 2)} K} \tag{10.6}$$

Thus as a system scales in size, the optimum block size decreases proportionally to $\sqrt{(P_x + P_y - 2)}$. If $P_x = P_y = \sqrt{P}$, then it decreases in proportion to $P^{1/4}$.

An example of the utilization of a single AD is shown in Figure 10.4. A system of 256 compute nodes is assumed in which $P_x = P_y = 16$ and, by the use of (10.6), the optimum block sizes are ~8, ~14, and ~22 for ADs of size 4, 16, and 64 PEs, respectively. The processing of several blocks can be seen in each case depicted in Figure 10.4. For example, in Figure 10.4b for an AD containing 16 PEs, the number of steps required to process a block of size 14 K-planes is 20 when using (10.4). As the number of PEs within the AD increases the optimum block size increases, but the average utilization of the PEs, and hence their efficiency, decreases.

The PCE is plotted in Figure 10.5 for various system sizes in terms of compute processor count (which is equal to the AD count), and the number of PEs per AD. Note that the optimum block size as given by (10.6) is used in all cases. It can be seen that the PCE decreases with increasing compute processor count as well with increasing AD PE count. The best PCE occurs when the AD contains only one PE. The worst PCE is only 10% on the largest AD PE count and compute processor count. Note that K is fixed at 1000 in all cases and the PCE increases as K increases.

10.3.2 Impact on the Cycle-Time

The impact on the cycle-time of a wavefront calculation can be assessed using the performance model (10.3). The AD effects only one component in this model: the compute time to process a single block, denoted as $B.T_c$

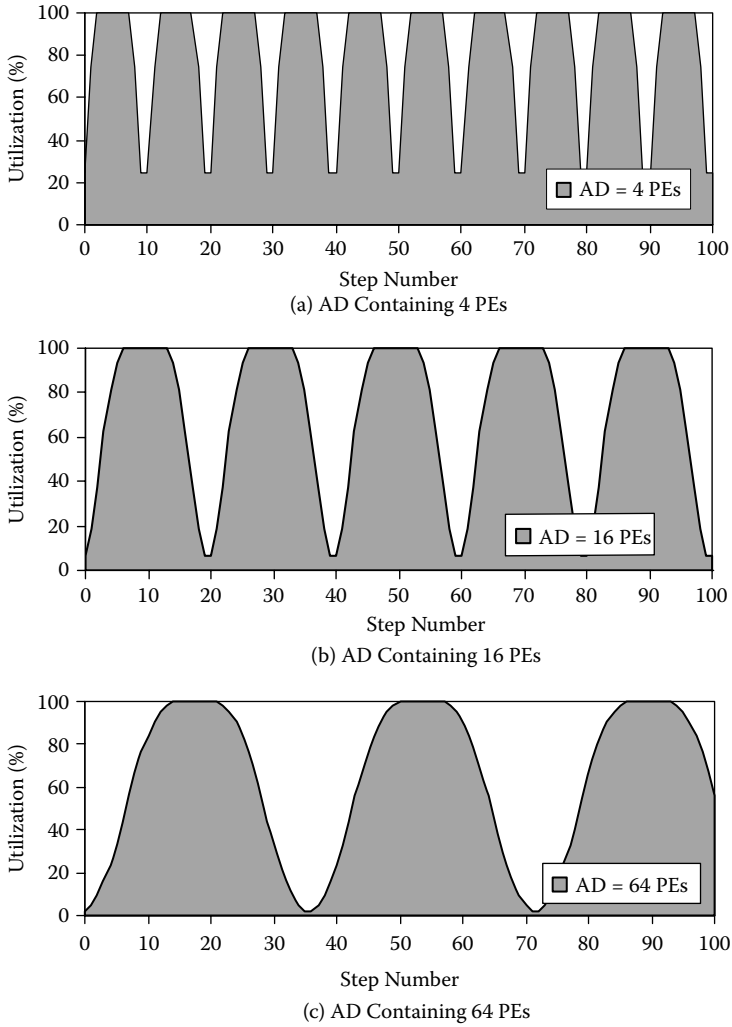


FIGURE 10.4
Utilization of an AD for the first 100 steps of a wavefront calculation.

in (10.3). Using an AD leads to a two-level use of the model. $B.T_c$ for an AD is given by

$$B.T_c = \frac{B}{B'}.(B'T_{AC} + 4.T_{msgAC}(B')) + (P'_x + P'_y - 2).(B'.T_{AC} + 2.T_{msgAC}(B')) \quad (10.7)$$

where B' is the height of a block size on the AD, T_{AC} is the compute time for a single wavefront on an AD PE, and T_{msgAC} is the inter-PE message time on the AD.

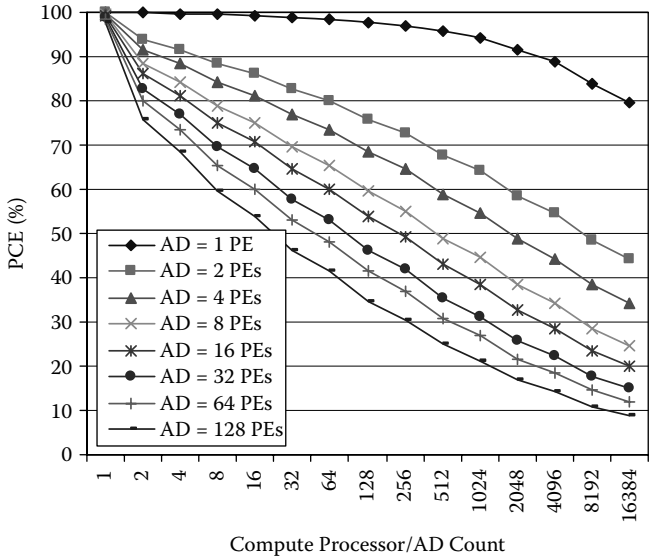


FIGURE 10.5
PCE for various PEs per AD and system sizes.

The overall cycle-time is given by incorporating (10.7) into (10.3). It should be noted that the message time between AD PEs will typically be far less than between compute processors as near-neighbor (usually on-chip) data transfers will be utilized. Heavy-weight MPI message-passing should not be required in such a case.

In order to calculate the cycle-time we consider various AD configurations. The three main parameters of: PE count, PE processing time, and inter-AD PE communication time are varied as listed in Table 10.2. The processing time on an AD PE is considered to be a multiple of the AMD Opteron processor performance corresponding to x1, x0.1, and x0.01.

Figure 10.6 shows the wavefront cycle-time for the case of $K = 1000$, the single-cell AD processing time of 70 ns, and the inter-PE latency and

TABLE 10.2
AD Performance Characteristics

Processor count	Per card	[1 … 128]
Compute time/cell	(ns)	{70, 700, 7000}
Inter-PE communication	Bandwidth (GB/s)	1
	Latency (ns)	50

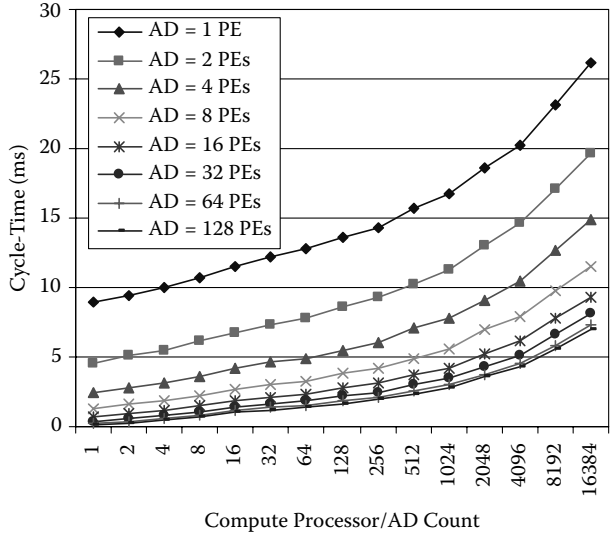


FIGURE 10.6
Cycle-time for various system sizes and PEs per AD.

bandwidth equal to 50 ns and 1 GB/s, respectively. It can be seen that at the largest scale the performance improvement by adding a factor of 128 more PEs to the AD only improves the performance by a factor of 3.5. Indeed there is very little difference in the performance when using an AD with 64 or 128 PEs. This is due to the increased pipeline length and greater inefficiencies that occur due to using smaller block sizes at large-scale.

In Figure 10.7 the parallel efficiency is shown for a range in the single-cell AD processing time. The efficiency for the 70 ns case reflects the cycle-time in Figure 10.6 for the 16 PEs per AD case. It can be seen that as the processing becomes more compute-bound the parallel efficiency approaches that indicated by the PCE. Note that the improved efficiency does not equate to improved performance, quite the contrary as shown in Figure 10.8. Here the cycle-time is shown for the three single-cell compute times. As the compute time per cell improves, so does the cycle-time for the wavefront calculation. It can also be seen that there is not always a performance advantage of using an AD for wavefront calculations. For instance, in the case of an AD PE having 1/10 the performance of an Opteron (700 ns), there is a scale below which the AD improves performance, and above which it is actually slower. This occurs at 512 compute processors as shown in Figure 10.8 for the case considered.

ADs can thus be used to provide a performance improvement for wavefront applications up to a certain system scale. In the next section we consider the effectiveness of the ClearSpeed CSX600 on a specific wavefront application, namely Sweep3D.

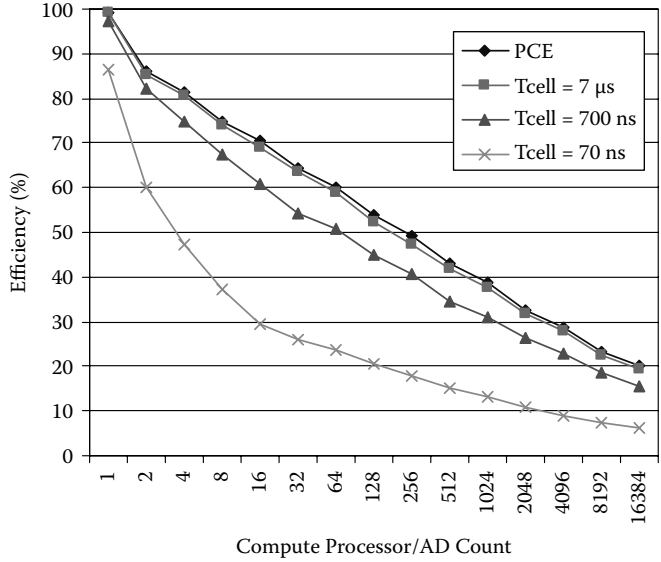


FIGURE 10.7
Parallel efficiency for a range in single-cell processing times (AD = 16 PEs).

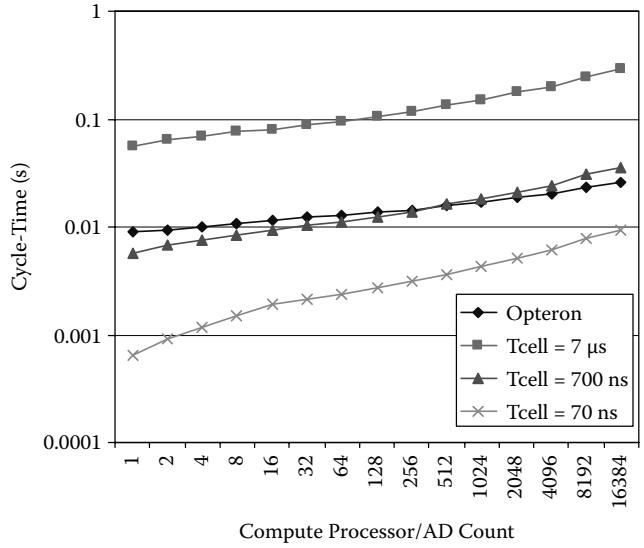


FIGURE 10.8
Performance comparison of a system with and without ADs (AD = 16 PEs).

10.4 Case Study: Sweep3D with ClearSpeed CSX600 Accelerators

The general analysis in Section 10.3 provided an insight into the processing of wavefront calculations using a conventional processing cluster with ADs. In this section we analyze a currently available AD—the ClearSpeed CSX600—using the Sweep3D application in a weak-scaling mode.

The CSX600 is a single-chip containing 96 SIMD PEs as shown in Figure 10.9. Each PE contains an integer ALU, a 16-bit integer multiply-accumulate, an FPU (capable of two double-precision flops per cycle), a general-purpose register file, 6-Kbytes memory, and connections to neighboring PEs as well as to external I/O. Each PE has some local autonomy; in particular each PE has its own address pointer into memory.

The 96 PEs are interconnected linearly in a 1-D ring such that each processor can simultaneously shift data to its left or right neighbors at a peak rate of 4 bytes per cycle thus achieving a high aggregate inter-PE communication bandwidth. However off-chip communication is done via external memory.

Access to external memory is achieved across a memory bus (shared by all PEs) that operates at a peak of 3.2 GB/s. Several chips can be interconnected in a linear array via two ClearConnect Bridge Ports.

The CSX chip is clocked at 250 MHz, resulting in a PE-peak performance of 500 Mflop/s and a chip-peak performance of 48 Gflop/s. Up to two chips can be placed on a single PCIx card, and typically two cards can be placed in a host node. The peak performance characteristics of the CSX600 and those used in this analysis are listed in Table 10.3.

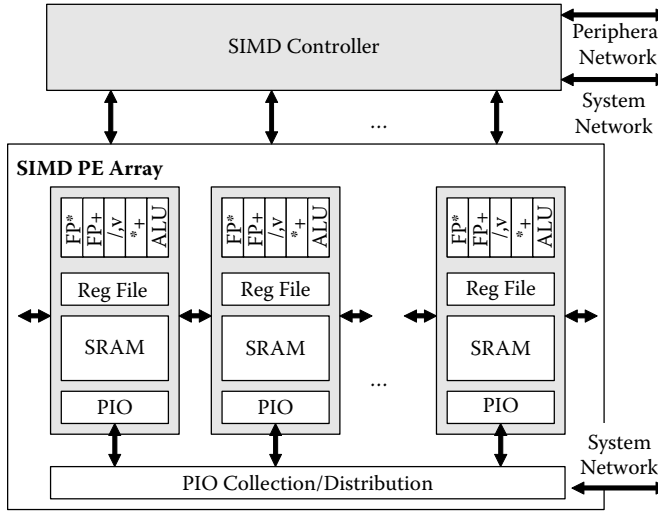


FIGURE 10.9
Schematic of a CSX600 chip.

TABLE 10.3

CSX600 Performance Characteristics

Characteristic		
PEs/chip		96
Chips/card (max)		2
Clock rate (MHz)		250
PE peak (Gflops)		0.5
Card peak (Gflops)		96
Inter-PE communications		
(Intrachip) Peak	Latency (ns)	10
	Bandwidth (GB/s)	1
(Interchip) Peak	Latency (ns)	100
	Bandwidth (GB/s)	3.2
Inter-PE bandwidth		
(Intrachip) logical X	(MB/s)	500
logical Y	(MB/s)	62.5
(Interchip) logical X	(MB/s)	200
logical Y	(MB/s)	62.5
Compute time/cell	(μs)	2.5

It is assumed that the PEs on a chip are arranged as an 8×12 logical 2-D array and as a 16×12 logical array when two chips are on the PCIx card. The effective communication bandwidth between processors varies depending on the communication direction. For a communication in the logical X direction the PEs will be logically neighboring with a peak bandwidth between PEs on-chip of ~1 GB/s (assumed 500 MB/s achievable). However, the peak bandwidth between chips is 3.2 GB/s and is shared by 12 PEs on the edge of the 8×12 logical array. Thus, a realistic figure of 200 MB/s is used for the communications in X (based on an assumed achievable peak of 2.4 GB/s divided by 12). For a communication in the logical Y direction, the achievable bandwidth of 500 MB/s is divided by 8, the distance a Y message shifts to reach its destination PE resulting in 62.5 MB/s. In this case the external bandwidth is sufficient to meet the demand from the 8 PEs on the edge of the 8×12 logical array.

10.4.1 Sweep3D Performance on a Two-Level Processing System Using the CSX600

The performance of Sweep3D is analyzed here on a system whose compute nodes consist of two-way Opteron processors interconnected using Infiniband 4x, and two ClearSpeed CSX600 cards. It uses the performance model as described in Section 10.3 as well as the characteristics of the CSX600 listed in Table 10.3. The system is analyzed as follows.

- An analysis of the compute time on a single CSX600 board compared to an AMD Opteron
- An analysis of the parallel performance of an Opteron cluster with and without the CSX600
- A sensitivity analysis considering a range for the CSX input parameters that were listed in Table 10.3

Note that the current implementation of Sweep3D is considered in this analysis without modification apart from the computation of a block being processed by the CSX600 accelerator. Internode communication is undertaken by the Opteron processors. Optimization of the communication has not yet been considered as a factor in this comparison.

The estimated compute time per cell for Sweep3D, as listed in Table 10.3, results from a detailed analysis of the innermost loop (Reddaway, 2005). This time corresponds to approximately 3.2% of peak when considering that each cell requires ~40 flops including a division (which is costly on the CSX600). It also assumes that access to the off-chip memory required for the next block in the wavefront processing can be overlapped with the computation of the current block; this is architecturally possible but has yet to be demonstrated for Sweep3D. The maximum number of cells that can be contained within a subblock is ~40 due to the small 6 Kbytes memory per PE.

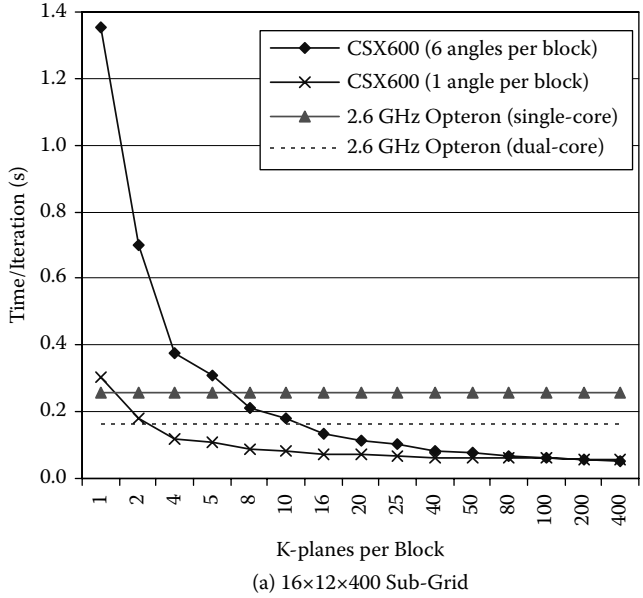
In this analysis several assumptions are made on the problem size being processed. This includes a beneficial view (to the CSX600) of the problem size per PE of $1 \times 1 \times 400$ (or $I_c = 1$ and $J_c = 1$) resulting in a subgrid of size $8 \times 12 \times 400$ and $16 \times 12 \times 400$ per chip and per board, respectively. The K dimension is set at 400 with 6 wavefront angles per cell. The number of angles is an added feature of the wavefront calculation in Sweep3D. We consider two cases that differ in terms of the number of angles per block, either 1 or 6, on the CSX600. It is also assumed that the CSX600 main memory is preloaded with the necessary variables for the wavefront calculation. The best blocking factors are used in all cases in the following analysis.

10.4.2 Sweep3D Single Processor/Single CSX600 Card Compute Time

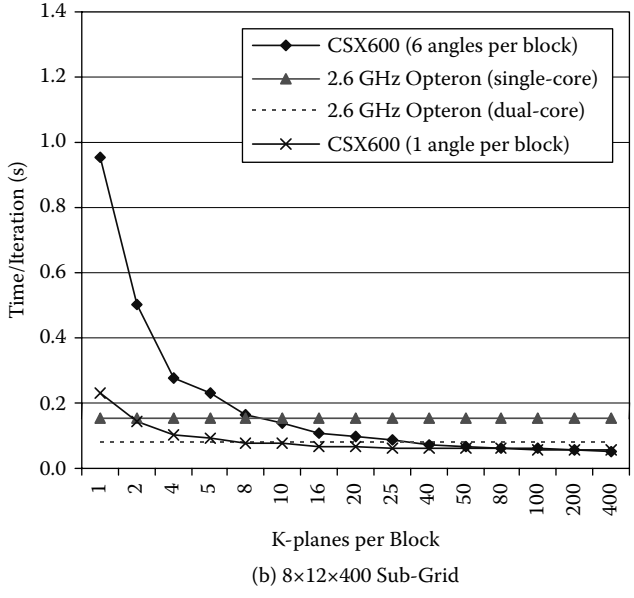
The expected iteration time for Sweep3D when varying the number of K-planes per block between 1 and 400 is shown in Figure 10.10 for a single CSX600 card and a single 2.6 GHz Opteron (single-core and estimated for a dual-core). Figure 10.10a shows the compute time for a $16 \times 12 \times 400$ subgrid (2 CSX600 chips on a card), and Figure 10.10b shows the case for an $8 \times 12 \times 400$ subgrid (1 CSX600).

It can be seen in Figure 10.10 that a CSX600 outperforms the dual-core Opteron processor when a block consists of more than 10 K-planes in the $16 \times 12 \times 400$ subgrid case with 6 angles per block, and more than 2 K-planes with 1 angle per block. Note that the use of only 1 angle per block is an optimistic performance estimate.

However, as we illustrated in Section 10.3, there is a trade-off concerning the block size: the larger the block size is, the poorer the parallel efficiency as



(a) 16×12×400 Sub-Grid



(b) 8×12×400 Sub-Grid

FIGURE 10.10
Sweep3D compute time on a single AMD Opteron and a single CSX600 card when varying the block size.

a system scales in size. This will alter the perceived advantage of the CSX600 shown in Figure 10.10.

10.4.3 Sweep3D Parallel Performance

Figure 10.11 shows the expected iteration time of Sweep3D as the node count increases. Two Opteron clusters are considered consisting of either single-core

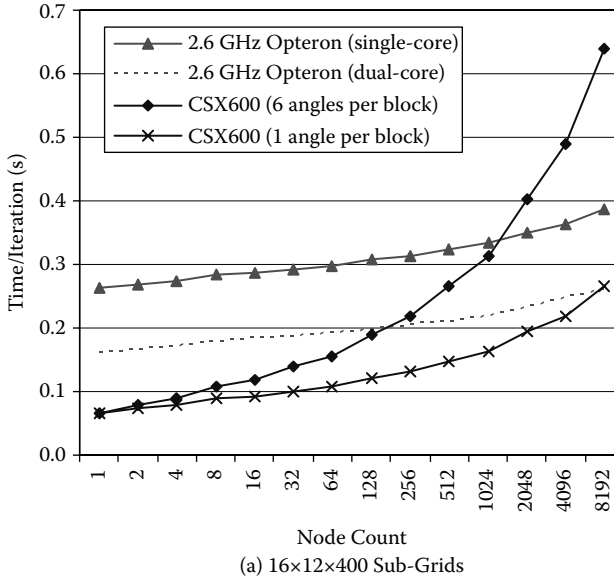
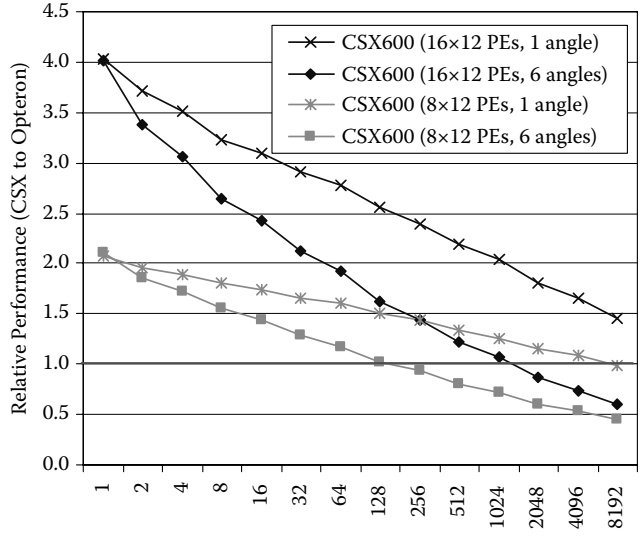
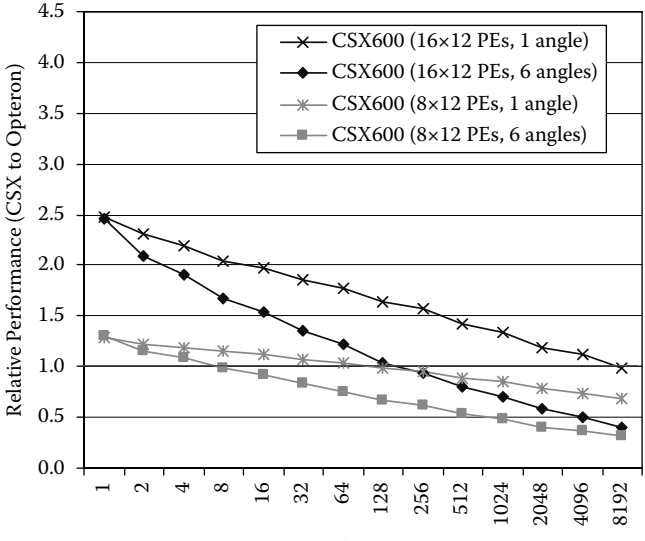


FIGURE 10.11
Expected performance of Sweep3D with and without the CSX600 accelerators.



(a) CSX600 To a Single-Core 2.6 GHz Opteron System



(b) CSX600 To a Dual-Core 2.6 GHz Opteron System

FIGURE 10.12
Relative performance of Sweep3D with and without CSX600 accelerators.

or dual-core processors with a clock speed of 2.6 GHz. The remaining two curves show the performance of Sweep3D when using the CSX600 with either one or six angles per block on either one or two CSX chips per card.

It can be seen that a system with the CSX600 is expected to outperform a 2.6 GHz Opteron Dual-core system up to 8192 nodes in the case of $16 \times 12 \times 400$ subgrids (two chips per card), and up to 128 nodes in the case of $8 \times 12 \times 400$ subgrids (one chip per card).

10.4.4 Improvement in Performance

Figure 10.12 shows the relative performance between the CSX600 system and the Opteron only systems for Sweep3D. Using six angles per block as well as the best case of only one angle per block on the CSX600 is plotted. A value greater than one indicates a performance advantage to the CSX600. The performance advantage of the CSX600 over that of the dual-core Opteron system is at best a factor of 2.5 for the $16 \times 12 \times 400$ subgrid case on a single node. However the advantage gradually decreases with scale to only a factor of 1.5 at 256 nodes, and to a factor of 1.2 at 2048 nodes. When using one CSX600 chip per board there is only a slight advantage up to 64 nodes.

The poorer performance at large-scale results from the increased pipeline length due to the parallelism of the CSX600 and the smaller blocks that are required as the scale increases. This was also seen in the general analysis in Section 10.3 (Figure 10.8).

This result shows that, for the wavefront processing contained within Sweep3D, the CSX600 is capable of providing an improved level of performance on smaller (capacity) sized systems but is not expected to provide a significant performance improvement on larger (capability) sized systems.

10.5 Conclusions

This work has provided an analysis of a two-level processing system on wavefront algorithms. The two-level system is characterized by a set of compute nodes containing conventional processors, interconnected via a high-speed network, and each having additional acceleration hardware. The main characteristic of wavefront algorithms is their dependency in the processing order of spatial cells, which can limit parallel efficiency.

We have shown that in the general case it is possible that acceleration hardware can be used to improve the performance of such applications. However, the level of performance depends upon the level of parallelism of the accelerator, the compute performance of each accelerator PE, and the performance of inter-PE communications.

In general as the parallelism in the accelerator increases, the parallel efficiency will decrease. This limits the potential performance improvement especially when the accelerators are used in large-scale parallel systems.

The ClearSpeed CSX600 chip containing 96 SIMD PEs was used as a case study to analyze the performance of Sweep3D whose wavefront processing is representative of part of the ASC workload. In this it was shown that the high level of parallelism contained within each CSX600 can improve the performance of the Sweep3D calculation when compared with a single- and a dual-core AMD Opteron system even though its clock-rate and single-PE processing time are smaller. However, in the case of large-scale systems, the accelerators are not expected to provide any significant performance improvement that is a direct result of the scaling characteristics of wavefront algorithms.

This makes an interesting observation for the CSX600 in that it may be more suited for a capacity computing situation where an application may make use of up to ~512 nodes, rather than a larger-sized capability computing situation. Accelerators with fewer but faster PEs compared to the CSX600 may be more suited to wavefront processing such as Sweep3D.

It should also be noted that this analysis was favorable to the CSX600 in several ways: the subgrid sizes were chosen to match the number of PEs on a CSX600 chip or card, the smallest possible blocking factors in sweep3D were assumed, and optimistic values for the performance characteristics of the CSX600 were used as input to the performance model. Relaxing some of these assumptions would decrease the magnitude of any advantage of the ClearSpeed CSX600.

Acknowledgments

The authors wish to thank Stewart Reddaway and Peter Rogina of Worldscape Defense for their insights into the performance of the innermost loop of Sweep3D on ClearSpeed. This work was funded in part by the Accelerated Strategic Computing program of the Department of Energy, and by the DARPA High Productivity Computing Systems program. Los Alamos National Laboratory is operated by Los Alamos National Security LLC for the U.S. Department of Energy under contract DE-AC52-06NA25396.

References

Almeida, F., Gonzalez, D., Moreno, L. M., and C. Rodriguez. 2005. Pipelines on heterogeneous systems: Models and tools. *Concurrency and Computation, Practice and Experience* 17(9):1173–95.

ClearSpeed Technology Inc. 2005. CSX Processor Architecture Whitepaper. PN-1105-0003.

Hoisie, A., Lubek, O. M., and Wasserman, H. J. 1999. Scalability analysis of multidimensional wavefront algorithms on large-scale SMP clusters. In *Frontiers of Massively Parallel Computing*, Annapolis.

Hoisie, A., Lubeck, O., and Wasserman, H. 2000. Performance and scalability analysis of teraflop-scale parallel architectures using multidimensional wavefront applications. *Int. J. of High Performance Computing Applications* 14(4):330–46.

Kahle, J. A., Day, M. N., Hofstee, H. P., Johns, C. R., Maeurer, T. R., and Shippy, D. 2005. Introduction to the cell multiprocessor. *IBM J. of Research and Development* 49(4/5):589–604.

Kerbyson, D. J. and Hoisie, A. 2006. Analysis of wavefront algorithms on large-scale two-level heterogeneous processing systems. In *Proc. 2nd Int. Workshop on Unique Chips and Systems, IEEE Int. Symposium on Performance Analysis of Systems and Software*, Austin.

Koch, K. R., Baker, R. S., and Alcouffe, R. E. 1992. Solution of the first-order form of the 3-D discrete ordinates equation on a massively parallel processor, *Trans. of the American Nuclear Soc.*, 65:198–199.

Mathis, M. M. and Kerbyson, D. J. 2005. A general performance modeling of structured and unstructured mesh particle transport computations, *J. of Supercomputing*, 34:181–99.

Reddaway, S. F. 2005. Personal communication, Worldscape Defense Inc.

11

Microarchitectural Characteristics and Implications of Alignment of Multiple Bioinformatics Sequences

Tao Li
University of Florida

CONTENTS

11.1 Introduction

In the last few decades, advances in molecular biology and laboratory equipment have allowed the increasingly rapid obtaining of an enormous amount of genomic and proteomic data [1]. Bioinformatics explores computational methods to allow researchers to sift through this massive biological data in order to provide useful information. Bioinformatics applications are widely

used in many areas of life science, such as drug design, human therapeutics, forensics, and homeland security. A number of recent market research reports estimate the size of the bioinformatics market is projected to grow to $176 billion by 2005, and $243 billion by 2010 [2].

Multiple sequence alignment (MSA), which lines up multiple genomes, is one of the most important applications in bioinformatics [3]. MSA plays a vital role in analyzing genomic data and understanding the biological significance and functionality of genes and proteins. Predicting the structure and functions of proteins, classification of proteins, and phylogenetic analysis are a few examples of the countless applications that use MSA.

Finding the optimal MSA for a given set of genomes is an NP-complete problem [4]. A significant body of work has been done to find heuristic solutions to the MSA problem [5]. However, there is little quantitative understanding of the performance of these MSA methods on modern microprocessor and memory architecture. To ensure good hardware performance, a detailed characterization of how the MSA software uses various microarchitectural features provided by the contemporary microprocessor is needed. Adhering to this philosophy, this chapter studies the performance and characteristics of 12 widely used MSA programs on the Intel Pentium 4 microarchitecture [6]. We examine basic workload characteristics and efficiencies of caching, TLBs, out-of-order execution, branch prediction, and speculative execution.

We chose the Pentium 4 architecture due to its advanced design and popularity. Inasmuch as MSA benchmarks are not well known from the architecture perspective, we believe that an in-depth analysis of a wide variety of MSA software on the representative architecture is crucial in understanding the implications of bioinformatics' multiple sequence alignment tools on today's market. Although the characteristics of MSA applications may vary for different architectures, we believe that our experiments are broad enough from the perspective of bioinformatics market needs. Note that our goal in this chapter is not to develop new MSA software, but to explore how advanced microarchitecture behaves for existing MSA applications.

The rest of the chapter is organized as follows. Section 11.2 provides a background of MSA and describes the selected programs. Section 11.3 describes the experimental methodology. Section 11.4 presents the detailed characterization of MSA applications and their architectural implications. Section 11.5 summarizes the major findings of this work and concludes the chapter.

11.2 Background: Multiple Sequence Alignment (MSA)

Studying evolutionary relationships between sequences is one of the main goals of bioinformatics. The majority of biological sequences are DNA and protein sequences. A DNA sequence is made from an alphabet of four elements, namely A, T, C, and G, called *nucleotides*. A protein can be regarded as

a sequence of *amino acids*. There are 20 distinct amino acids. Thus, a protein can be regarded as a sequence defined on an alphabet of size 20.

MSA is the process of aligning three or more sequences with each other so as to match as many residues (nucleotides or amino acids) as possible. Alignment of multiple sequences involves placing the residues that derive from a common ancestor to the same column. This is achieved by introducing gaps (which represent insertions or deletions) into sequences. Thus, an alignment is a hypothetical model of mutations (substitutions, insertions, and deletions) that occurred during sequence evolution. The best alignment will be the one that represents the most likely evolutionary scenario. Sometimes, the evolutionary history of the sequences cannot be determined precisely. In such cases, usually a computable measure, such as a sum-of-pairs score is used to determine the quality of the multiple alignments. The *sum-of-pairs score* is defined as the sum of the scores of the underlying alignments of all pairs of sequences in the resulting multiple alignment, where a score is computed for a pair of sequences based on the matching and mismatching characters. Figure 11.1 shows a multiple alignment among the DNA sequences A = "AGGTCAGTCTAGGAC", B = "GGACTGAGGTC", and C = "GAGGACTGGCTACGGAC".

The number of multiple sequence alignment methods has been increased steadily. Most MSA algorithms can be classified as one of the following categories: exact, progressive, iterative, anchor-based, and probabilistic methods. Given a set of sequences, exact methods deliver an alignment optimal with respect to a computable objective function, such as sum-of-pairs score, through exhaustive search. Progressive methods find a multiple alignment by iteratively picking two sequences from this set and replacing them with their alignment (i.e., consensus sequence) until all sequences are aligned into a single consensus sequence. Thus, progressive methods guarantee that more than two sequences are never simultaneously aligned. The choice of sequence pairs is the main difference among various progressive methods. Iterative methods start with an initial alignment; they then repeatedly refine this alignment through a series of iterations until no more improvements can be made. Depending on the strategy used to improve the alignment, iterative methods can be deterministic or stochastic. Anchor-based methods use local motifs (short common subsequences) as anchors. Later, the unaligned regions between consecutive anchors are aligned using other techniques. Probabilistic methods precompute the substitution probabilities by analyzing known multiple alignments. They use these probabilities to maximize the substitution probabilities for a given set of sequences.

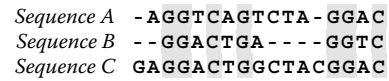


FIGURE 11.1
An example of MSA (the aligned DNA sequences match in seven positions).

Of the many algorithms, we selected a subset of 12 programs based on their popularity, availability, and how representative they were of aligning multiple sequences in general. We briefly describe the MSA tools we selected.

Msa [7] uses high-dimensional dynamic programming (DP) to exhaustively produce all possible alignments of the input sequences. The number of dimensions is equal to the number of sequences compared. It then chooses the alignment with the highest sum-of-pairs score. It uses the distances between pairs of sequences to eliminate unpromising alignments to improve efficiency.

Clustal w [8] first finds a phylogenetic tree for the multiple sequences to be aligned. The phylogenetic tree shows the ancestral relationships among sequences. If two sequences are derived from the same ancestor, they are then located in a subtree rooted at their parent. *Clustal w* progressively aligns pairs of sequences that are siblings on this tree starting from the leaf nodes until all sequences are aligned.

Treealign [9] is similar to *clustal w*. It builds a phylogenetic tree with minimum parsimony on the input sequences. It then aligns pairs of these sequences using dynamic programming starting from the tips of the phylogenetic tree.

T-coffee [10] computes the distance between every pair of sequences. It then computes a phylogenetic tree from these distances using the neighbor joining method. It uses this tree as a guide to align sequences progressively.

Poa [11] progressively aligns pairs of sequences. Unlike *clustal w*, *poa* represents each sequence or the alignment of multiple sequences using graphs. Every node of this graph corresponds to a nucleotide. *Poa* aligns such graphs, instead of sequences, at every step until all sequences are aligned.

Probcons [12] uses a hidden Markov model (HMM) to compute the posterior probability of aligning every pair of letters. It then builds a guide tree (similar to the phylogenetic tree) for the given sequences using these probability values. Finally, it aligns the sequences progressively by following the guide tree.

SAGA [13] employs a genetic algorithm to optimize the sum-of-pairs score of the multiple alignment. It first finds a population of possible solutions. These solutions are then updated iteratively with random mutations to find better alignments.

Muscle [14] computes a k-mer (subsequence of length k) distance for every pair of sequences. Next, it builds a guide tree using these distances. It progressively aligns sequences with the help of this guide tree. Later, it iteratively computes the Kimura distance between aligned nucleotides and realigns the sequences.

Mavid [16] finds common subsequences with the help of a suffix tree. It then chooses such subsequences to align sequences at these positions. *Mavid* uses an anchor-based method for alignments of large numbers of DNA sequences.

Mafft [17] converts nucleotide sequences to sequences of real numbers by storing the volume and polarity of each nucleotide. It assumes that two subsequences are similar if they have similar volume and polarities. *Mafft* uses fast Fourier transformation of these sequences to find the positions where

TABLE 11.1

Five Main Classes of MSA and the Selected Programs for Each Class

Type	Tool
Exact	Msa
Progressive	Clustal w, Treealign, Poa, Probcons, Muscle, T-coffee
Iterative	SAGA, Muscle
Anchor-based	Mafft, Dialign, Mavid
Probabilistic	SAGA, Hmmer, Probcons, Muscle

these sequences have similar volumes and polarities. These positions are used as anchors and the nucleotides at these positions are aligned together.

Dialign [18] aligns pairs of sequences to find long gap-free similar subsequences using dynamic programming. Later, it greedily chooses the subsequence pairs with the largest similarity score and anchors two sequences at that location until all similar subsequences are exhausted. If the position of such a subsequence conflicts with an existing anchor, then that subsequence is discarded.

Hmmer [19] employs hidden Markov models (profile HMMs) for aligning multiple sequences. Profile HMMs are statistical models of multiple sequence alignments. They capture position-specific information about how conserved each column of the alignment is, and which residues are likely.

Table 11.1 summarizes the selected MSA programs and their algorithm categories.

11.3 Methodology

To observe the architectural characteristics of MSA algorithms and how they utilize various microarchitecture features, we conducted our experiments using hardware performance counters. This section describes our experimental setup.

11.3.1 System Configuration

All experiments were run on a 3-GHz Pentium 4 (Prescott) processor [6] with 1 GB of DRAM running RedHat 9.0 Linux kernel version 2.4.26. All MSA benchmarks were compiled using Intel's C/C++ Linux compilers with the

maximum level of optimizations. The input datasets for the MSA benchmarks were chosen from a highly popular biological database, the National Center for Biotechnology Information (NCBI) [20] *Bacteria* genomes databases. In this study, the 317 *Ureaplasma*'s gene sequences [21] were used as the inputs for all the MSA benchmarks. All MSA benchmarks were executed to completion.

11.3.2 Pentium 4 Microarchitecture

The front end of the Prescott microarchitecture fetches and decodes x86 instructions. It builds the decoded instruction into sequences of μops called traces, which are stored in the execution trace cache. The Pentium 4 processors have two areas where branch predictions are performed: in the front end of the pipeline and at the execution trace cache (the trace cache uses branch prediction when it builds a trace). The pipeline in Prescott has 31 stages, so a pipeline flush due to poor branch prediction can result in a much larger clock cycle penalty. The front-end BTB (branch target buffer, 4-K entries) is accessed on a trace cache miss and a smaller trace-cache BTB (2-K entries) is used to detect the next trace line. The trace-cache BTB, together with the front-end BTB, uses a highly advanced branch prediction algorithm. Static branch prediction will occur at decode time if the front-end BTB has no dynamic branch prediction data for a particular branch. Dynamic branch prediction accuracy is also enhanced by adding an indirect branch predictor. The out-of-order execution engine, which consists of the allocation, renaming, and scheduling functions, can issue three μops per cycle to the next pipeline stage. To exploit the instruction-level parallelism (ILP) in the programs, the Prescott microarchitecture provides a very large window of instructions (up to 126) from which the execution units can choose.

The Prescott memory subsystem contains an eight-way, 16-KB L1 data cache and an eight-way, 1-MB, write-back L2 unified cache with 128 bytes/cache line. The levels in the cache hierarchy are not inclusive. All caches use a pseudo-LRU (least recently used) replacement algorithm. The Pentium 4 microarchitecture supports both hardware- and software-controlled prefetching mechanisms.

11.3.3 Pentium 4 Hardware Counters

We used the Pentium 4 hardware counters to measure various architectural events [22]. The Pentium 4 performance counting hardware includes 18 hardware counters that can count 18 different events simultaneously in parallel with pipeline execution. The 18 counter configuration control registers (CCCRs), each associated with a unique counter, configure the counters for specific counting schemes such as event filtering and interrupt generation. The 45 event selection control registers (ESCRs) specify the hardware events to be counted and some additional model-specific registers (MSRs) for special mechanisms such as replay tagging [23]. These counters collect various

statistics including the number and type of retired instructions, mispredicted branches, cache misses, and so on. We used a total of 59 event types for the data presented in this chapter.

11.4 Workload Characteristics

This section provides a detailed workload characterization of MSA benchmarks on the studied microarchitecture. The examined architectural features include instruction distribution, out-of-order execution, cache and TLB performance, branch and efficiency of branch prediction.

11.4.1 Instruction Characteristics

The total number of instructions executed on the studied MSA workloads ranges from hundreds of billions to thousands of billions. This indicates that the computation requirement to align a large set of DNA/protein sequences is nontrivial. The use of performance counters (instead of simulation) allows us to examine the entire program characteristics running on the realistic and meaningful datasets.

Figure 11.2 presents the dynamic instruction profile of the MSA programs. The dynamic instructions are broken down into five categories: load, store, branch, floating point (FP), and integer. As can be seen, the most frequently executed instructions are loads. This is because all these tools need to read data from the dynamic programming matrix and write the results back onto the same matrix many times. The percentage of loads is significantly more than that of store in all the programs because the dynamic programming algorithm has to read multiple entries from the DP matrix to update a single

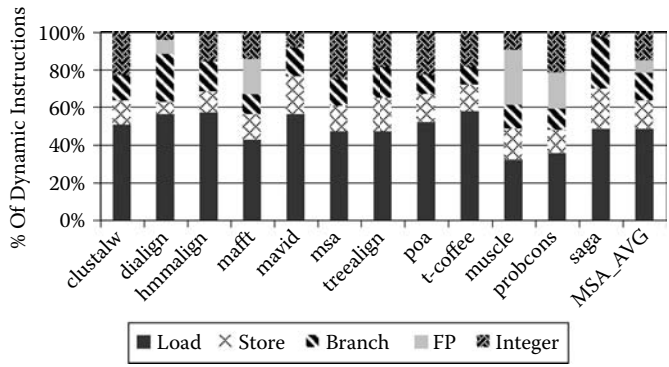


FIGURE 11.2
Dynamic operations profile.

entry. As a whole, memory operations occupy a significant share of the total instruction mix, which is 63% on average. Therefore, MSA workloads are data-centric in nature. This indicates that MSA applications can benefit from techniques to improve memory bandwidth in general.

Branch instructions exhibit significant differences from algorithm to algorithm. For example, 27% of dynamic instructions in benchmarks *dialign* and *SAGA* are branches. This can be explained as follows. *Dialign* usually generates a large candidate set of anchors, which then needs to be analyzed to find a set of nonconflicting anchors. This analysis involves a large number of comparisons among candidate anchors. *SAGA* evaluates and compares all the members of the solution population per iteration. As the population of solutions and the number of iterations increases, the number of comparisons also increases. A more detailed analysis on the branches and branch prediction can be found in Section 11.4.6. The majority of MSA workloads contain few floating-point operations. Only methods that calculate statistics and likelihood values or phylogenetic trees in their algorithms use floating-point instructions. For example, *Mafft* computes the Fourier transformations of the volumes and polarities of the amino acids for different combinations of sequences. *Muscle* incurs floating-point operations during the computation of Kimura distances. *Probcons* computes the posterior probability of aligning every pair of letters.

11.4.2 IPC and µPC

Using the events that count the number of cycles and number of instructions retired during the program execution, we computed the IPC (instruction-per-cycle) of the studied MSA benchmarks. On the high-performance processors such as Pentium 4, the IPC metric indicates how efficiently the microprocessors exploit instruction-level parallelism (ILP). In order to improve the efficiency of superscalar execution and the parallelism of programs, each x86 instruction is further translated into one or more µops inside the Pentium 4 processor. Typically, a simple instruction is translated into around one to three µops. The results of the measured IPC and µops per cycle (µPC) on the benchmarks are shown in Figure 11.3.

The greatest IPC values come from *clustal w*, *dialign*, and *poa*. The lowest IPC values are *msa* and *muscle*. The IPC ranges from 0.15 to 0.93, with an average around 0.60. A lower IPC can be caused by an increase in cache misses, branch mispredictions, or pipeline stalls in the CPU. For example, MSA methods (*mafft* and *muscle*) extensively using floating-point instructions yield lower IPCs due to the pipeline stalls on the long latency floating-point operations. The IPC is remarkably low on benchmark *msa* due to the excessive data cache misses. These cache misses are incurred because the exhaustive search strategy of *msa* reads and writes large amounts of data. The µPC ranges from 0.23 to 1.32, with an average around 0.94. Only six benchmarks (*clustal w*, *dialign*, *treealign*, *poa*, *t-coffee*, and *SAGA*) achieve more than one µops per cycle. This implies that, for the majority of MSA applications, the available ILP that can be exploited by the Pentium 4 microarchitecture is limited.

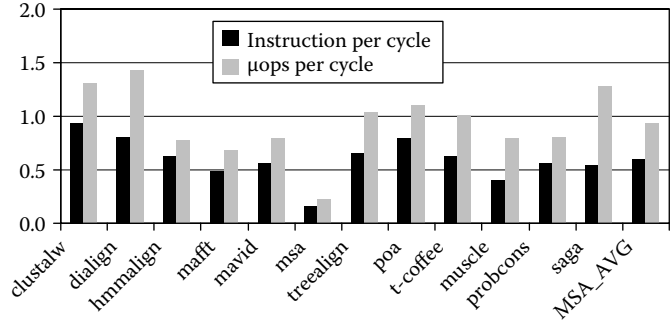


FIGURE 11.3
IPC versus µPC.

11.4.3 Trace Cache

As the front end, the Prescott trace cache sends up to three µops per cycle directly to the out-of-order execution engine, without the need for them to pass through the decoding logic. Only when there is a trace cache miss does the front end fetch x86 instructions from the L2 cache. There are some exceedingly long x86 instructions (e.g., the string manipulation instructions) that decode into hundreds of µops. For these long instructions, the Prescott fetches µops from a special µops ROM that stores the canned µops sequence.

Figure 11.4 shows the proportion of the µops fetched from the L2 cache, the trace cache, and the µops ROM, respectively. As can be seen, a dominant fraction (93%) of the µops is supplied by the trace cache. On benchmarks *dialign*, *mavid*, *treealign*, and *t-coffee*, around 2–8% of the µops come from the µops ROM, implying that these workloads use x86 complex instructions more frequently. The µops ROM contributes 20% and 39% of the dynamically executed µops on benchmark *muscle* and *SAGA*. This is because these two programs excessively use the string manipulation instructions to handle biological sequences. For example, *SAGA* repeatedly mutates existing

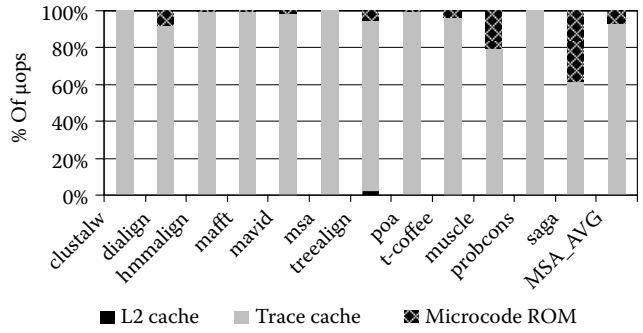


FIGURE 11.4
Source of the µops.

population of alignments which involves costly string operations. Moreover, *SAGA* uses x86 FSQRT (floating-point square root) instructions. The L2 cache contributes less than 1% of the μops on most of the benchmarks (except *treealign*). The instruction footprint generated by benchmark *treealign* yields more trace cache misses. A closer investigation shows that this benchmark performs operations on both graphs and phylogenetic trees alternately. The codes performing these two operations conflict with each other in the trace cache. Nevertheless, on the majority of benchmarks, the Prescott trace cache is highly efficient in providing the μops to the rest of the pipeline. This indicates that the instruction footprints of MSA applications are small and cache misses due to instruction fetches are negligible.

The trace cache operates in two modes: deliver mode and build mode. The deliver mode is the mode in which the trace cache feeds stored traces to the execution logic to be executed. This is the mode in which the trace cache normally runs. When there is a trace cache miss, the trace cache goes into build mode. In this mode, the front end fetches x86 instructions from the L2 cache, translates into μops, builds a trace segment with it, and loads that segment into the trace cache to be executed. Figure 11.5 shows the percentage of nonsleep cycles that the trace cache is delivering μops versus decoding and building traces. Overall, the utilization of the trace cache is extremely high except on the benchmarks *treealign*.

11.4.4 Cache Misses

Figure 11.6 presents the counts of cache misses per 1000 instructions retired. We see that instruction-related cache misses are nearly fully satisfied by the trace cache. Data cache miss ratios are higher because the data footprint is much larger than the instruction footprint. For example, *msa* can cause more than 40 L1 data cache misses on every 1000 instructions executed. This can be explained as follows. Unlike other methods, *msa* fills a multidimensional dynamic programming matrix. As the number of dimensions (i.e.,

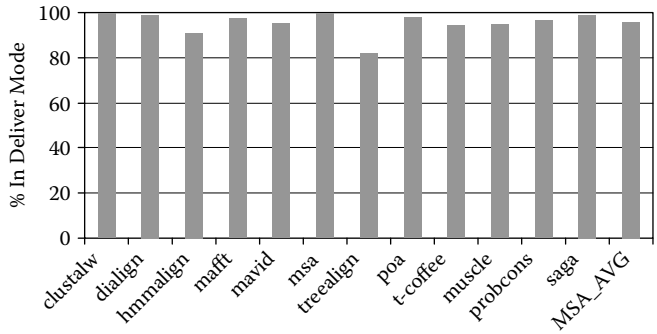


FIGURE 11.5
Percentage of TC deliver mode.

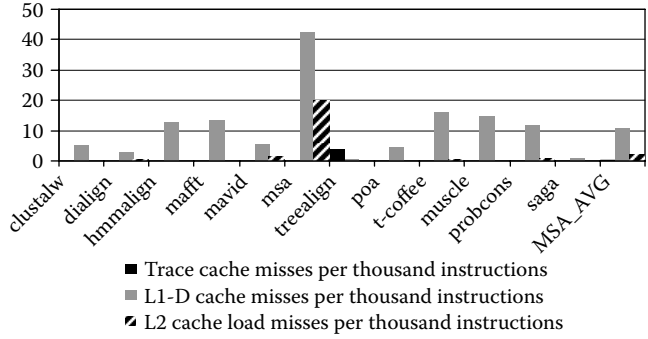


FIGURE 11.6
Cache miss rates.

the number of sequences compared) increases, the number of matrix entries needed to compute a single DP entry increases exponentially. Thus, these entries do not fit into the cache resulting in cache misses every time a new value is computed. On average, the studied bioinformatics applications generate 11 L1 cache misses per 1000 retired instructions.

We found that the L1 data cache misses on most of the benchmarks can be nearly fully satisfied by the L2 cache. The Pentium 4 processors use automatic hardware prefetch to bring cache lines into the unified L2 cache based on prior reference patterns. Prefetching is beneficial because many accesses to the biological sequences are sequential, and thus, predictable.

Interestingly, the benchmarks (*msa*) with the highest L1 data cache misses also have the highest L2 misses, implying their poor data locality. This is because, in order to compute the alignment score for an entry of the DP matrix, *msa* needs to access the information in all neighboring entries of that entry. As the dimensionality of the DP matrix (i.e., number of sequences) increases, the locations of these entries get exponentially far away from each other causing poor data locality. Figures 11.3 and 11.6 show a fairly strong correlation between the L2 misses and IPC, which indicates that the L2 miss latency is more difficult to be completely overlapped by out-of-order execution. We observed that overall prefetching and L2 cache can efficiently handle the working sets of MSA applications.

11.4.5 TLB Misses

The Pentium 4 processor uses separate TLB (translation lookaside buffer) to translate the virtual address into the physical address for instruction and data accesses. Prescott has a 128-entry, fully associative instruction TLB (ITLB) and a 64-entry, fully associative data TLB (DTLB). Figure 11.7 presents the ITLB and DTLB miss rates across the studied benchmarks. The ITLB miss rates are well below 1% on most benchmarks. Figure 11.7 also shows that most of the DTLB accesses can be handled very well by the

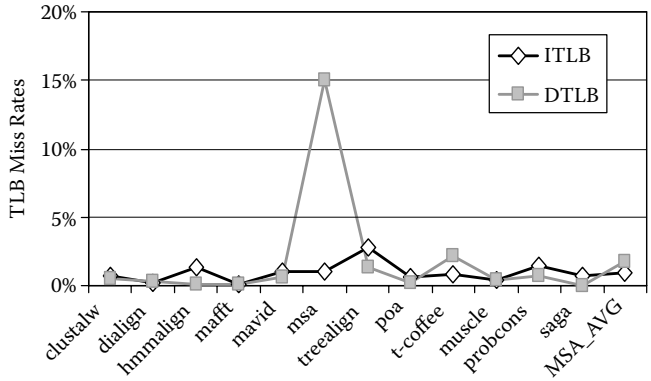


FIGURE 11.7
TLB miss rates.

Pentium 4 processor. Nevertheless, *msa* yields high (16%) DTLB miss rates due to its large data memory footprint. This is mainly caused by the high-dimensional DP matrix that *msa* uses. Because all MSA software run the same input dataset, it is clear that the internal data structures created by the algorithms largely affect the DTLB behavior.

11.4.6 Branches and Branch Prediction

Figure 11.8 presents the fraction of branches that belong to conditional branches, indirect branches, calls, and returns. Conditional branches, ranging from 54% (*muscle*) to 99% (*hmmalign*) of the dynamic branches, dominate the control flow transfers in the MSA applications. Indirect branches account for more than 10% of the dynamic branches on benchmarks *treealign*, *muscle*,

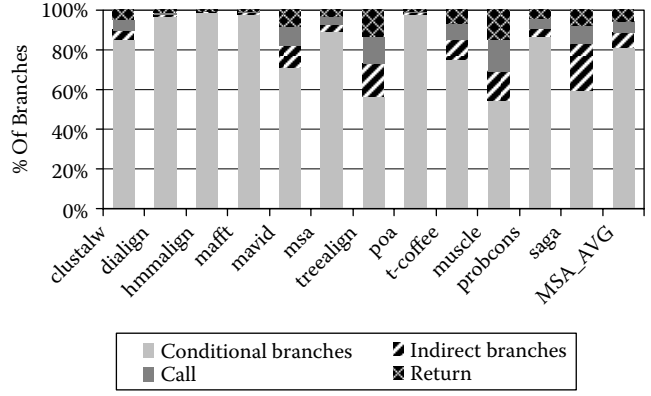


FIGURE 11.8
Dynamic branch mix.

and *SAGA*. We further examined the source code and found that the percentage of indirect branches is caused by the software programming style and they are not an inherent part of the algorithm. For example, the benchmarks *treealign*, *t-coffee*, and *SAGA* embed the case-switch statements in various loops to determine sequence format, or to select one operation from all possible choices to process the sequence elements. The benchmark *muscle*, programmed with C++, uses additional virtual functions to implement the algorithm. On the average, conditional branches, indirect branches, call, and return contribute to 81%, 8%, 6%, and 6% of the total dynamic branches, respectively.

Figure 11.9 shows the branch misprediction rates on the MSA applications. The overall branch misprediction rates exceed 5% on 6 out of the 12 benchmarks. The misprediction rates on the indirect branches are less than 2% on all studied benchmarks. Typically, the targets of indirect branches are difficult to be predicted accurately using a conventional branch target buffer. The results show that the advanced indirect branch prediction mechanism used in the Pentium 4 processor works well on the MSA software. Figure 11.9 also shows that calls and returns can also be predicted accurately with the 16-entry return address stack. To further improve branch prediction accuracy of MSA software, efforts should focus on the conditional branch prediction.

11.4.7 Speculative Execution

To reach high performance, the Pentium 4 machine fetches and executes instructions along the predicted path until the branch is resolved. In case there is a branch misprediction, the speculatively executed instructions along the mispredicted path are flushed. The speculative execution factor or the ratio of the total number of instructions decoded to the total number of

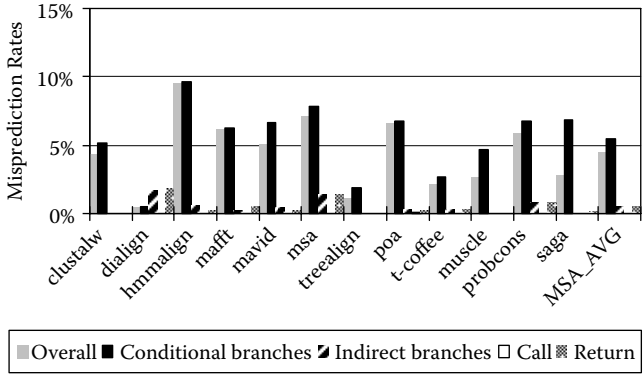


FIGURE 11.9
Misprediction rates.

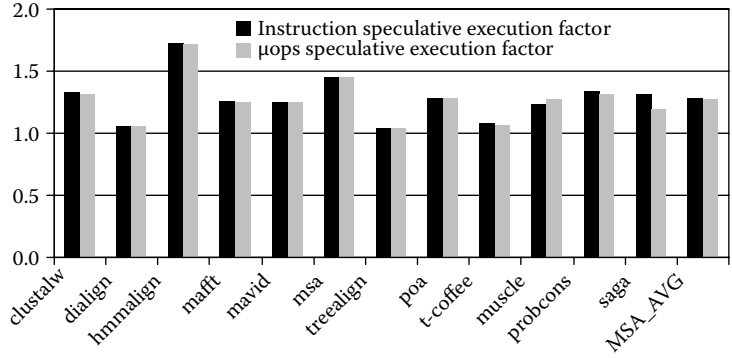


FIGURE 11.10
Speculation factor.

instructions retired quantitatively captures how aggressively the processor executes the speculated instructions.

Figure 11.10 shows the speculative execution factors for instructions and µops on the MSA software. On the average, the processor decodes 27% more instructions than it retires. Note that there is a fairly strong correlation between the branch prediction accuracy and the speculative execution factor on these programs. Due to the use of deeply pipelined design (31 stages behind the trace cache) to reach high operation clock frequency, the accuracy of branch prediction plays an important role on Prescott pipeline performance. MSA benchmarks with higher mispredicted branches per instruction have higher speculated instructions, indicating these applications can further benefit from more accurate branch prediction.

11.4.8 Phase Behavior

Recent computer architecture research has shown that program execution exhibits phase behavior, and these behaviors can be seen even on the largest of scales [27]. Program phases can be exploited to design adaptive microarchitecture, guide feedback compiler optimization, and reduce simulation time. To reveal the phase behavior of MSA applications, we sampled performance counters at a time interval of 0.1 second. Figure 11.11 shows the sampled IPC of six MSA applications. As can be seen, the studied MSA applications show heterogeneous phase behavior. For example, benchmark *t-coffee* shows periodic spikes where program execution yields high IPC. The phase behavior of benchmarks *msa* and *treealign* is highly predictable for the entire program execution. Benchmarks *muscle*, *clustalw,* and *t-coffee* exhibit irregular and unpredictable phase behavior during the program execution.

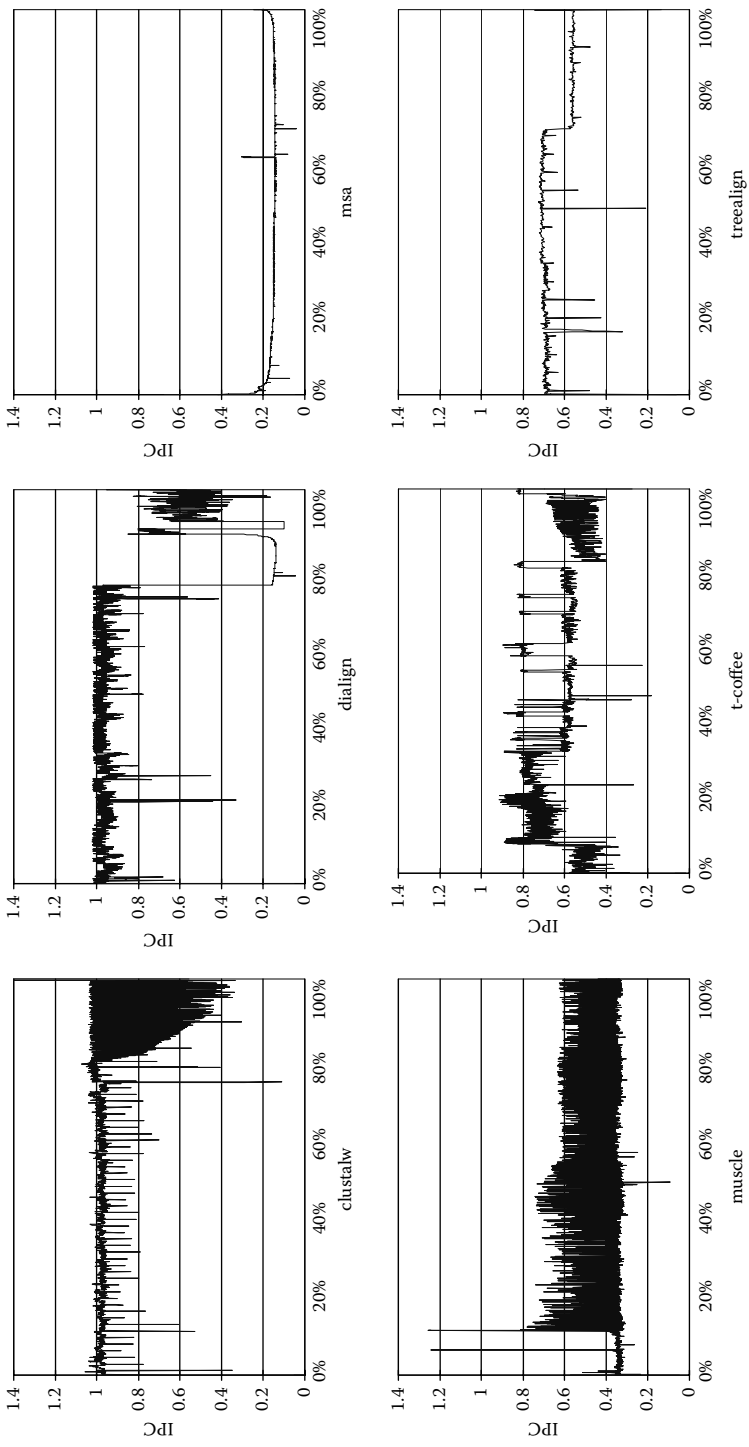


FIGURE 11.11
Phase behavior of MSA workloads.

11.5 Conclusions

As requirements for the processing of biological data grow, bioinformatics becomes an important type of application domain. The assembly of a multiple sequence alignment has become one of the most common tasks in bioinformatics. Despite the amount of attention dedicated to the MSA problem, it is largely unknown how various MSA methods use the advanced microarchitectural features provided by modern processors. Our work studies architectural properties for several widespread multiple sequence alignment algorithms on an actual Intel Pentium 4 processor.

We found that bioinformatics multiple sequence alignment workloads benefit from many advanced Pentium 4 microarchitecture features such as trace cache, prefetching and large size L2 cache, and advanced indirect branch predictor. We believe that several observations we made in this study can be useful for performance optimization of MSA workloads from the architectural point of view. For example, MSA workloads intensively access (i.e., read) memory and the access patterns can be captured by the hardware prefetcher. Thus, a smaller L1 data cache with multiple read ports and prefeteching can provide higher memory bandwidth while reducing cache hit latency. We also observed that despite the relatively good behavior on cache and branch prediction, the IPC performance of MSA workloads is still poor. To fully utilize the superscalar capability provided by the advanced microarchitecture, we believe that additional techniques, such as value prediction [24] and more aggressive compiler optimizations (e.g., superblock [25] and hyperblock [26]), should be used.

The results obtained in this chapter open up new avenues for future MSA algorithms. For example, to reduce excessive amounts of loads and stores, heuristic methods can be applied to MSA algorithms to further reduce the amount of search space explored. To effectively reduce branch misprediction rates and pipeline flushes, new MSA algorithms should explore the search space more deterministically. That is, unpromising alignments need to be eliminated pre-emptively using better strategies. These improvements can be obtained by summarizing and indexing the search space and statistically analyzing the sequences. In future work, we will explore the hardware and software techniques to optimize the performance of MSA tools.

References

[1] http://www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[2] Bioinformation Market Study for Washington Technology Center, Alta Biomedical Group LLC, www.altabiomedical.com, June 2003.

[3] C. Notredame, Recent progress in multiple sequence alignment: A survey, *Pharmacogenomics*, Jan. 3(1): 131–144, 2002.
[4] L. Wang and T. Jiang, On the complexity of multiple sequence alignment, *Journal of Computational Biology*, 1(4): 337–348, 1994.
[5] D. W. Mount, *Bioinformatics: Sequence and Genome Analysis*, Cold Spring Harbor Laboratory Press, 2001.
[6] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, The microarchitecture of the Pentium 4 processor, *Intel Technology Journal*, 1st quarter 2001.
[7] D. Lipman, S. Altschul, and J. Kececioglu, A tool for multiple sequence alignment, *Proc. Natl. Acad. Sci. USA* 86: 4412–4415, 1989.
[8] J. D. Thompson, D.G. Higgins, and T.J. Gibson, Clustal W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting, positions-specific gap penalties and weight matrix choice, *Nucleic Acid Research*, 22(22): 4673–4680, 1994.
[9] J. J. Hein, TreeAlign, in *Computer Analysis of Sequence Data*, edited by A. M. Grffin and H. G. Griffin. Humana Press, Totowa, NJ, pp. 349–346, 1994.
[10] C. Notredame, D. Higgins, J. Heringa, T-Coffee: A novel method for multiple sequence alignments, *Journal of Molecular Biology*, 302: 205–217, 2000.
[11] C. Lee, C. Grasso, and M. Sharlow, Multiple sequence alignment using partial order graphs, *Bioinformatics* 18: 452–464, 2002.
[12] C. B. Do, M. Brudno, and S. Batzoglou, ProbCons: Probabilistic consistency-based multiple alignment of amino acid sequences, *ISMB* 2004.
[13] C. Notredame and D.G. Higgins, SAGA: Sequence alignment by genetic algorithm, *Nucleic Acid Research*, 24: 1515–1524, 1996.
[14] R. C. Edgar, MUSCLE: Multiple sequence alignment with high accuracy and high throughput, *Nucleic Acids Research* 32(5): 1792–1797, 2004.
[15] M. Kimura, A simple method for estimating evolutionary rates of base substitutions through comparative studies of nucleotide sequences, *Journal of Molecular Evolution*, 16: 111–120, 1980.
[16] N. Bray and L. Pachter, MAVID: Constrained ancestral alignment of multiple sequences, *Genome Research*, 14: 693–699, 2004.
[17] K. Katoh, K. Misawa, K. Kuma, and T. Miyata, MAFFT: A novel method for rapid multiple sequence alignment based on fast Fourier transform, *Nucleic Acid Research*, 30: 3059–3066, 2002.
[18] B. Morgenstern, DIALIGN 2: Improvement of the segment-to-segment approach to multiple sequence alignment, *Bioinformatics,* 15: 211–218, 1999.
[19] S. R. Eddy, Profile Hidden Markov Models, *Bioinformatics Review*, 14(9): 755–763, 1998.
[20] NCBI, http://www.ncbi.nlm.nih.gov/.
[21] The NCBI Bacteria Genomes Database, ftp://ftp.ncbi.nih.gov/genomes/Bacteria/.
[22] B. Sprunt, The basics of performance monitoring hardware, *IEEE Micro*, July-August, pp. 64–71, 2002.
[23] Intel Pentium 4 Processor Optimization, Reference Manual, Intel Corporation, 2001.
[24] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, Value locality and load value prediction. In *Proceedings of the International Symposium on Computer Architecture*, 1996.

[25] W. W. Hwu and S. A. Mahlke, The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, vol. 7, issue 1–2; 224–233, May 1993.

[26] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, Effective compiler support for predicated execution using the hyperblock. In *The International Symposium on Microarchitecture*, 1994.

[27] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, Automatically characterizing large scale program behavior. *In Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2002.

12

Towards System-Level Fault-Tolerance Using Formal Methods and SoC Methodologies

Kristina Lundqvist
Massachusetts Institute of Technology

CONTENTS

12.1 Introduction

Software is ubiquitous in the mission-critical systems that are used today, ranging from embedded systems such as flight control to stand-alone systems that manage international financial flows. Real-time embedded systems are a special class of mission-critical systems, which have to satisfy both dependability and timeliness requirements. The traditional constraints on these systems were processing power and memory availability; however, with the quantum increases in computing power and miniaturization of electronics, the challenges revolve around integrating multiple embedded components within a larger systems context and managing the evolution of technology. Building, evolving, and ensuring confidence in these complex systems in a cost-effective and schedule-compliant manner is becoming increasingly more difficult. As Boehm (1981) noted, the most difficult part of developing a software-intensive system is not in the software development phase itself, but in the design, integration, and verification phases. There needs to be a unifying framework that covers the complete system life cycle, from initial requirements all the way to sustainment and eventual retiring.

The emergence of system-on-chip technologies for rapid prototyping, and the use of formal methods for verification of both the design and the implemented system provide new approaches for building complex real-time embedded systems. These new approaches do not completely mitigate the classic problems of embedded systems development, namely, understanding the impact of the operating system services on system behavior, maximizing processor utilization, and ensuring deterministic behavior of the implemented system without sacrificing design flexibility or system evolution. These new technologies enable the exploration of newer implementation architectures and alternative approaches to provide increased dependability. Model-checking tools can now be used to explore large-state space systems, and models of the application software and underlying operating system services can be effectively composed and verified through guided state space exploration. In this chapter, we present the design of a hardware implemented runtime kernel for an implementation architecture involving multiple processors. We present the design space exploration that was carried out to determine the implementation of delay queues for the kernel.

The remainder of the chapter is organized as follows. The Gurkh framework section details how the approach used within Gurkh differs from the traditional approach used to build real-time embedded systems from the perspective of system architecture, system design, verification, and

ensuring overall system dependability. The Foundations section details the building blocks used, namely the underlying operating system model, the formal modeling and verification tools, and the prototyping tools. The Gurkh Framework Components section provides a high-level overview of the four elements of the framework and details the first hardware implemented runtime kernel that was prototyped for a single-processor environment, and motivates the need for supporting multiprocessor environments. The RavenHaRT-II section provides the overall architecture of the new RavenHaRT-II kernel, and the Modeling and Implementation sections cover the design space exploration of queues at both analysis and implementation levels. The chapter concludes with a recap of the results and provides guidance on further work.

12.2 The Gurkh Framework

The Gurkh framework was created as a first step towards the development of an integrated framework that covers the life cycle of a mission critical real-time embedded system (Asplund and Lundqvist, 2003). The framework draws from the domains of concurrent software design, formal verification and hardware-software co-design, to ensure efficient design space exploration, maximum utilization of computational resources, and high-confidence in the implemented system. The differences between the Gurkh approach and the traditional approach to mission-critical systems development across multiple levels of abstraction are summarized in Table 12.1.

TABLE 12.1

Contrasting the Traditional Approach to the Gurkh Approach

	Traditional Approach	Gurkh Approach
Architecture	Stable COTS processor/microcontroller	Stable COTS processor with programmable hardware components
Design	Cyclic executive-based scheduling	Priority-based pre-emptive scheduling
	Software operating system	Hardware implemented operating system
	Application capability implemented solely in software	Application capability partitioned between hardware and software
Verification	Formal verification of software (excluding OS)	Formal verification of system (application and OS)
Dependability	Intrusive monitoring or redundancy	Nonintrusive monitoring and reconfiguration

12.2.1 System Architecture

The traditional approach to embedded real-time system development has been to select a commercial off-the-shelf microprocessor or microcontroller as the underlying hardware platform, and corresponding operating system, over which the application software is developed. This approach is extremely effective in moving all remaining design decisions to the application level, but significantly constrains the design space that can be explored. Recent advances in the three areas of hardware-software codesign, prototyping technology, and formal modeling and verification enable more effective exploration of the design space through prototyping and simulation. In the Gurkh framework, the underlying system architecture is defined by the Xilinx Virtex-II Pro platform architecture, wherein software runs on an embedded PowerPC, and additional system capabilities can be implemented on the FPGA.

12.2.2 System Design

The four major tasks in the design of an embedded system are (Wolf, 1994):

- Partitioning the requisite capabilities into interacting components
- Allocating the components to specific computational elements
- Scheduling the times at which the functions are executed on a given computational element
- Mapping the specification to an implementation

As was highlighted in the system architecture section, using traditional approaches simplifies the embedded system design problem to one of selecting the right operating system, and scheduling the application tasks. The most widely used scheduling paradigm is the cyclic executive (CE) approach, where the execution of several processes on the CPU is explicitly and statically interleaved. This leads to a deterministic system from the ground up, but exhibits a crippling inflexibility in the sense that the slightest modification of the system often requires a complete redesign of the predetermined schedule. Given that the schedule generation process is known to be an NP-hard problem, the CE approach does not scale as the number of processes increases. Furthermore, the necessity for tasks to share a harmonic relationship imposes artificial timing requirements (Vardanega, 1999) that can be wasteful in processor bandwidth (Punnekkat, 1997).

The approach adopted in Gurkh on the other hand is to use a hardware-implemented runtime kernel (either RavenHaRT (Silbovitz, 2004) or RavenHaRT-II (Naeser and Lundqvist, 2005)) to provide operating system services, and enable the use of priority-based pre-emptive scheduling techniques to manage the execution of the application processes. This approach provides flexibility along two dimensions: it allows for postponement of the mapping decisions to hardware and software components until later in the development life cycle, and provides greater flexibility in application design.

Unlike the CE approach, minor design changes result mostly in small implementation modifications.

12.2.3 System Verification

In the traditional development approach, formal verification is restricted to the application, as the underlying operating system may not have a formal definition. The formal basis of the RavenHaRT kernel allows for the formal verification of the complete system (both the application and the operating system). The tools within the Gurkh framework enable formal verification at both the design and integration stages.

12.2.4 System Dependability

Dependability is obtained in mission-critical embedded systems through a combination of fault prevention, fault removal, fault tolerance, and fault forecasting. Fault prevention and removal are more effective in the design stages; fault tolerance and forecasting are critical during the operational stages of the system. Given the wide recognition of the fact that there is no cost- and schedule-effective means of completely eliminating all faults from the system prior to its fielding, fault tolerance is critical in mission-critical embedded systems. The traditional approach (Abbott, 1990) to fault tolerance is use of intrusive monitoring (watchdog timers, software monitors) to detect errors, and recover either through forward error recovery or backward error recovery through a redundant system, and provide continued service. If intrusive monitoring is being used for error recovery, additional software has to be added to the application, changing the timing behavior of the system. In mission-critical embedded systems, this addition of monitoring software changes the timing analysis performed on the system, the overall scheduling of tasks, and consumes valuable processor resources. In the Gurkh approach, system dependability is provided through the use of a monitoring chip (MC), which provides nonintrusive monitoring for error detection, and a modified backwards error recovery approach to provide continued service (Gorelov, 2005).

12.3 Gurkh Framework Foundations

The Gurkh framework is built around the Ravenscar tasking profile of the Ada 95 programming language. It exploits the analysis capabilities provided by the UPPAAL toolset, and leverages the prototyping capabilities provided by the Xilinx Virtex-II Pro.

12.3.1 The Ravenscar Tasking Profile

The core of the Ada language is mandatory for all language implementations, also known as profiles. A set of annexes is defined to extend the language in

order to fulfill special implementation needs. The Ravenscar profile (Burns et al., 1998, 2003) for Ada 95 defines a safe subset of the Ada language features. From the annex perspective, the real-time annex is mandatory for Ravenscar. The profile does not allow tasks to be dynamically allocated (other than at software start time), and allows only for a fixed number of tasks. None of the tasks may terminate, hence each consists of an infinite loop. Tasks have a single invocation event that can be called an infinite number of times. The invocation event can be time-triggered or event-triggered. Time-triggered tasks make use of a *delay until* statement.

Tasks can only interact by using shared data in a synchronized fashion through the use of protected objects (POs). POs may contain three different types of constructs, the Protected Function, the Protected Procedure, and the Protected Entry. A Protected Function is a read-only mechanism, whereas the Procedure is a read-write mechanism. The Protected Entry is associated with a Boolean barrier variable and both implement a mechanism used for event-triggered invocation of tasks.

12.3.2 The UPPAAL Model Checker

Model checking has most often been applied to hardware design, but has also been shown very useful for software design. Model checking is a method that algorithmically verifies a formal system by verifying if the model of the hardware or software design satisfies a formal specification written as a set of temporal logic formulas. The UPPAAL model checker tool suite (Larsen et al., 1997; Behrmann et al., 2004) contains an editor, simulation tool, and verification tool for networks of timed automata. A timed automaton is a finite-state automaton, augmented with time, clocks, Boolean variables, integer variables, and synchronization channels. Shared variables and synchronization channels can be used by two or more automata to communicate data and synchronize.

Each automaton consists of an initial location, indicated by an inner circle, a fixed number of locations and transitions between locations. In the explanation of the queues below, the notation $n_1 \rightarrow n_2$ represents a transition from location n_1 to location n_2. Transitions can contain guards, synchronizations, and assignments. An automaton can transition from a location if the guard on the transition is satisfied. When a transition is taken, the assignment part of the transition is executed. During a synchronous step, where two automata communicate over a channel, the assignments of the sending automaton are made before those of the receiving automaton. A transition can synchronize at most on one channel. An exclamation mark after the channel name is used to indicate that the channel is used for sending and a question mark is used to indicate receiving. Locations can be marked as committed or urgent to force specific temporal behavior (an encircled c, respectively, an encircled u). Unmarked locations have no restrictions, committed locations can be used to create atomic chains of transitions, and an automaton in a committed location must leave the location before any other noncommitted transition may

be taken in the system. Committed locations can be used to synchronize over multiple channels in a chain of transitions. Urgent locations indicate that outgoing transitions from the location have precedence over time transitions. Time transitions can be taken whenever there are no automata in committed or urgent locations that can make transitions. Failure is reported during verification or simulation if an automaton cannot leave a committed location.

The UPPAAL verification tool is used to explore whether user-defined properties hold in the timed automata model. If a property cannot be verified (proven correct) UPPAAL automatically generates a counterexample that can be explored in the simulator part of the tool.

12.3.3 Prototyping Tools

A system-on-chip (SoC) is an implementation technology that typically contains one or more processors, a processor bus, a peripheral bus, a component bridging the two buses and several peripheral devices (Mosensoson, 2000). The SoC development platform used in the Gurkh framework consists of Xilinx's Virtex-II Pro hardware (Xilinx, 2004). The ML310 boards used for development have two PowerPC processors each, along with over 30,000 FPGA fabric logic cells and over 2400 kb of block RAM. Xilinx's ISE foundation version 6.2.0.3i, with Xilinx's Embedded Development Kit software version 6.3, along with Mentor Graphics' ModelSim SE Plus 5.8e were the tools used for design entry, simulation, synthesis, implementation, configuration, and verification.

12.4 Gurkh Framework Components

The Gurkh framework, Figure 12.1, consists of four main components:

1. A Ravenscar compliant runtime kernel that can be synthesized on the FPGA in two forms: as RavenHaRT for single-processor environments (Silbovitz, 2004), and RavenHaRT-II in multiprocessor environments (Naeser and Lundqvist, 2005).
2. A set of tools for translating VHDL (Nehme, 2004) and Ada (Naeser, 2005) to both an intermediate formal notation (Naeser et al., 2005) as well as timed automata. The intermediate formal notation is used to enable translation across various tools, as well as for visualization purposes. The timed automata representation is used for verification of timing and behavioral properties of the application in conjunction with the runtime kernel (RTK).
3. An Ada Ravenscar to PowerPC and RTK cross-compiler called pGNAT (Seeumpornroj, 2004).

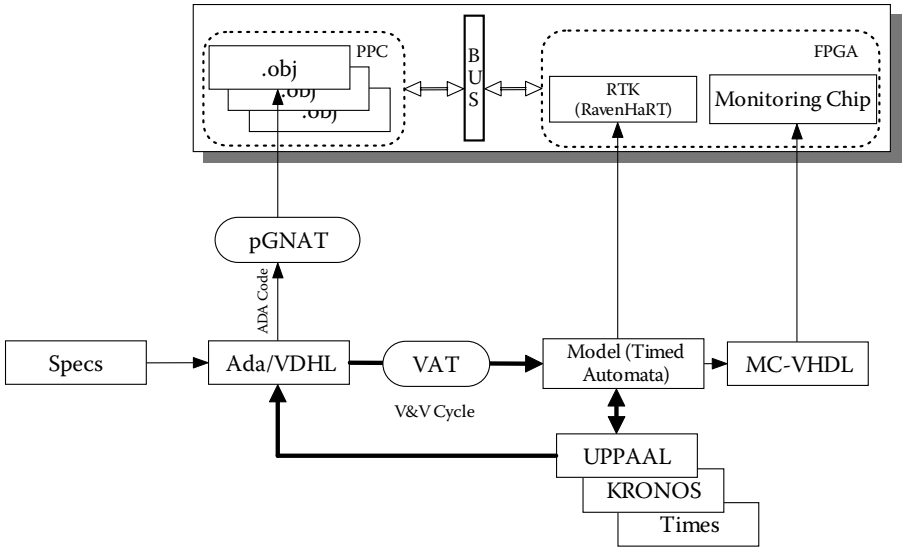


FIGURE 12.1
Uniprocessor instantiation of the Gurkh architecture.

4. A nonintrusive online hardware monitoring device, called the monitoring chip (MC), created using a model of the target system application (Gorelov, 2005).

Real-time embedded systems have strict timing requirements, and must satisfy the additional constraints of predictability and determinism. This creates a significant challenge when using a traditional software-implemented operating system (OS), particularly when multitasking must be done in a system with a single processor. When the OS also runs on the processor, in addition to application tasks, the OS interrupts the processor at regular intervals by performing clock-tick interrupts. When interrupted in this manner, the processor must stop the task it is running so that the OS can check to see if another task should be running instead, and then resume. Even if the same task continues to run, the interrupt still occurs. This results in less effective processor utilization. In addition, the time taken for actions such as scheduling varies with the number of tasks, which introduces jitter and makes the whole system less deterministic.

To save processor time and increase determinism, many of the capabilities of a software OS can be implemented in hardware including task handling (such as creation, deletion, and scheduling), synchronization (such as semaphores, flags, and resource sharing), and timing (such as delays, periodic starts, watchdogs, and interrupts). When all task management is performed in hardware, scheduling is done in parallel to running application tasks, thereby enabling better utilization of processor time. The only necessary processor interrupts occur when a task is changing. This eliminates the need

for clock-tick interrupts, and this change alone can give the processor up to 20% more time for running tasks (Klevin, 2003).

The RavenHaRT kernel (Silbovitz, 2004), was the first hardware implementation of the Ravenscar-compliant kernel specified in Lundqvist and Asplund (2003) for a single-processor environment. Although RavenHaRT was successful in demonstrating the concept, it did not completely address the optimizations necessary to extend the implementation to multiprocessor environments. In order to address three of the critical system-on-chip design challenges of silicon minimization, power minimization, and increased insight into timing behavior, the RavenHaRT kernel was extended to RavenHaRT-II (Naeser and Lundqvist, 2005).

The Open Ravenscar Run Time Kernel (ORK; de la Puente and Zamorano, 2001a; de la Puente et al., 2001b) also implements the Ravenscar profile. Dynamic validation by software faults injection of ORK is described in Maia et al. (2003) where verification of an implemented kernel is attempted. The ORK approach does not suit the RavenHaRT-II kernel because it is specialized in accordance with the final system's actual characteristics. For example, the delay queue can be specialized for the actual task setup when the number of delaying tasks is known or can be easily deduced using code inspection. This kind of optimization will not only help to reduce the size of the final hardware implementation but also reduce the size of the state space during verification and thus allow for larger systems to be verified.

12.5 The RavenHaRT-II Kernel

The RavenHaRT-II kernel provides the basic services for applications running on the embedded PowerPC. These services include support for scheduling application software tasks, communication and synchronization between tasks, handling processor allocation, and access to shared objects. These different tasks of the kernel can be implemented in a modular architecture, with separate components such as the ready queue, the delay queue, the protected object handler, and the interrupt handler as seen in Figure 12.2. This architecture makes it easier to modify the design and implementation of each individual part of the kernel to meet a system's specific demands and requirements in either software or hardware (Naeser and Lundqvist, 2005).

The designs of the queues were modeled using timed automata and verified together with models of both the other kernel operations as well as of an example application, using the UPPAAL model checker. The queues, and the rest of the kernel, are part of the Gurkh framework that enables the analysis of the temporal properties of safety-critical systems that are implemented in both software and hardware. Although there are tools for hardware analysis (Laramie, 2004), the ability to analyze the temporal behavior of the full system, where the system is partially implemented in hardware, made us

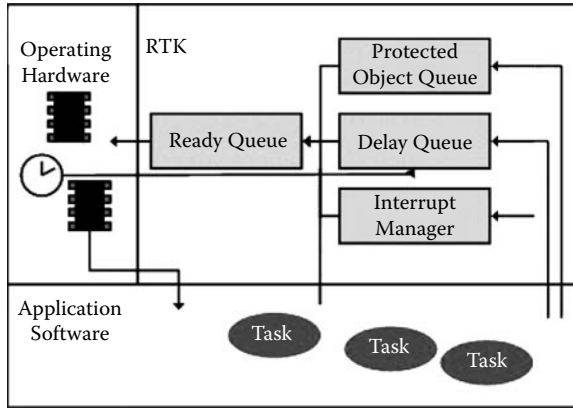


FIGURE 12.2
The RavenHaRT-II architecture.

design the queues and all other parts of the kernel and application using the UPPAAL tool suite. The UPPAAL models of the delay queue, transformation of timed automata to VHDL, and metrics of the FPGA implementation are discussed in later sections.

The desired properties of the RavenHaRT-II kernel are the same as those of software implemented runtime kernels: high speed, predictable behavior, optimal resource utilization, and small size. Timing properties of individual kernel components are also important because they will have a significant impact on the level of possible parallelism. A slower component can become a bottleneck if interacting components operate faster.

The behavior of delay queues is critical to the overall performance of the runtime kernel. Their operation determines the overall efficiency of the kernel. Having accurate models of the delay queues enables optimal utilization of processor resources as well as fine-grain analysis during verification.

The interface of the delay queue is shown in Table 12.2, and the basic operation of the delay queue is as follows.

TABLE 12.2

Delay Queue Interface Description

Input Signals Consumed by the Delay Queue	
delay(T_{id}, *time*)	Delay task T with identity *id*, T_{id}, until time is reached.
tick	Signaled when the system clock increase.
Output Signals Produced by the Delay Queue	
suspend(T_{id})	Remove T_{id} from ready-queue.
unblock(T_{id})	Put T_{id} last within its priority.
runnable(T_{id})	Signal that T_{id} is ready to run.

1. When a task is delayed, a preliminary quick check to decide if the task will be suspended is done.
   a. If the delay time is the current time, or in the past, the task should not be suspended and this is signaled with the *unblock* signal. On receiving an *unblock* the ready queue will move the task to the last position among tasks with the same priority.
   b. If the delay time is in the future, then the task should be suspended and a *suspend* is signaled. Suspension makes the ready queue remove the task from the running tasks and pre-empt it from the processor where it is running.
2. When the release time of a task is reached, the ready queue is signaled to make that task runnable again.

The resources the delay queue uses to store information about the delayed tasks and the way in which it monitors the releases vary in different queue models. Four different queue models are described below, with corresponding different behavior.

12.6 UPPAAL Models of Delay Queues

Minimizing the size of the kernel components allows larger systems to be verified. Some parameters can be changed to optimize the implementation size of the delay queue:

1. The size of the stored delay times
2. How the delays are stored
3. The amount of parallelism used

The behavior of the queue, that is, if and when the queue will cause unwanted stalling of the RTK, depends on the amount of parallelism used in the implementation. Furthermore, the behavior of the queue also depends on whether work is done when delaying or releasing tasks. Some of the parameters depend on each other and some combinations can be eliminated, sorted arrays with all work taking place at release time. The models of delay queues Q_1, Q_2, Q_3, and Q_4 presented below explore different combinations of the parameters.

The delay times can be stored and used as *absolute times* or as *delta times*. An absolute time T is the release time of the task and, as discussed in Zamorano et al. (2001), requires a minimum of 41 bits to represent 50 years at 1-ms resolution, as required by the Ada Reference Manual (ALRM, 2001). However, the number of bits required can be reduced in a system with periodic tasks where the cycle times of all delaying tasks are known. A delta time Δ_T represents the number of ticks remaining until the release of a task and can be

used with a countdown timer to delay tasks. A safe estimation of the number of bits needed for the delta times is the number of bits needed to represent the cycle-time of the task with the longest period.

The array (or queue) where information about the delayed tasks is stored can be managed in two ways, either as a sorted queue ordered by the release times or as an array indexed by the task identities. The two forms of storage increase the work when delaying the queue, or when releasing the indexed array. A delay queue using a sorted list will have to re-sort the queue of delayed tasks when a task is delayed whereas an indexed queue will have to find the next task to release whenever a task is released. At the time a task is delayed the sorted queue can be made to respect the order in which the tasks are released and implemented, for example, FIFO or a priority release policy. An indexed array cannot keep this kind of information and will hence release the tasks in some kind of identifier indexed order. However, a priority-based release can be achieved by ordering the task identities in priority order. Note that the first position of the array is not needed inasmuch as the task ID *zero* is reserved for null processes, as they never delay.

Increasing the amount of parallelism within RavenHaRT-II will reduce the time that kernel components can be blocked by each other, but introduces the possibility of communication delays. Another reason for using parallelism carefully is that it increases the amount of chip area that the hardware implementation will use. To ensure the correct operation of the different delay queues, their behavior is formally verified, using additional models of the other kernel components and sample application systems. Once the queues are verified, the designs are transformed into VHDL and finally synthesized in the FPGA. The designs of four queues are presented and analyzed in following sections.

12.6.1 Delay Queue Q_1

The first delay queue design, shown in Figure 12.3, uses an indexed array of absolute release times. The computation needed to delay a task is minimal, $n_0 \rightarrow n_1 \rightarrow n_0$; the queue writes the release time in the position corresponding to the task in array *DQd*. The queue records in the variable *next,* the index of the task with the closest release time. If there are several releases at the same time the one with the lowest identity is stored. When the release time of *next* is reached, $n_0 \rightarrow n_2$, the task scheduled to be released is made ready to run and the array is searched for the next task to release. In n_3 the first delayed task is found and set to be the next task. Further searching is continued in n_4. If several tasks are scheduled to be released at the same clock-tick, the queue will release all of them, $n_4 \rightarrow n_2 \rightarrow n_3$. Tasks released at the same tick are released in index order to enforce deterministic behavior of the releases. This forced order makes it possible to achieve better performance easily. Ticks from the clock will initiate no action if no task is scheduled to be released, $n_0 \rightarrow n_0$. The worst release case for a single task occurs in the case where all tasks are scheduled to be released at the same time and the task

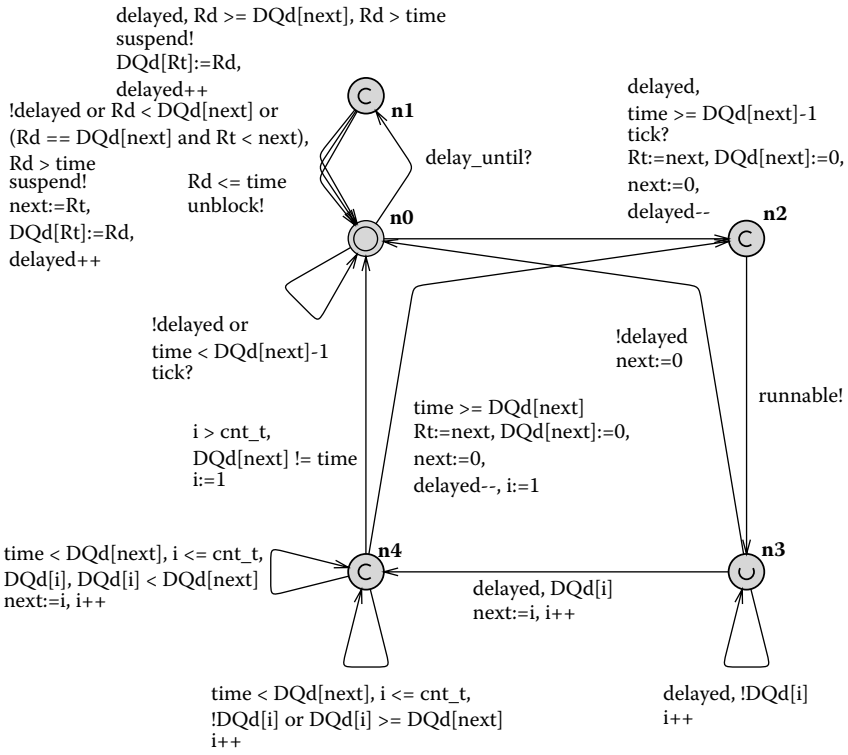


FIGURE 12.3
UPPAAL model of Q_1.

with the highest index would be released after the release of all other tasks. This property is the same for all queues. For Q_1 the expected area requirements on the FPGA for the queue is linear to the number of delaying tasks plus area for some additional variables used to remember the number of currently delayed tasks and the next released task.

12.6.2 Delay Queue Q_2

The second delay queue, shown in Figure 12.4, manages delayed tasks by using a time counter for every delayed task. Each position in the array *DQd* holds a counter for the associated task. The counter for a task is set to the delta time Δ_T when the task is delayed, $n_0 \rightarrow n_1 \rightarrow n_0$. All stored delta times are decremented by one at every system clock-tick, in the transitions leading to and from location n_2. A task associated with a Δ_T decremented to zero during a tick is released using transition $n_2 \rightarrow n_3$. When all positions in the array have been processed, the automaton returns to its initial location and waits for the next tick.

For Q_2 the expected area requirement is lower than that of Q_1 because delta times are used rather than absolute times. No attempt to improve the queue's

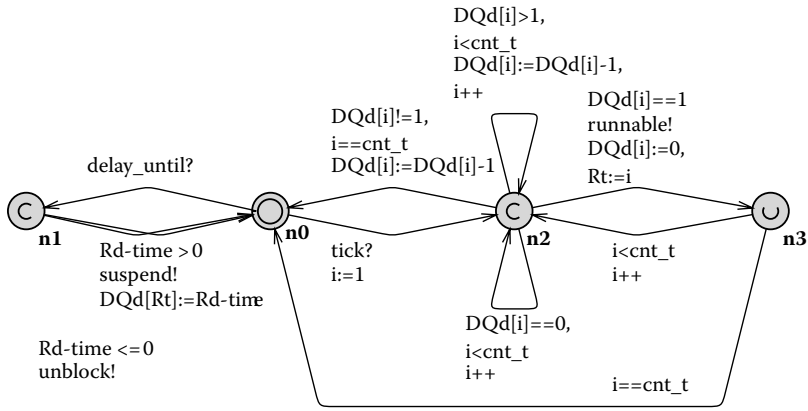


FIGURE 12.4
UPPAAL model of Q_2.

performance by tracking the lowest and highest indices of the delayed tasks has been attempted because the mechanisms for tracking them are likely to be a waste of valuable chip area.

12.6.3 Delay Queue Q_3

The third queue, shown in Figure 12.5, is a parallelized version of Q_2 where each delaying task gets a countdown timer of its own. The delay queue's functionality is achieved by the work of all the counters running in parallel. The parallelism is used to minimize the time to delay and release the tasks. Nodes to the left of the initial node n_0 are used if a task cannot be delayed (tries to delay to a time not in the future), and the nodes to the right of n_0 are used if the task can be delayed. When a task is delayed, the Δ_T for the delay is calculated and stored in a time counter, activating the automaton. The individual values of active counters are decremented at every clock-tick, $n_3 \rightarrow n_3$. When the release time for a task is reached, the counter releases the task and returns to the initial node.

Node n_4 is urgent (and not committed) because there can be several delay queues contending to release their delayed tasks at the same time. In the

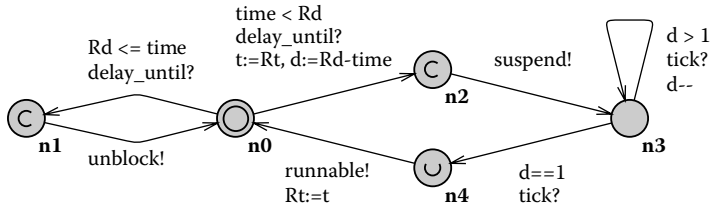


FIGURE 12.5
UPPAAL model of Q_3.

model presented here there is no way to enforce a specific release order of tasks released at the same tick. The UPPAAL tool will explore all possible executions of the releases, and different ways to handle the releases can be found in the queue design analysis section.

The area used by each individual queue is minimal but extra space will be needed to accommodate one state machine for each task that delays. As shown in the implementation section, the area cost for the parallelism is very high.

12.6.4 Delay Queue Q_4

The fourth delay queue shown in Figure 12.6, uses two memory arrays. The *DQt* array contains the task queue and the *DQd* array is used to store delta times. The delta times represent the remaining time for tasks to be released once a task is the next task to be released. Every time a task is delayed the queue takes the transition $n_0 \rightarrow n_1$. If the task is the only task that is delayed, the queue transitions back to the initial state. If there are delayed tasks, the new task should be queued in *DQt* according to its delta time. *DQt* is a circular queue whose head is pointed at by *next* and the position where the delayed task should be inserted is found using $n_2 \rightarrow n_2$. When the insertion position has been found, all tasks to the right of it are shifted right, $n_3 \rightarrow n_3$. When the delta time of a *next* is reached, $n_0 \rightarrow n_4$, all tasks delayed to the same time will be released using $n_4 \rightarrow n_4$.

A possible area optimization for this queue can be achieved if the greatest number of tasks that can be delayed at a single time is known. If the maximum number of simultaneously delayed tasks is known, the *DQt* array can be reduced to accommodate at most this number of tasks. A possible speed optimization can be achieved by checking if the array should grow at the beginning rather than at the end. If there are n delayed tasks, shifting the

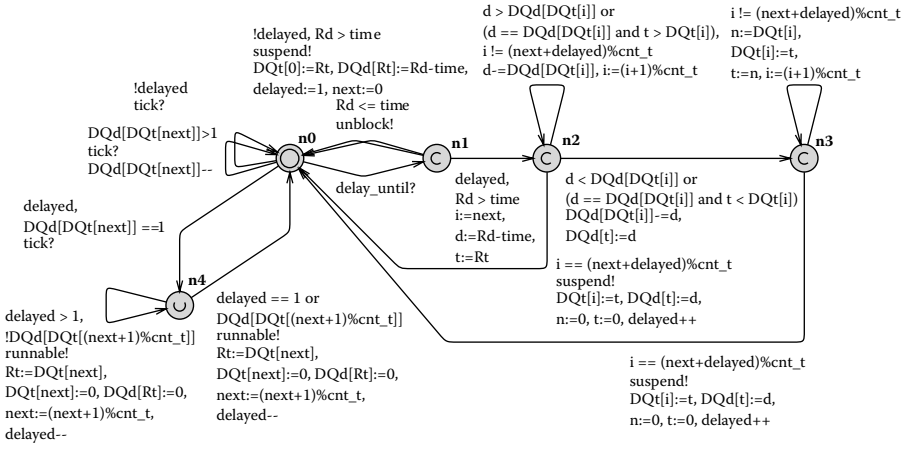


FIGURE 12.6
UPPAAL model of Q_4.

task identities to the left is preferable if the task to be inserted is among the $n/2$ tasks with the nearest release times.

12.6.5 Queue Design Analysis

The delay queue and kernel are designed to be synthesized for a specific target application system. This specialization enables optimizing resource utilization. In the UPPAAL models in the delay queues section, an optimization of the number of bits used to represent delta times was presented. The optimization used knowledge about the cycle-time of cyclic executive tasks. The length of the memory array needed to remember release times can be optimized if delaying tasks are given sequential task identities.

The common procedure when releasing tasks is to lock (stop) the dispatching before releasing tasks, for example, as done in de la Puente et al. (2001a). The need for locking is only necessary in a system where a task of lower priority T_{lo} can be released from the delay queue ahead of a task of higher priority T_{hi} when a batch of tasks is released at the same time. If the dispatcher is not locked, a situation can occur where T_{lo} is loaded on a processor only to be pre-empted when T_{hi} is released. The situation is avoided if tasks are released in priority order, with the release of the highest-priority task first. It is safe to dispatch and start loading T_{hi} because no task in the same release batch can force T_{hi} to be pre-empted. A FIFO order within each priority makes the release behavior even more deterministic if several tasks can have the same priority.

In a system where all tasks have their unique priorities, the index order can be used as the priority order. Queues Q_1 and Q_2 enforce priority-ordered release if the task indices are ordered in priority order. To achieve FIFO release these queues would have to be extended with memory to store the priority of the tasks and the arrival order of the tasks. The dispatch order of Q_3 can be defined if the communication between the counters and the ready queue follows a protocol, which prioritizes the signals from the counters according to the priorities of the tasks. However, FIFO order is outside the immediate reach of Q_3 because it would place too many requirements on communication or synchronization to be usable. The Q_4 queue releases according to index order and FIFO order. As with the first two queues, Q_4 will have to be extended with more memory to implement FIFO if several tasks can have the same priority.

The Ravenscar profile allows only absolute delays where the release time is given explicitly; that is, there are no relative delays. The main reason for not supporting relative delays is that it makes system analysis easier. Furthermore, the kind of systems the profile focuses on, cyclic executives, uses absolute delays. The four queues presented can be easily extended to handle relative delays by adding an extra interface function that doesn't calculate the delta-times. The formal automata could also be extended in the same way.

The delay queues presented are not limited for use in a hardware RTK but can be used as stand-alone components to help a processor manage delayed tasks. For example, the delay queues could be implemented using a memory-mapped

bus interface to allow them to exist in a hardware–software codesign. The interface should contain the system time and implement the clock-tick generation. The task status, runnable/suspend/unblock, should be included in a readable register and it should also be wired to an interrupt pin at the processor. Additionally, a readable task identity register should be included.

12.7 Implementation

The timed automata models of the four delay queues were manually translated to VHDL state machines. The VHDL state machines were augmented with extra glue logic, for example, for communication, and were then synthesized to the target device. The Xilinx ISE Foundation 6.2.03i tool (Xilinx ISE, 2004) was used for synthesis and the target device was a Virtex-II Pro 2vp7ff672-7 FPGA (Xilinx FPGA). The FPGA has an on-chip PowerPC (IBM) processor on which the tasks are run.

The basic translation of the UPPAAL automata to VHDL finite-state machines (FSM) is straightforward but constructs such as UPPAAL's channels, urgent locations, and committed locations are not present in VHDL. Because these constructs define the timing behavior, they must be handled with care. A transition from an urgent location should be taken before the next system clock-tick. This can be accomplished if the implementation ensures that the state machines finish urgent parts within a system clock-tick. We have ensured this by having the clock speed of the kernel run so fast that all work in an FSM can be completed within time. As described in the Foundations section, committed locations are used for atomic transactions, for example, in the UPPAAL model of Q_1, Figure 12.3, the transitions $n_0 \rightarrow n_2 \rightarrow n_3$ form an atomic chain where the automata receive over a channel and then send over a channel. This behavior can be optimized in the implementation by using separate *Rt* for input (respectively, output; cf. *Rt* and *rdy_Rt* in Table 12.3). We have chosen to translate the communication and synchronization channels of

TABLE 12.3

Hardware Signals/Buses

Signals	
tick	Clock signal generated at system-level frequency that decides the delay accuracy available to the application
delay	Signal triggering the state machines to insert a task (Rt) with delay time (Rd)
delay_end	Signal to synchronize with a bus interface that a delay call has finished
Buses	
Rt	Identity of the task that perform a delay call
rdy_Rt	Identity of the task that will be runnable/suspended/unblock

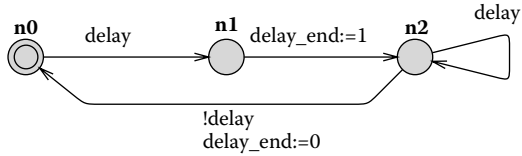


FIGURE 12.7
State graph for communication.

timed automata into a call-and-acknowledge protocol, shown in Figure 12.7. With our translation, the transition $n_0 \rightarrow n_1$ corresponds to a delay call and the transition $n_1 \rightarrow n_2$ corresponds to an acknowledge call. Transition $n_2 \rightarrow n_0$ is used to complete the communication/synchronization sequence. An alternative translation involving a more complex channel implementation is described in Silbovitz and Lundqvist (2003).

Delay queue designs Q_1, Q_2, and Q_4 use arrays to store information such as task identities and delay values. For the target technology we use, arrays can be implemented with registers or with block RAM memory. A register implementation needs larger area because it requires a register to be coded in the FPGA whereas a memory implementation can use the memory blocks of the FPGA. There is a performance penalty for using block RAM, inasmuch as memory access takes one clock cycle, whereas register access takes zero clock cycles; however, in Virtex-II Pro, this penalty is not significant. An example of how we handle block RAM accesses is shown in Figure 12.8. Transition $n_0 \rightarrow n_1$ is the DQt[next] access and transition $n_1 \rightarrow n_2$ is the DQd[DQt[next]] access. The last transition $n_2 \rightarrow n_3$ is the DQt[next]:=0 access. In other words, we try to set the address in advance so as not to lose an extra clock cycle. When this is not possible, we insert an extra state.

To initialize memory arrays, a reset state, shown in Figure 12.9, looping through arrays and initializing variables has been introduced. This location implements UPPAAL's initialization of its variables but is not shown in the state graphs of the different queue implementations, for example, in Figure 12.10.

The hardware signals/buses described in Table 12.3 are a translation of the component design interface shown in Table 12.2. The reset and clock signals are not included because they generally exist in any FSM implementation and do not contribute to the understanding.

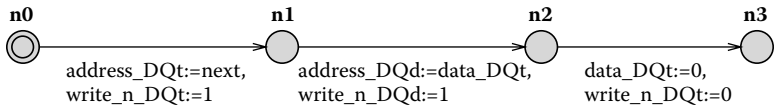


FIGURE 12.8
State graph for assigning the value 0 to DQd[DQt[next]],DQt[next].

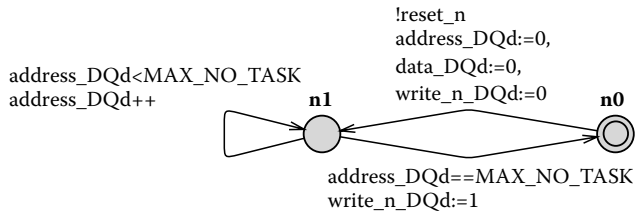


FIGURE 12.9
State graph for resetting the DQd array.

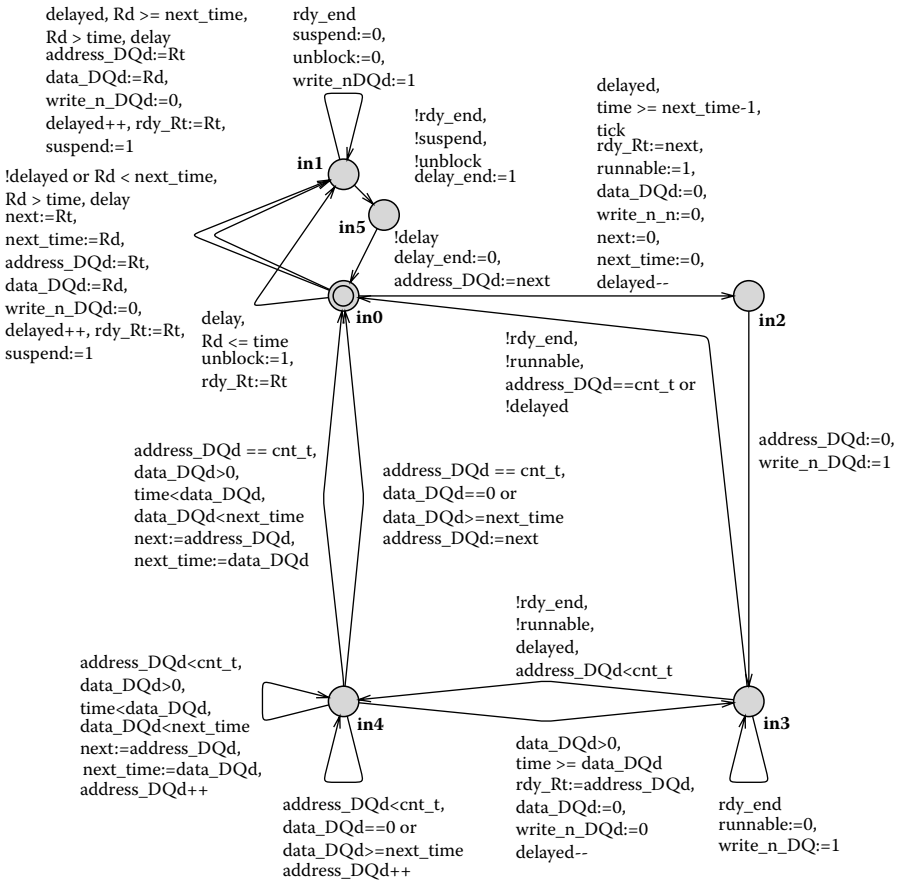


FIGURE 12.10
State graph for delay queue Q_1.

12.7.1 Delay Queue Q_1

The implementation of Q_1 consists of the FSM shown in Figure 12.10. The state machine has the same states as the UPPAAL model (cf. Figure 12.3), and block memory is used to implement the *DQd* array. The transition $n_0 \rightarrow n_1$ is replaced with $in_0 \rightarrow in_1 \rightarrow in_5$ to implement the call-and-acknowledge protocol. The extended parallelism of the implementation allows Rt_rdy to be set to *runnable* when the Rt value is received, which allows $n_4 \rightarrow n_2$ to be translated into $in_4 \rightarrow in_3$. Address_DQd replaces the temporary i variable used for looping in the UPPAAL model. Because Address_DQd is cnt_t bit wide, transition $n_4 \rightarrow n_0$ has been changed to handle the fact that Address_DQd cannot be greater than cnt_t.

12.7.2 Delay Queue Q_2

The implementation of Q_2 is similar to that of Q_1. The state machine has the same states that the Q_2 model has (cf. Figure 12.4), and like the FSM for Q_1 it uses block memory to implement the *DQd* array. The implementation also uses the same variable and location eliminations as described for Q_1. The cycle needed to access the RAM block storing *DQd* is implemented by introducing an extra location in the FSM.

12.7.3 Delay Queue Q_3

The countdown timers of the third queue design are implemented using the finite-state machines shown in Figure 12.11. The state machines, FSM_n with locations $in_0 - in_3$, are an interface for distribution of delayed tasks to their dedicated state machine. The dedicated state machines are described by machine FSM_m with location im_0. FSM_n calculates Δ_T and performs the Δ_T checks to decide if a delaying task should unblock or suspend. FSM_n also informs the ready queue when a task's delay time has expired and the task becomes runnable. The FSM_p, that is, ip_0, serializes the run signals from the FSM_ms for FSM_n. Each FSM_m implements a register counter, which is decremented at each clock-tick until it becomes zero. The machine signals that the task should be released when the counter value to be decreased is one.

The model of the counters (cf. Figure 12.5), corresponds to FSM_n, FSM_m, and FSM_p. The reason for partitioning the model is to save hardware area and to enable priority-ordered task activation. If priority-ordered release is used, it can be managed by FSM_p. The area required by the design is reduced by using one interface state machine FSM_n, instead of multiple similar interfaces for each FSM_m. The choice of using a register rather than the smaller memory blocks is that the access speed mentioned above, one clock cycle, applies to the access of a RAM block that no other FSM uses. Using a block to hold information about a single task is resource waste and having several queues use and access the same block will be another implementation of Q_2.

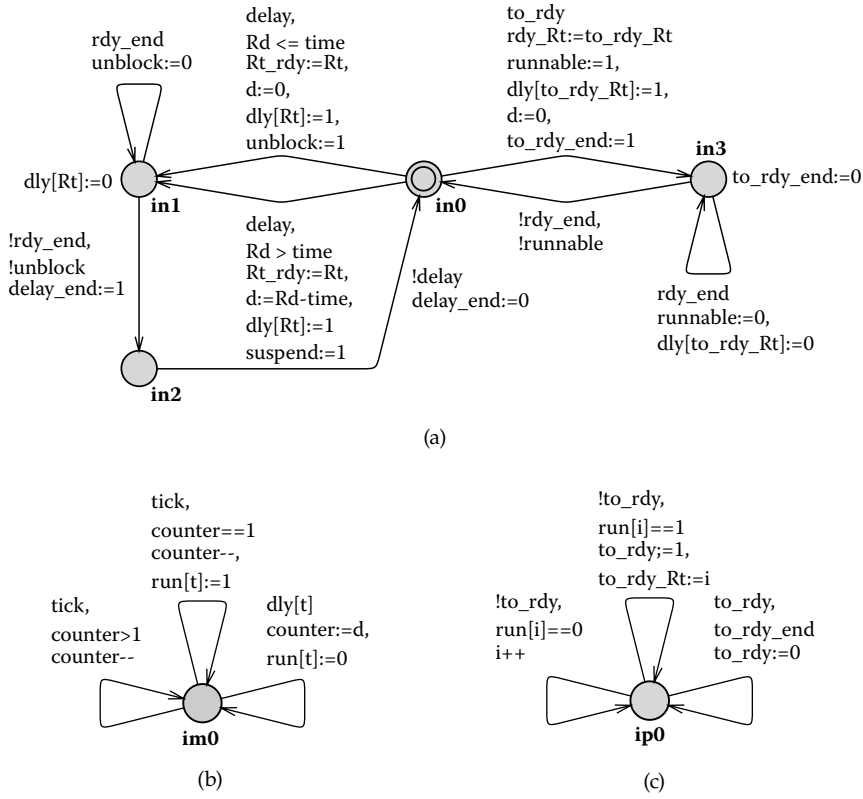


FIGURE 12.11
State graphs FSM_n, FSM_m, and FSM_p for delay queue Q_3.

12.7.4 Delay Queue Q_4

The FSM created while implementing Q_4 closely resembles the automaton in Figure 12.6. Both the time-counter array and task-queue array, *DQd* and *DQt*, are implemented in block memory. The same variable and location eliminations described for Q_1 and Q_2 are used and extra locations are added to handle the extra clock cycles needed when accessing the *DQd* and *DQt* memories.

12.8 Results

To investigate the properties of the implementations, we synthesized systems with different numbers of delaying tasks and timer widths. The bit times are selected to represent systems with high rate cyclic executives (16-bit time), medium rate CE (32-bit time), and finally 41-bit time to handle the 50 years required by the Ada standard.

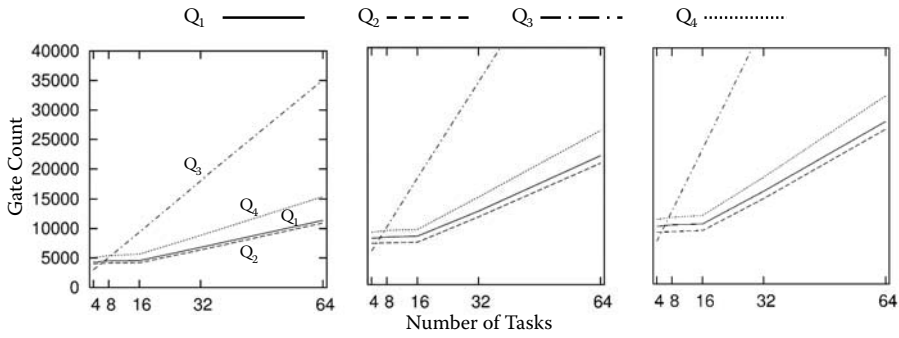


FIGURE 12.12
Gate usage of the four queue implementations.

12.8.1 Area

The results we present in this section are based on synthesis with the clock timing constraint set to 10 ns, that is, creating a kernel running with a kernel clock frequency of 100 MHz, and without any area constraints. In addition, the synthesis tool default settings have been used. The gate count is roughly equivalent to the chip area used by the implementations. The gate counts used by the synthesized systems are presented in Figure 12.12.

It is not unexpected that it is more efficient to use memory rather than registers when the number of tasks increases. The small gate count growth between 4 and 16 task configurations for designs Q_1, Q_2, and Q_4 is due to the synthesis tool using 16 × 1 memory primitives for the arrays. The size growth of the queues is, as expected, close to linear. Queues Q_1, Q_2, and Q_4 use RAM blocks to store data, and the number of RAM blocks used to implement the single array is the same for Q_1 and Q_2 whereas Q_4 uses a little bit more to implement its two arrays. Because Q_3 does not use RAM blocks to code its registers, the cost of the variables is included in the gate count.

Q_2 uses the smallest area, followed by Q_1 and by Q_4. However, Q_3 uses the smallest area for systems with four delaying tasks, but using registers is not efficient for larger task sets. Furthermore, Q_3 quickly outgrows the other implementations in terms of area.

The memory utilization of the designs is very small compared to that available on the target system. For example, the 4200 gates used by a Q_2 implementation with 16 tasks and 16-bit time is small compared to the 811,008 gates that the target Virtex-II Pro device supports. A 4-task 16-bit time synthesized system for any of the queues uses 1–3% of the FPGA's resources in slices, four input LUTs, and slice flip-flops. Queues Q_1, Q_2, and Q_4 use 6–10% of the slices and LUTs and 1–2% of the slice flip-flops when synthesized for a 64-task system with 41-bit time. However, the register queue, Q_3, uses about 80% of the available slices and LUTs and close to 30% of the slice flip-flops for this configuration. Fitting the queues, in addition to the register queue,

on the target FPGA can be easily accomplished, and the majority part of the resources is left for the rest of the kernel and other system components.

12.8.2 Speed

The execution properties of the delay queue implementations depend on the behavior of the rest of the kernel. The communication times between kernel components will influence the execution time of the delay queue. Other kernel components can force the delay queue to wait. For example, when a batch of tasks is released, the ready queue will accept them in serial order at the rate it can process them. The application will also influence the execution time of the delay queue. For example, the application will instantiate the queue with the number of tasks that delay.

Let S_{TDly} be the set of *tasks that delay* in an application and let $|S_{TDly}|$ be the cardinality of that set. Let C_{run} be the number of clock cycles the ready queue uses to make a task runnable and let C_{sus} be the number of clock cycles it uses to suspend a task. The worst-case execution for Q_1 and Q_4 to delay a task occurs when the delay request arrives during the release of a batch of tasks. The worst-case execution time is described in Equation (12.1).

$$((|S_{TDly}| - 1) * (3 + C_{run}) + 1) + (4 + C_{sus}) \tag{12.1}$$

The first part of the expression, $((|S_{TDly}| - 1) * (3 + C_{run}) + 1)$, describes the number of kernel clock cycles used to *time out* all tasks in the delay queue; that is, the task with the lowest priority will have to wait for all other tasks to be handled by the ready queue. The second part, $(4 + C_{sus})$, describes the number of cycles used to manage the insertion of the call into the queue. The worst delay for Q_2, shown in Equation (12.2), is similar to the one of Q_1 and Q_4.

$$((|S_{TDly}| - 1) * (2 + C_{run}) + 1) + (4 + C_{sus}) \tag{12.2}$$

The worst case for Q_3 is different because delays are made to private time counters in parallel. Equation (12.3) takes the shared interface machine into consideration.

$$(3 + C_{run}) + (4 + C_{sus}) \tag{12.3}$$

It is important to note that Q_3 prioritizes delay calls before each clock-tick and that it is possible because it uses one FSM for each task's delay counter. It is not possible to prioritize delay calls with the other delay queues because this would risk a system clock-tick being missed. The frequency of the kernel clock, *KerClk*, needs to ensure that the queue's work, together with any time added by interaction with other kernel components, can be completed within a system clock-tick. If this cannot be guaranteed, then the kernel risks missing system ticks. Moreover, the kernel clock frequency must also support the Ravenscar profile delay accuracy of 1 ms of the system ticks. Table 12.4 shows the maximum clock frequency at which the queues can be synthesized for a 16-task configuration. To check that the queues satisfy the

TABLE 12.4

Maximum Clock Frequency in MHz

Time Width	Q1	Q2	Q3	Q4
16 bit	145	173	283	155
32 bit	142	156	242	144
41 bit	132	150	225	138

1-ms requirement we made a coarse overestimation of the worst number of kernel cycles used to delay a task. We found this number of cycles to be 350 kernel cycles. All these cycles must be completed within a system clock-tick for the operation of the kernel to be guaranteed correct. In Table 12.4 we see that the slowest queue, Q_1, can be synthesized to a maximum of 132 MHz. This speed would allow the system to be synthesized with a system clock frequency of 0.38 MHz, which clearly supports the 1 kHz (1 ms) required by the Ravenscar profile.

The calculation presented here makes no optimizations of the system clock-tick management. The kernel can run at a slower speed by using a buffer for the system clock-ticks. Management using a buffer that can store ticks would allow the delay queue to spread its worst-case work over the number of ticks the buffer can hold. This is based on the simple reasoning that a worst case cannot be followed by another equally bad case because the first case will lead the system to a system state where the equally bad state is impossible. For example, if the worst case is one where all tasks are delayed and released at the same time, they will not be delayed during the next tick, making it impossible to repeat the release. A system with a buffer could make it easier to synthesize the system and produce an efficient hardware kernel.

12.9 Conclusions

This chapter presented formal models and hardware implementations of four delay queues suited for multiple processor systems. The queues express different properties regarding hardware requirements, possible parallelism, and execution times. Different task release policies and how they can be supported by the queues, and translations from the original timed automata designs to VHDL, together with metrics of the hardware implementations have been presented.

Surprisingly, the queue using the most parallelism, Q_3, shows not only the best response time properties but also the least chip area usage for systems where four or fewer tasks use the delay queue. In systems with more than

five delaying tasks, Q_3 quickly outgrows the other queues in terms of area. Otherwise Q_2 uses the least amount of chip area. All queues can meet the Ravenscar profile's timing demand of a granularity of 1 ms.

Although not attempted here, an interesting study would be that of a framework where the properties verified in the initial design, made in a high-level verification tool, could be transformed into properties of the hardware tool used for synthesis to hardware. Enabling verification of the high-level properties could be a step in validating software to hardware translation.

Acknowledgments

I would like to thank Dr. Gustaf Naeser, Johan Furunäs, and Prof. Lars Asplund for their valuable contributions at different stages of this work, and Jayakanth Srinivasan for his engagement and large number of excellent suggestions during the process.

References

Abbott, Russel J. "Resourceful Systems for Fault Tolerance, Reliability, and Safety," *ACM Computing Surveys*, Vol. 22, #1, March 1990, pp. 35–68.

ALRM. 2001. *The Consolidated Ada Reference Manual*. LNCS 2219. Springer-Verlag, New York.

Asplund, L., and K. Lundqvist. 2003. The Gurkh project: A framework for verification and execution of mission critical applications. In *22nd Digital Avionics Systems Conference*.

Behrmann, G., A. David, and K. G. Larsen. 2004. A tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems*.

Boehm, B. 1981 *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, NJ.

Burns, A., B. Dobbing, and G. Romanski. 1998. The Ravenscar tasking profile for high integrity real-time programs. In *Reliable Software Technologies—Ada Europe 1998*. LNCS 1411, Springer-Verlag, New York.

Burns, A., B. Dobbing, and T. Vardanega. 2003. Guide for the use of the Ada Ravenscar profile in high integrity systems. In *University of York Technical Report YCS-2003-348*.

Gorelov, S. 2005. A non-intrusive fault tolerant framework for mission critical real-time systems. Masters Thesis. Massachusetts Institute of Technology, Department of Aeronautics and Astronautics.

IBM Microelectronics and Motorola Inc. The PowerPC microprocessor family: The programming environments. In *IBM Microelectronics Document MPRPPCFPE-01*. Motorola Document MPCFPE/AD (9/94).

Klevin, T. 2003. Multitasking operations require more hardware based RTOSes. In *EE Times.* www.eetimes.com/story/OEG20030221S0027.

Larsen, K., P. Pettersson, and W. Yi. 1997. UPPAAL in a nutshell. In *Int. Journal on Software Tools for Technology Transfer*. Springer-Verlag, New York.

Lundqvist, K., and L. Asplund. 2003. A Ravenscar-compliant run-time kernel for safety critical systems. In *Real-Time Systems*. 24(1).

Maia, R., F. Moreira, R. Barbosa, et al. 2003. Verifying, validating and monitoring the open Ravenscar real time kernel. In *Ada Letters*. XXIII(4).

Mosensoson, G., 2000. Practical approaches to SOC verification, Technical Paper, Verisity Design, Inc.

Naeser, G. 2005. Transforming temporal skeletons to timed automata. In MMRTC report ISSN 1404-3041 ISRN MDH-MRTC-187/2005-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University.

Naeser, G, and K. Lundqvist. 2005. Component-based approaches to run-time kernel specification and verification. In *ECRTS05*.

Naeser, G., K. Lundqvist, and L. Asplund. 2005. Temporal skeletons for verifying time. In *Proceedings of SIGAda 2005*. ACM, New York.

Nehme, C. 2004. The VAT tool, automatic transformation of VHDL to timed automata. Masters Thesis. Massachusetts Institute of Technology, Department of Aeronautics and Astronautics.

de la Puente, J., J. Ruiz, J. Zamorano, et al. 2001a. Open Ravenscar real-time kernel—Operations manual.

de la Puente, J., J. Zamorano, J. Ruiz., et al. 2001b. The design and implementation of the open Ravenscar kernel. In *AdaLetters*. XXI(1).

Punnekkat, S, 1997. Schedulability analysis for fault tolerant real-time systems, PhD. Thesis, University of York, Department of Computer Science.

Seeumpornroj, P. 2004. pGNAT: The Ravenscar cross compiler for the Gurkh project. Masters thesis. Massachusetts Institute of Technology.

Silbovitz, A. 2004. The Ravenscar-compliant hardware run-time kernel. Masters thesis. Massachusetts Institute of Technology.

Silbovitz, A., and K. Lundqvist. 2003. A hardware implementation of a Ravenscar-compliant run-time kernel. In *Digital Avionics Systems Conference*. IEEE.

Vardanega, T. 1999. Development of on-board embedded real-time systems: An engineering approach, In *Technical Report ESA STR-260*, European Space Agency.

Vardanega, T., J. Zamorano, and J.-A. de la Puente. 2005. On the dynamic semantics and the timing behaviour of Ravenscar kernels. In *Real-Time Systems*, 29.

Wolf, W.H., 1994. Hardware-software co-design of embedded systems [and Prolog], *Proceedings of the IEEE* , 82(7): 967–989.

Xilinx Virtex-II Pro Embedded Development Platform Documentation. 2004. ML310 User Guide.

Xilinx Inc. 2004. Xilinx ISE 6 Software Manuals and Help.

Xilinx Inc. 2004. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet.

Zamorano, J., and J. de la Puente. 2002. Precise response time analysis for Ravenscar kernels. In *AdaLetters*. XXII(4).

Zamorano, J., J. Ruiz, and J-A. de la Puente. 2001. Implementing Ada.real_time.clock and absolute delays in real-time kernels. In *6th International Conference on Reliable Software Technologies.*

13

Forward Error Correction for On-Chip Interconnection Networks

Praveen Bhojwani, Rohit Singhal, Gwan Choi, and Rabi Mahapatra
Texas A&M University

CONTENTS

13.1 Introduction

The emergence of networks-on-chips (NoC) as the communication infrastructure alternative to bus-based communication in systems-on-chips (SoC) has presented the SoC design community with numerous challenges. Designing energy-efficient, high-performance, reliable systems requires the formulation of strategies to rectify operational glitches.

The design of low-power systems has highlighted the contribution of interconnect power, up to 50% of total system power [1]. To reduce interconnect energy consumption, voltage scaling schemes are being used, which in turn reduce the circuit's noise margin. The decrease in noise margin makes the interconnect less immune to errors during transmission. Furthermore,

internal noises such as power supply noise, crosstalk noise, and intersignal interference and external noises such as thermal noise, electromagnetic noise, slot noise, and alpha-particle induced noise also present reliability concerns. Thus, combining low-power strategies with data reliability in SoC has become a daunting task for the designers.

As identified by Shanbhag [2], noise mitigation and tolerance are the two alternatives to addressing the reliability concerns. But due to the energy inefficiency of the former, noise tolerance is the preferred approach. In order to deal with interconnect errors in an energy-efficient way, suitable encoding and decoding schemes need to be employed [3]. Reliability concerns can be addressed using either error-detecting codes (EDC) or error-correcting codes (ECC). VLSI self-checking circuits use error-detection codes such as parity, two-rail, and other unidirectional EDCs (m-out-of-n and Berger codes) [4]. Because crosstalk is bidirectional [5], these codes would not be sufficient. Bertozzi et al. [6] make a case for the use of Hamming code [7] on on-chip data buses, highlighting its capability to handle single and double errors, its low complexity, and flexibility as a purely detecting code or a purely correcting code.

For on-chip networks, Worm et al. [8] suggest using Hamming code for error detection and Dumitras et al. [9] utilize cyclic redundancy check (CRC) [10] to detect errors over every hop. Retransmissions are then used to correct the detected errors. When it comes to using ECCs in a design, Bertozzi et al. [6] compare the energy efficiency of forward error correction (FEC) versus error detection and retransmission for on-chip data buses. The reported results indicate that FEC is energy inefficient in described applications. However, the overhead for FEC is expected to subside in emerging NoCs that span large devices using an increasing number of hops and complex buffering/signaling structures. Use of FEC may be cost inefficient when the size of the network is small and the cost of FEC codecs is high. But as network size increases and error rates increase, error detection and retransmission schemes become unacceptable with respect to energy use and latency.

Turbo code [11] is perhaps the most popular code for FEC in communication systems and its coding gain approaches very close to the Shannon limit. Numerous researchers have revealed the high implementation complexity and the high latency associated with the turbo decoders. For the low latency and hardware-overhead requirement of the SoC designs, use of turbo code-based FECs is prohibitive. Hamming codes, on the other hand, can be decoded using simple hardware structures. These, however, have very poor bit error rate (BER) performance when compared to a similar rate turbo code. Rivaling the performance of turbo codes, Gallager [12] proposed a class of linear block codes referred to as low-density parity check (LDPC) codes. This code is suitable for low-latency, high-gain, and low-power design because of its streamlined forward-only data flow structure.

A number of LDPC decoder architectures have been previously reported. Blanksby and Howland [13] demonstrated a 690-mW LDPC decoder. A low-power decoder architecture was also presented by Mansour and

Shanbhag [14]. The objectives of these designs are throughput and very high coding gain. The application targets for these decoders include optical channel, magnetic-media storage, and wireless communication devices among other error-prone devices. And as a consequence, the complexity of decoder designs presented in the aforementioned research is very high and infeasible at the SoC level.

We ascertain that LDPC code decoder design can be tailored to suit the performance and overhead requirements imposed by NoC designs. A novel LDPC decoder design that minimizes the hardware requirement by utilizing only the minimum precision necessary to achieve objective error rate is presented in this research.

Ideally, a transparent forward error-correction scheme is desired for SoC application. Error-correction schemes must be (1) complete, i.e., it does not require interruption to or from the network controller, (2) compact and power thrifty enough to be implemented as an integral component of an on-chip network interface, and (3) yield high coding gain and cover a wide range of error models specific to the SoC design.

This research has the following contributions:

- It presents the case for FEC-based reliability in on-chip networks in high error rate scenarios.
- It presents experimental results using a variant LDPC code that achieves the aforementioned FEC design objectives with remarkable energy efficiency.
- It provides for an improvement in the communication latency, which benefits real-time communications in on-chip networks.

13.2 Preliminaries

This section introduces the concept of the NoC architecture and the basics of LDPC-based FEC. The energy model used to estimate the energy use and the assumptions made in the energy consumption analysis are also presented.

13.2.1 NoC Architecture

Researchers have suggested the use of regular layouts (such as folded torus or the mesh) to integrate intellectual property (IP) cores constituting the SoC [15, 16]. One or more IP cores are placed in a *network tile,* which is the basic building block of the NoC. These tiles are connected to an on-chip network that routes flits between them. Each tile consists of routing logic responsible for routing and forwarding the packets, based on the routing policy of the network. The router logic is constituted of in-ports, out-ports, and a switch fabric to connect them. The NoC architecture and its components are shown in Figure 13.1.

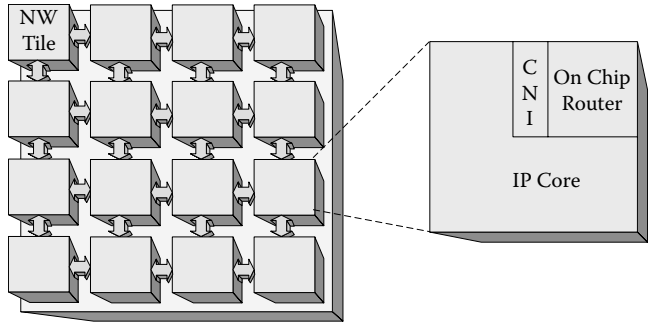


FIGURE 13.1
Generic NoC architecture layout and NW tile structure.

Another important component of the tile is the *core-network interface* (CNI) [17, 18]. The CNI allows the IP cores to speak the "language" of the network. It will also be the site for the error-detection and forward error-correction modules for communication reliability. The following section provides a brief introduction to LDPC-based FECs.

13.2.2 FEC Basics

LDPC codes are linear block codes and have a sparse parity check matrix H. A special class of LDPC codes have H that has the following properties:

- The number of 1s in each column is j.
- The number of 1s in each row is $k > j$.

As with the other linear block codes, encoding is simple and involves matrix operations such as addition and multiplication that can be implemented using a highly regular hardware structure consisting of simple gates. Decoding of LDPC codes is iterative and uses the log maximum-likelihood a priori (LOG MAP) algorithm [19]. There are two decoding methods for LDPC decoder: soft decision and hard decision decoding. In hard decision decoding, the received code-word is sampled as zeros or ones and then the parity check equation is implemented as XORs for each check in the H matrix. If the parity check is satisfied, the code bit is not flipped; otherwise it is flipped. Each code bit will receive j values from the above operation and then do majority voting to decide the final update. This is an iteration of hard decision. In soft decision iteration, quantized values rather than zeros or ones are used for the inputs. The parity check operations involve multiplication of hyperbolic tangent values of the quantized information. In our design, we use only the minimum number of bits to satisfy the precision requirement of the coding gain: one hard decision stage, one soft decision stage with three-bit quantization, and lastly a hard decision stage for $N = 264$ bits to provide optimum frame error rate results. In the context of NoC, frame error rate

(FER) rather than BER is important because retransmission occurs when a frame error is detected.

The code-word size of 264 bits was found to be an ideal size for both the NoC and the LDPC. The rate code for this design was set at 75% and we also used the same rate for the EDCs.

13.2.3 Energy Model

The energy consumed in transmitting a flit (flow digit) from the source to destination network tile—in an error-free environment—can be estimated using the expression:

$$E_{\text{flit}} = (n + 1) * (E_{\text{i}} + E_{\text{o}} + E_{\text{sw}}) + n * E_{\text{link}} \tag{13.1}$$

where E_i is energy consumed in the tile in-port, E_o is energy consumed in the tile out-port, E_{sw} is energy consumed in the tile switch, E_{link} is energy consumed on the link between tiles, and n is the number of hops. This expression is similar to those proposed in [9, 20].

When applied to the different reliability schemes, Equation (13.1) can be modified to estimate the energy consumption pertaining to that implementation.

The energy consumption of the input/output controllers is dominated by the register read and writes. These have been estimated to be 0.075 pJ/bit at 180-nm technology [9]. The energy consumption for the links at 50% driver supply, at TSMC 0.18 micron and using differential signaling has been found to be 0.12 pJ/mm/bit [21]. As mentioned earlier, the flit size being used here is 264 bits.

13.2.4 Motivation

An error detection and retransmission scheme has been shown to be suitable for bus-based communication [6, 8]. Although this solution is elegant for small-length bus design, it needs to be re-evaluated in the context of low-latency requirements of real-time applications mapped onto SoCs. Because the cost of FEC implementation has been a confining factor for earlier researchers, an analysis is needed to determine the strength of the FEC scheme from the above perspective. The results shown in this research will further highlight the benefits of using FEC.

Formulating an optimal design requires determination of a target FEC decoder complexity. Our analysis of energy consumption (see Figure 13.2a) and average flit latency (see Figure 13.2b) over different hop lengths for varying FERs has aided us in making decisions regarding the error-recovery requirement of the FEC. The communication considered here was noncongestive, inasmuch as our goal was to examine the trend in such situations.

The cost for retransmission (energy and average flit latency) has been found to be almost the same for FERs less than 0.01. This allowed us to design a scaled-down LDPC decoder that had a target FER of 0.01.

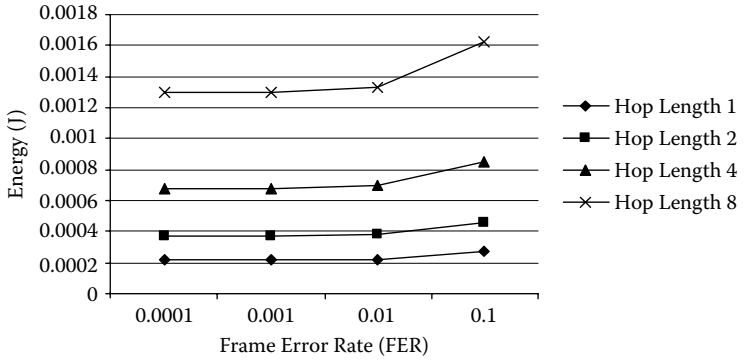


FIGURE 13.2A
Motivation behind selecting target FER for FEC module: energy versus FER at varying hop lengths.

Figure 13.3 shows the FER plot of the number of decoding iterations at different signal-to-noise ratios (SNR). The number of iterations n shown in the figure correspond to initial hard decision iteration followed by $n-1$ iterations of soft decision decoding and a hard decision. From Figure 13.3, $n = 1,2$ has poor FER performance compared to $n = 3,4,5$. Beyond $n = 3$, the performance saturates and hence the $n = 3$ configuration is adopted. This configuration corresponds to an iteration of hard decision decoding followed by a three-bit precision soft decision iteration and a hard decision. This configuration achieves a FER of less than 0.01 for a wide range of operating SNRs. And as determined earlier, the NoC requires a FER of 0.01 for the given error-correction scheme to provide any energy savings.

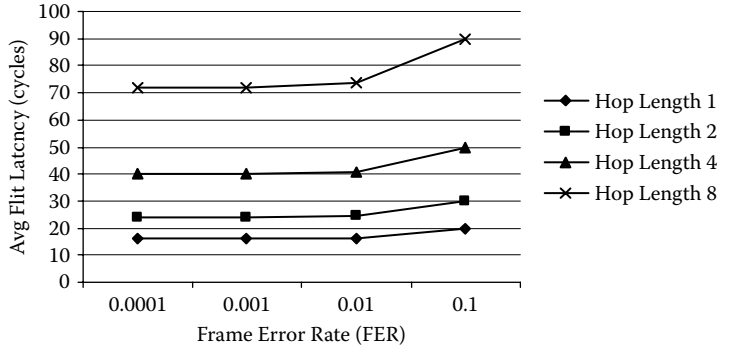


FIGURE 13.2B
Motivation behind selecting target FER for FEC module: average flit latency versus FER at varying hop lengths.

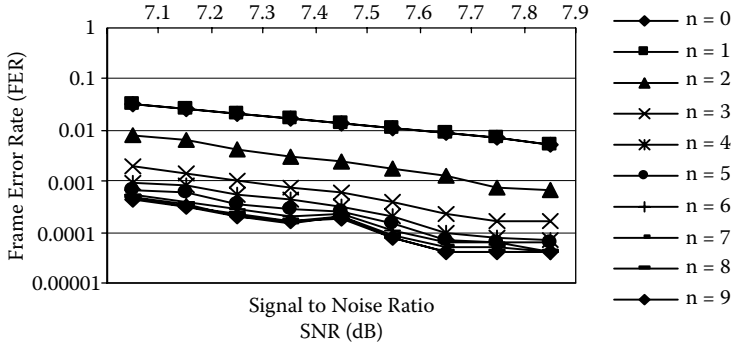


FIGURE 13.3
Frame error rate versus signal-to-noise ratio for varying LDPC iterations.

13.2.5 On-Chip Communication Data Reliability

The challenge of providing cost-effective data reliability in NoCs is not merely protecting the application data. Network control signal (flit headers) reliability is also critical for correct operation of the SoC. We chose to provide independent reliability schemes for both the control and application data lines because most of the strategies discussed in this section are not conducive to providing equal protection to both. This unequal error protection can be tuned at design time to achieve cost efficiency. Because the proportion of the control lines, to those of the data, is comparatively lower, a simple forward error correction through Hamming codes is used to facilitate the control signal reliability.

Application data reliability can be achieved via two strategies:

- Error detection and retransmission (ED + R)
- FEC and limited retransmissions (FEC + R)

The following sections discuss the possible scenarios of operation in each strategy and their associated costs.

13.3 Error Detection and Retransmission (ED + R)

In the ED + R strategy, the transmitter encodes the data to be sent. At the receiver, a decoder determines whether an error has occurred in the transmission. If an error is detected, a retransmission request is made to the sender. ED + R can operate in two scenarios:

- End-to-end
- Hop-to-hop

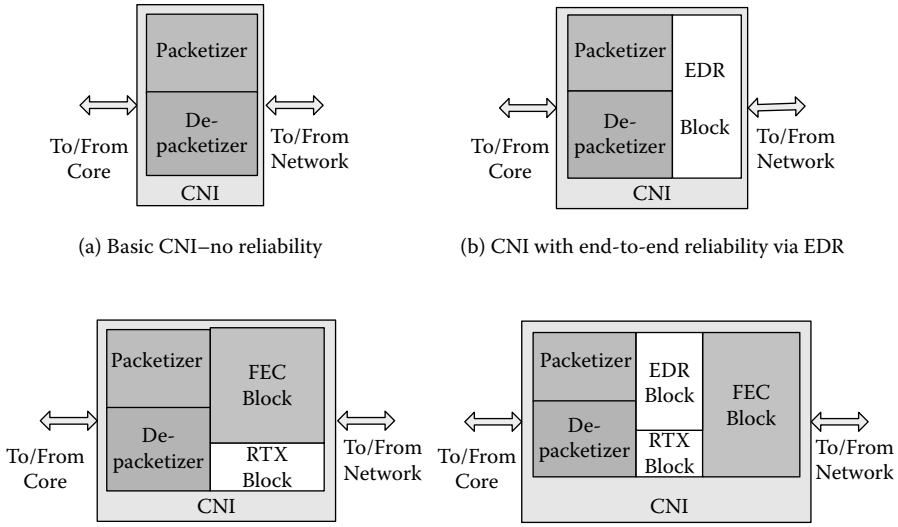


FIGURE 13.4
CNI structure.

13.3.1 End-to-End Retransmission

In this scenario, data is transferred from the source to the destination tile and is checked for errors at the destination CNI. If an error is detected, a retransmission request is made to the source CNI via a negative acknowledgment (NAK) flit. This scheme uses an error-detection and retransmission request module as shown in Figure 13.4b.

The overhead for such a scenario is the need for an encoder and decoder pair in the CNI of every tile. An increased buffer requirement at the sending CNI will be needed to hold flits until they are correctly delivered to the destination tile. Because we did not use a positive acknowledgment (ACK) for correctly received flits, the buffers were periodically purged, based on a time-out. The value set for the time-out will be dependent on the target SoC application and size of the NoC. Traditional issues with using time-outs, like that of lost flits, are not applicable in the NoC designs because we use credit-based communication and so no flits are lost in the network due to buffer overflows.

Extending Equation (13.1), we can estimate the energy consumption in an end-to-end reliability scenario. The energy per flit and the corresponding transmission energy over a noisy network will be:

$$E_{flit} = (n + 1) * (E_i + E_o + E_{sw}) + n * E_{link} + E_r \tag{13.2}$$

$$E_{transmission} = (1 - FEP)E_{flit} + \text{FEP}(E_{transmissionN} + E_{transmissionR}) \tag{13.3}$$

where *FEP* is frame error probability, E_r is the energy cost of providing reliability, $E_{transmissionN}$ is the energy consumed in transmitting the negative acknowledgment, and $E_{transmissionR}$ is the energy consumed in transmitting the original flit. The E_r for CRC was found to be 19.8 pJ at 0.18 microns, whereas that of Hamming was 17.4 pJ at the same technology.

Because (1 – *FEP*) will tend to 1, Equation (13.3) will recursively expand to:

$$E_{transmission} = E_{flit} (1 + 2\ FEP + 4\ FEP^2 + 8\ FEP^3 + \cdots) \tag{13.4}$$

13.3.2 Hop-to-Hop Retransmission

In this scenario, data is transferred from the source to the destination tile, and is checked for errors at every hop through to the destination. This scheme is implemented between the in-ports and out-ports of neighboring network tiles. So in this scenario, each in-port and out-port will have a decoder and encoder, respectively. The buffer requirement in this case is much lower when compared to end-to-end.

As in the case above, energy consumption in the hop-to-hop scenario can also be estimated. The energy per flit and the corresponding transmission energy over a network with a frame-error probability of *FEP* is given by

$$E_{flit} = (n + 1) * (E_i + E_o + E_{sw} + E_r) + n * E_{link} \tag{13.5}$$

$$E_{transmission} = (1 - FEP)E_{flit} + FEP(E_{transmissionR}) \tag{13.6}$$

$$E_{transmission} = E_{flit} (1 + FEP + FEP^2 + FEP^3 + \cdots) \tag{13.7}$$

In the hop-to-hop scenario, the energy consumption term for the negative acknowledgment is absent (because it is only over a single hop).

The expected latency in the hop-to-hop scenario is higher for lower FERs, but it remains lower when the FER exceeds 0.05 (see Figure 13.5). The energy consumption is higher than that of the end-to-end scenario, but at higher FERs, it does not grow as rapidly (Figure 13.6).

The results in Figures 13.5 and 13.6 are for a CRC-based error-detection module. The energy consumption for our design of the Hamming error detector was slightly lower, but it followed a similar trend. Another benefit of such a scenario is the availability of network link status information for network routing reliability purposes.

The cost associated with this implementation, in terms of the area overhead, makes the scheme infeasible for larger NoCs. Each network tile will require four times as many encoder–decoder pairs as compared to the end-to-end scenario. The gate count for CRC was estimated at 1874 gate equivalents at 0.18 microns. When the communication channel becomes noisy, variation in the average flit latency and energy consumption can be as much as 25%. See Figure 13.7. We now take a look at the prospect of using FEC to help circumvent this degradation.

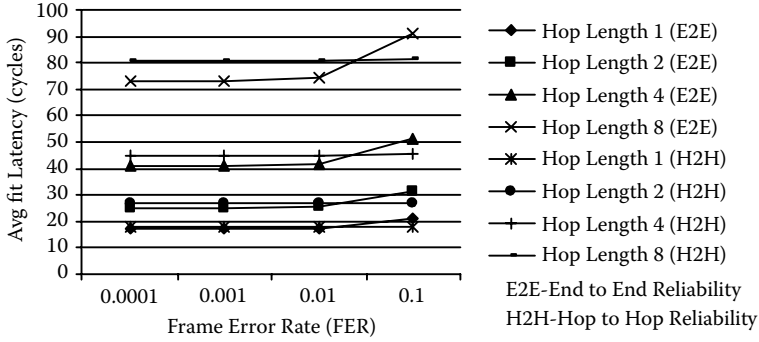


FIGURE 13.5
Average flit latency for end-to-end versus hop-to-hop reliability (CRC).

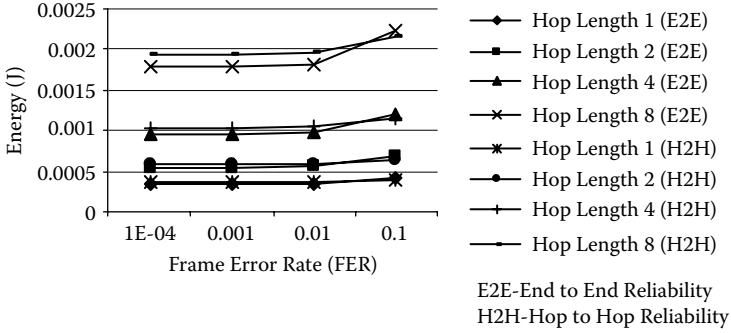


FIGURE 13.6
Energy for hop-to-hop versus end-to-end reliability.

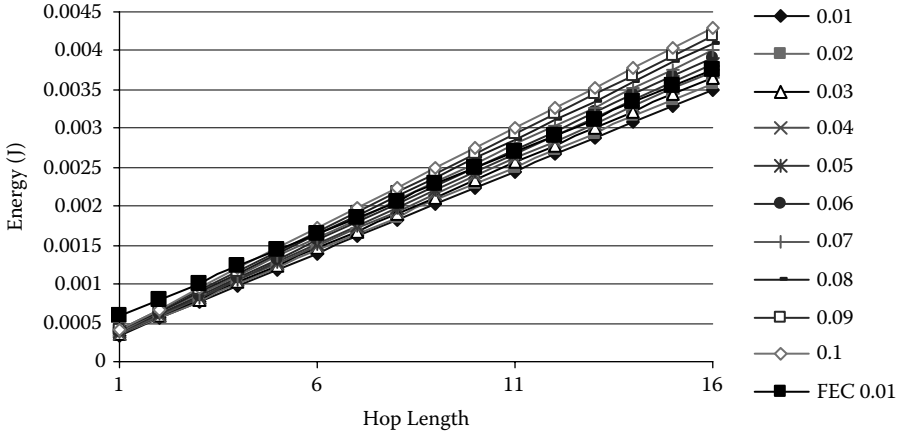


FIGURE 13.7
Flit fields/overheads for different schemes.

13.4 Forward Error Correction (FEC + R)

With the variation in average flit latency and energy consumption going up by about 25% for channels with high FERs, the challenge of meeting communication constraints becomes difficult.

Because the energy consumption in the ED + R strategy—for long hop distances—is dominated by that on the interconnection network, controlling the number of retransmissions is the key to total energy consumption. Using an FEC strategy will reduce the number of retransmissions and also provide for better performance at high FERs and long hop distances.

In general, the use of FECs becomes critical when the communication has real-time constraints, or when the cost of retransmission exceeds that of FEC.

The cost of providing FEC is:

- Area overhead
- Higher energy consumption (when compared to error detection)

The FEC design selected earlier was used to evaluate the FEC + R strategy. The modules of the design were included in the CNI (see Figure 13.4c).

In this strategy the data to be transmitted is encoded with error-correcting codes so as to transmit a code-word that allows for error recovery. The decoder at the receiver then decodes the code-word and extracts the application data. By designing a FEC with a higher FER—to lower implementation cost—the need for retransmission cannot be eliminated.

The last stage of FEC decoding is a hard decision checksum calculation of code-word bit values. If any of the checksum output is high, it is an indication that an error is present in the bit values and the code-word has not converged/corrected to a valid word. A logical OR operation of all check-values is carried out to determine if retransmission is necessary.

FEC + R will only be used in an end-to-end fashion, because the area and energy cost for a hop-to-hop implementation will be very high. Our gate count estimate for the LDPC architecture at 0.18 microns TSMC was 27,126 gate equivalents (per tile). Although this value may seem high, it is comparatively lower with respect to the routing elements.

The energy evaluation of this scheme will be similar to Equations (13.2) through (13.4). The difference is in the E_r term, which reflects the energy consumption of the FEC scheme. E_r for LDPC was found to be 262.1 pJ at 0.18 microns TSMC and was obtained via synthesis with Cadence tools.

The FEC decoder that was used provided an improvement in the energy consumption for FERs over 0.04 (see Figure 13.7). The average flit latency was better for FERs above 0.01 (see Figure 13.8).

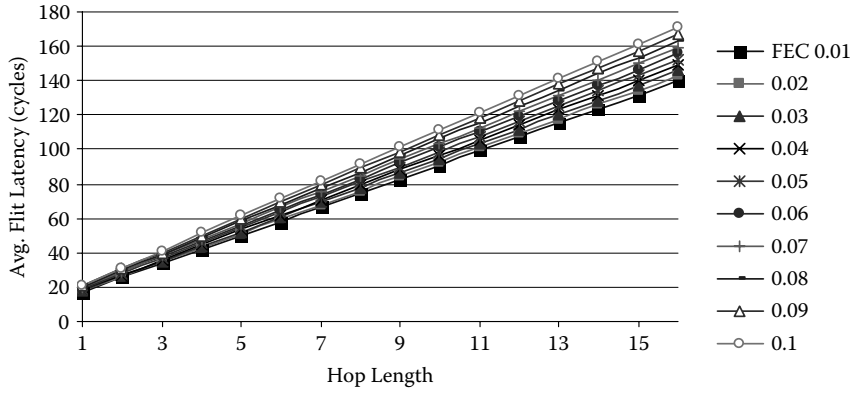


FIGURE 13.8
Energy of ED + R and FEC + R versus hop length for varying FERs.

13.5 Hybrid Scheme (FEC/ED + R)

Identifying the strengths of the aforementioned strategies, we developed a hybrid scheme that uses error detection and retransmission for shorter hop distances, and a FEC scheme for longer distances and real-time constrained communications. We introduced a simple controller into the CNI, whose function is to decide on the type of error reliability scheme that is to be used for the transmitted flit. This scheme required the presence of both ED + R and FEC modules in the CNI (see Figure 13.4d). To attain target energy and latency constraints, a compromise will be required towards the network logic area.

For the hybrid scheme to operate, the decision between the reliability schemes has to be obtained from either the transmitting IP core or the routing table. To achieve energy efficiency through the FEC scheme over long

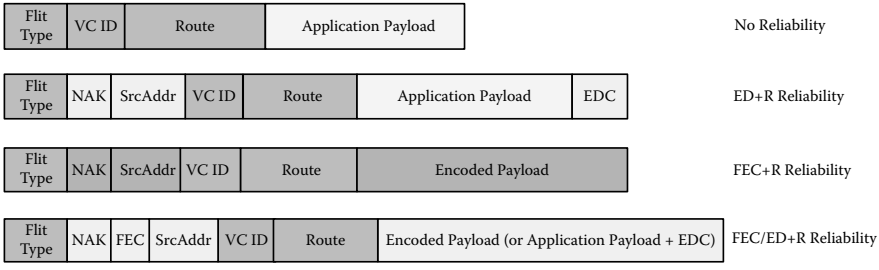


FIGURE 13.9
Average flit latency for ED + R and FEC + R versus hop length at varying FERs.

hop distances, we determined the hop distance beyond which it would be beneficial. We obtained the crossover point from Figure 13.8. Therefore if the operational FER were to be around 0.09, the crossover point would be 6 hops. So communications beyond 6 hops would use FEC-based communication to control the energy consumption.

Figure 13.9 enumerates the structure of the flits used in our experiments. The header contents and field sizes are dependent on the size, topology, and routing policy of the network.

13.6 Summary and Conclusions

This research compares the energy and latency performances for error detection with retransmission (ED + R), forward error correction with retransmission (FEC + R), and the hybrid scheme. The ED + R scheme was implemented for two scenarios: end-to-end and hop-to-hop. The end-to-end scenario provides for better energy and latency performance at low error rates. But the graceful performance degradation in the hop-to-hop scenario would make it more attractive. The area overhead for the hop-to-hop scenario is four times larger when compared to end-to-end. But for high FERs, the degradation in latency and energy is as much as 25% and this may not be acceptable for communication with real-time constraints.

The FEC + R scheme was used to address the performance degradation at high FERs. We formulated a streamlined LDPC-based FEC decoder to provide reliability at an FER of 0.01 (inasmuch as the benefit for a lower FER is negligible). The energy efficiency of this scheme at long hop distances and the corresponding reduction in the average flit latency make a strong case for FEC-based reliability.

To obtain maximum energy efficiency, we designed a hybrid scheme that utilized ED + R for shorter hop distances and FEC + R for communication over long hop distances and with real-time communication constraints.

The results obtained here make a case for FEC-based communication reliability in large on-chip networks under lower noise thresholds.

References

[1] D. Liu and C. Svensson, Power consumption estimation in CMOS VLSI chips, *IEEE J. of Solid-State Circuits*, vol. 29, 1994, pp. 663–670.

[2] N. R. Shanbhag, Reliable and efficient system-on-chip design, *IEEE Computer*, vol. 3, 2004, pp. 42–50.

[3] V. Raghunathan, M. B. Srivastava, and R. K. Gupta, A survey of techniques for energy efficient on-chip communication, in *Proc. IEEE Design Automation Conference (DAC)*, 2003, pp. 900–905.

[4] R. L. Pickholtz, *Digital Systems Testing and Testable Design*. New York: Computer Science Press, 1990.
[5] C. Metra and B. Ricco, Optimization of error detecting codes for the detection of crosstalk originated errors, in *Proc. IEEE,* (DATE), 2001.
[6] D. Bertozzi, L. Benini, and G. De Micheli, Low power error resilient encoding for on-chip data buses, in *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2002, pp. 102–109.
[7] S. Lin and D. J. Costello, *Error Control Coding: Fundamentals and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1983.
[8] F. Worm, P. Ienne, P. Thiran, and G. De Micheli, An adaptive low-power transmission scheme for on-chip networks, in *Proc. 15th International Symposium on System Synthesis (ISSS)*, 2002, pp. 92–100.
[9] T. Dumitras, S. Kerner, and R. Marculescu, Towards on-chip fault-tolerant communication, in *Proc. Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2003.
[10] A. Leon-Garcia and I. Widjaja, *Communication Networks*. New York: McGraw-Hill, 2000.
[11] C. Berrou, A. Glavieux, and P. Thitimajshima, Near Shannon limit error correcting codes and decoding, in *Proc. IEEE Intl. Conf. on Communications*, 1993, pp. 1064–1070.
[12] R. Gallager, Low-density parity-check codes, *IRE Transactions Information Theory,* vol. IT-8, 1962, pp. 21–28.
[13] A. J. Blanksby and C. J. Howland, A 690mW 1-Gb/s 1024-b, rate-1/2 low density parity-check code decoder, *IEEE J. Solid State Circuits,* vol. 2, 2002, pp. 402–412.
[14] M. M. Mansour and N. R. Shanbhag, Low-power VLSI decoder architecture for LDPC codes, in *Proc. Intl. Symp. Low Power Electronics and Design*, 2002, pp. 284–289.
[15] B. Towles and W. J. Dally, Route packets, not wires: on-chip interconnection networks, in *Proc. Design Automation Conference (DAC)*, 2001, pp. 684–689.
[16] S. Kumar, A. Jantsch, J. P. Soininen, M. Forsell, M. Millberg, et al., A network on chip architecture and design methodology, in *Proc. IEEE Computer Society Annual Symposium on VLSI*, 2002, pp. 105–112.
[17] P. Bhojwani and R. Mahapatra, Interfacing cores with on-chip packet-switched networks, in *Proc. 16th Intl Conference on VLSI Design*, 2003, pp. 382–387.
[18] P. Bhojwani and R. Mahapatra, Core network interface architecture and latency constrained on-chip communication, in *Proc. Intl. Symposium on Quality Electronic Devices (ISQED)*, San Jose, 2006, pp. 358–363.
[19] D. J. C. Mackay, Good error-correcting codes based on very sparse matrices, *IEEE Trans. Inf. Theory,* vol., 1999, pp. 399–431.
[20] T. T. Ye, G. De Micheli, and L. Benini, Analysis of power consumption on switch fabrics in network routers, in *Proc. Design Automation Conference (DAC)*, 2002, pp. 524–529.
[21] R. Ho, K. Mai, and M. Horowitz, Efficient on-chip global interconnects, in *Proc. Symposium on VLSI Circuits*, 2003, pp. 271–274.

14

Alleviating Thermal Constraints while Maintaining Performance via Silicon-Based On-Chip Optical Interconnects

Nicholas Nelson, Gregory Briggs, Mikhail Haurylau, Guoqing Chen, Hui Chen, Eby G. Friedman, and Philippe M. Fauchet
University of Rochester
David H. Albonesi
Cornell University

CONTENTS

14.1 Introduction

Growing transistor densities, less than ideal scaling of global wires, and increasing clock frequencies have led to excessive interconnect wire delay and significant heat dissipation in general-purpose microprocessors. The industry

move to multicore chips creates the quandary of how to balance the need for high-speed, high-bandwidth communication and reasonable power density levels. These two criteria are often at odds as the former calls for functionality to be tightly packed, and the latter requires separation. This chapter demonstrates that silicon-based on-chip optical interconnect technology is a promising solution to this growing problem.

In addition to interconnect delay, delay uncertainty has grown significantly. Greater delay uncertainty necessitates the introduction of registers along long distance lines, reducing the amount of useful work that can be accomplished within a clock cycle. Delay uncertainty is further increased by local and global temperature swings.

Increased power dissipation is a critical concern in microprocessors. The heat generated by localized high-power dissipation leads to on-chip hot spots, producing potentially unstable circuit operation and local electromigration concerns. A solution to the problem of hot spots is to physically separate the high-power density components [13]. This strategy, however, exacerbates the problem of long lines and delay uncertainty. The temperature of a block is dependent on the amount of power dissipated in that block, and the temperature of the surrounding blocks. Highly active blocks interspersed with blocks containing low activity will reduce the maximum temperature, although the overall power dissipation will remain the same.

This separation of microprocessor functions to alleviate thermal constraints has the undesirable effect of longer cycle times or deeper pipelines. A *clustered processor microarchitecture* separates processing units into *clusters*, with a dedicated interconnection network used for intercluster communication. Steering algorithms are used to limit intercore forwarding, thereby limiting the increase in delay. A possible solution to long interconnect delay in such *distributed microarchitectures* is the use of transmission-line connections [6]. Although transmission-line connections provide fast communication, these structures are highly bandwidth limited. Wide thick lines also consume a significant amount of the upper metal layer area, limiting the number of possible connections.

Optical interconnects have previously been suggested as a potential solution to the global wire delay problem [25]. Traditionally, the use of on-chip optical interconnections requires the integration of new materials, a prohibitively costly change, or bonding the optical components to a silicon CMOS circuit, also an expensive option. Accordingly, it was believed that optical interconnections are inappropriate for intrachip communication [24]. Recent advances in silicon-based optical devices have solved many of the issues associated with CMOS-based optical devices. These proposed devices are constructed using traditional CMOS processing and materials, and significant progress has been made in electrical/optical conversion [8]. By 2010, for a 1 cm on-chip interconnect length, the propagation delay of an optical link is expected to be half that of an optimal electrical link with repeaters [9].

Although on-chip optical interconnects have recently been evaluated from device and circuit-level perspectives, similar work has yet to be performed at

the architectural level. Thus, it is unclear from a systems perspective whether the use of optical interconnects to replace global on-chip wires is an attractive solution. In this chapter, silicon-based optics for on-chip interconnects are investigated for a large-scale Clustered Multi-Threaded (CMT) processor microarchitecture [14]. Projections for optical and electrical interconnects for 45 nm CMOS are presented based on prior work [6,8,9,16]. One potential benefit of optical interconnects is explored. Specifically, the processing elements are separated and interleaved with L2 cache banks to alleviate heat constraints, and low-latency optical connections from the centralized front end to these back-end elements prevent undue performance loss. The resulting architecture exhibits a significant reduction in heat dissipation (translating into an increase in clock speed and improved reliability) for the same total power level with higher IPC. Although these results are obtained for a large-scale CMT organization, similar benefits can be achieved in a chip multiprocessor microarchitecture.

14.2 Optical System

The successful introduction of optical interconnects onto a microprocessor requires overcoming a number of barriers, the most significant being compatibility with a monolithic (silicon) microelectronic device technology. Due to the poor light-emitting properties of crystalline silicon, the most viable option is to use an external light source (VCSEL laser, etc.) for optical signal generation. An external light source allows more compact and energy-efficient electrooptical modulators as optical information transmitters. Furthermore, low-refractive index polymer waveguides for light propagation and SiGe detectors as receivers are potentially satisfactory candidates.

14.2.1 Modulator

An important example of an ultrafast silicon-based modulator has been demonstrated by Liu et al. [23]. The authors herein indicate that the physical device structure (without considering the driver delay) can operate at speeds in excess of 8 GHz. Moreover, Liu et al. mention that by thinning the gate oxide and using an epitaxial overgrowth technique, it is possible to enhance the phase modulation efficiency. Through additional device geometric optimization, it is also possible to increase the optical mode/active medium interaction volume. Thus, it is reasonable to assume that with technology improvements, the modulator speed will operate in the 30–40 GHz range by 2015. However, because the chosen device structure is a Mach–Zehnder interferometer, this type of modulator has a large footprint, resulting in excessive power consumption and increased driver delay. Simulations and initial experiments performed by Barrios et al. [2,3] show that an alternative

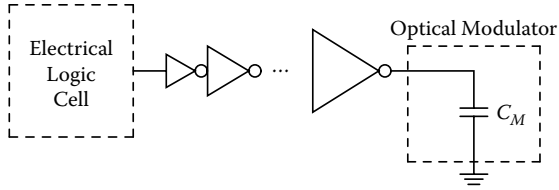


FIGURE 14.1
Circuit model of an optical transmitter.

modulator topology—an optical microcavity—can drastically decrease the modulator area to 10–30 μm while maintaining the same operating speed. Based on these considerations, the capacitance of the modulator structure is estimated to be 1.36 pF.

A block diagram of a driver circuit is shown in Figure 14.1. The microcavity-based optical modulator is assumed to be a purely capacitive load. A series of tapered inverters is used to drive the capacitor [11].

14.2.2 Receiver

The role of an optical receiver is to convert an optical signal into an electrical signal, thereby recovering the data transmitted through the lightwave system. The optical receiver has two primary components: a photodetector that converts light into electricity, and receiver circuits that amplify and digitize the electrical signal. A simplified equivalent circuit model is shown in Figure 14.2. In the context of on-chip optical interconnects, only those technologies that are fully compatible with silicon microelectronics are considered. A practical solution is a SiGe photodetector operating at a 1.3 μm wavelength.

Many types of photodetectors exist due to the many different device structures and operating principles. Interdigitated SiGe p-i-n photodiodes and SiGe

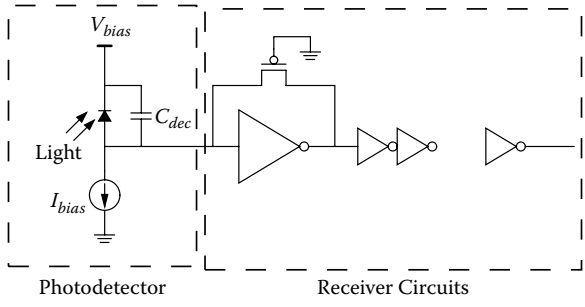


FIGURE 14.2
Circuit model of an optical receiver.

TABLE 14.1

Delay (ps) in a 1 cm Optical Data Path as Compared with the Electrical Interconnect Delay [9]

Modulator driver	25.8
Modulator	30.4
Waveguide	46.7
Photo-detector	0.3
Receiver amplifier	10.4
Total optical	113.6
Electrical	200.0

Metal-Semiconductor-Metal (MSM) detectors are considered here because these detectors tend to respond faster with the same quantum efficiency. In 2002, an interdigitated SiGe p-i-n detector fabricated on a Si substrate with a 3 dB bandwidth of 3.8 GHz at a 1.3 μm wavelength was demonstrated [26].

A summary of the delays of the individual elements along the optical data path is listed in Table 14.1. Note the significant delay advantage over optimal electrical interconnects with repeaters for a target length of 1 cm. More details describing the device/circuit aspects of the optical technology can be found in [6,8,9,16].

14.3 Architectural Design

The baseline processor is a clustered multi-threaded (CMT) machine [14] with a unified front-end, and 16 cores containing functional units, register files, and data caches for a back end, as shown in Figure 14.3. The simulator is based on Simplescalar-3.0 [5] for the Alpha AXP instruction set with the Wattch [4] and HotSpot [18] extensions. Processor parameters are listed in Table 14.2.

14.3.1 Core Layout

A floorplan of the processing core (back end) is illustrated in Figure 14.4. Each back end is linearly scaled from the Alpha 21264 floorplan [19] to the 2010 (45 nm) technology node. Units whose parameters differ from the 21264 (i.e., there are 64 integer registers rather than 80) are also linearly scaled.

The layout of the processor requires that each core has a level one data cache. The cache is assumed to use a simplified coherence scheme. The mesh interconnect network is inherently unordered, and the delay from one point to another point is nonuniform. The cache coherence actions are performed in the order seen by the simulator. The level two data cache is a nonuniform

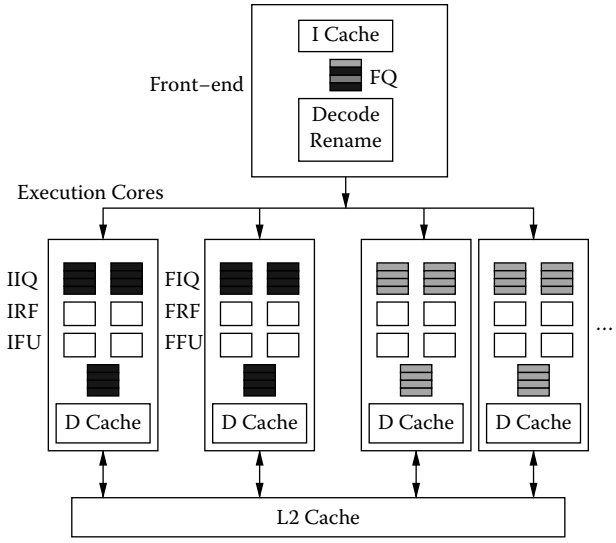


FIGURE 14.3
Clustered multithreaded architecture with two cores per thread.

TABLE 14.2

Processor Parameters

Cluster	
L1 data cache	16 KB per core 2 way, 2 cycles
Load/store queue	64 entries
Register file	64 Int, 64 FP
Issue queue	64 Int, 64 FP
Integer units	2 ALU, 1 Mult
Floating point	1 ALU, 1 Mult
Front end	
Combined branch predictor	2048 entry BTB
Return address stack	32 entries
Branch mispredict penalty	12
Fetch queue size	64 shared
Fetch width	32 instructions from 2 threads
Dispatch	16 shared
Commit	12 per thread
Reorder buffer	256 per thread
L1 instruction cache	32 kB 2 way
Unified L2 cache	64 MB 32 way
TLB (each, I and D)	128 entries, 8KB fully associative per thread
Memory latency	200 cycles

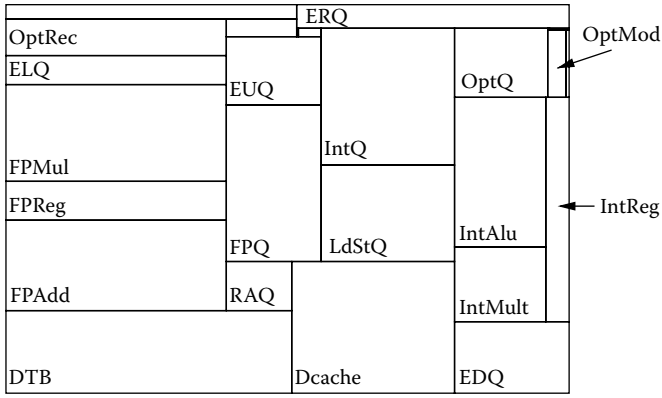


FIGURE 14.4
Core floorplan.

access time structure; for simplicity, however, it is simulated as a uniform access time structure. This approximation is accurate if the cache allows frequently accessed blocks to be moved closer to the utilizing cores [12].

14.3.2 Processor Layout

Two layout strategies are compared to demonstrate the advantages of on-chip optical interconnects. The grid floorplan, as shown in Figure 14.5, is

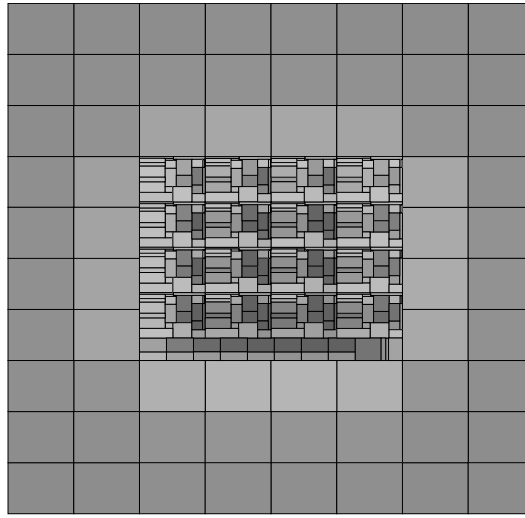


FIGURE 14.5
Grid floorplan. The back-end cores are in the center, above the common front end, completely surrounded by a 64 MB unified L2 cache.

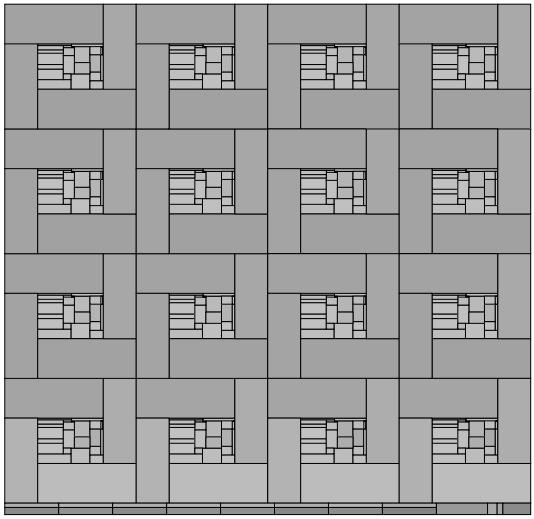


FIGURE 14.6
Checkers floorplan. Each core is surrounded by four unified L2 cache banks. The front end is along the bottom edge of the layout.

the baseline configuration, in which the cores are closely packed to minimize intercluster delay. This floorplan consists of 16 replicated cores surrounded by 64 banks of a unified level 2 cache. The second floorplan, shown in Figure 14.6, is proposed to reduce the maximum temperature while maintaining IPC performance. This floorplan has the advantage of spreading out the hot cores, thereby allowing the cool cache to reduce the temperature. Each of the 16 cores is surrounded by four banks of a unified level 2 cache.

A mesh Manhattan interconnection scheme is simulated; each core can communicate via electrical links with neighbors at a cost of one cycle. Communication between distant cores requires multiple hops, and congestion is considered. All of the electrical links are capable of serving two 64 bit values (two registers) per cycle for each layout configuration. The shared front-end is located along the bottom of the core elements. In this study, optical links are only used for direct communication between the front-end (shown at the bottom) and each core. Communication over these optical links requires two cycles, compared to a worst case of seven cycles for wire interconnects.

14.4 Methodology

In this analysis, the maximum transient temperature of any functional unit limits the clock frequency. The maximum temperature is determined by executing the workload on a checkers layout (see Figure 14.6) without an optical

front-end communication network for a mix of benchmarks. To obtain the frequency for a grid layout (see Figure 14.5) with the same maximum temperature, three different clock frequencies are simulated and interpolated. (In the region of interest, the temperature is approximately linear with the clock frequency.)

To measure the effect of the impact on performance by spreading out the processing cores, the IPC performance of a microarchitecture with optical links between the front end and back end is compared with a system with only electrical interconnects. In future work, the use of optical interconnects to reduce long distance inter-back-end communication latencies will also be investigated.

14.4.1 Power Model

Wattch version 1.02 [4] is used to compute the dynamic power of the units. Parameters for the 45 nm technology node are derived from the ITRS [30]. The wire resistance and capacitance scaling factors are determined by log–log extrapolation from the technology nodes supplied with Wattch. Similarly, the sense voltage factor is determined by linear extrapolation from earlier technology nodes.

A simple temperature-dependent computation of leakage power is applied. Gate oxide leakage is assumed to not be significant (as a result of the adoption of a high-*k* dielectric technology) [20]. Therefore, only subthreshold leakage is considered. The units are divided into logic and SRAM groups, due to differences in ITRS predictions [28] for these two groups. The power is determined from the ITRS-predicted transistor density, static power per transistor width, and several additional assumptions: an average W/L of 3 for the SRAM circuitry and 3.6 for the logic, each PMOS transistor leaks twice as much as an NMOS transistor, and the NMOS and PMOS transistors are each on 50% of the time. The ITRS value for leakage power at room temperature provides a reference, and the BSIM3 model [35] is used to correlate leakage power with temperature. Equation (14.1) is used to adjust the leakage power of each unit based on the temperature of that individual unit, continually recalculated as the temperature changes.

$$P = \frac{P_{static}(W/L)L_{gate}Q_{density} * area}{T_{ITRS}^2} T^2 \ Watts \tag{14.1}$$

where

$$P_{static} = \frac{\tau_{N,leak}P_{N,static} + 2\tau_{p,leak}P_{N,static}}{2} \tag{14.2}$$

Equation (14.2) is given in terms of watts per meter of the transistor gate width with $\tau_{leak,N}$ and $\tau_{leak,P}$ referring to the fraction of the time that the *N* and *P*

TABLE 14.3
HotSpot Parameters

Heat Sink	
Convection resistance	0.02 K/W
Convection capacitance	140.4 J/K
Thermal Interface Material	
Thickness	30 μm
Thermal resistivity	0.14 mK/W

transistors, respectively, dissipate leakage (rather than dynamic) power. L_{gate} is the printed length of the gate, $Q_{density}$ is the density of transistors, and *area* is the actual die area of the device. T refers to the absolute temperature of the unit and is a function of time.

14.4.2 Temperature Model

Chip temperatures are derived from the power numbers using the HotSpot (version 2) [18] simulation tool. HotSpot determines the transient temperatures, so maximum transient temperatures are used. (Steady-state temperatures are not used because potential short-period hot spots are ignored.)

The HotSpot parameters are listed in Table 14.3. High-end cooling technologies are assumed, because cooling will be more important in future processors. For the heat sink, the resistance of a "folded-fin" heat sink is used [22], as well as a thermal interface material with a resistivity of 0.14 mK/W [1] and a thickness of 30 μm. This thickness is about half of the coverage thickness used as a default in HotSpot or assumed by the Arctic Silver specifications [1]. Because the thermal interface material may play an important role in dissipating heat from the hot spots, it is assumed that by 2010 the thickness will be reduced from the current 70 μm. Parameters not explicitly listed are the same as the default values specified in the HotSpot software.

14.4.3 Benchmarks

Two classes of workloads are considered, mixes of SPEC2000 CPU benchmarks (groupA) and SPLASH-2 benchmarks operating in multithreaded mode (groupB). Using the same classification system as [14], two communication bound workloads and an instruction-level parallelism (ILP) bound workload are examined. The mixes are listed in Table 14.4.

TABLE 14.4

Single-Threaded Mixes

Load	Benchmarks Included	Bound
Mix 1	bzip, parser, art, galgel	Communication
Mix 2	bzip, vpr, gzip, parser, perlbmk, lucas, art, galgel	Communication
Mix 3	gcc, mcf, twolf, applu, mgrid, swim, equake, mesa	ILP

GroupA benchmarks are mixes of independent threads. These benchmarks do not share virtual memory address space and therefore there is no interthread communication. Each SPEC benchmark in this group is run with the reference input set. The benchmarks are individually fast forwarded as suggested in [29], and run simultaneously until each thread reaches 100 million instructions. The geometric mean of the speedup of all of the threads is used as the performance metric.

GroupB benchmarks are parallel programs from the SPLASH-2 benchmark suite [36]. The relevant parameters are listed in Table 14.5. The threads share virtual address space and communicate with one another by means of shared memory facilitated by cache coherence. Each benchmark in groupB is run to completion. Speedup is calculated as the ratio of the execution times in cycles.

Each individual thread has exclusive access to two adjacent cores. Prior research has shown that the communication delays involved with additional cores negate any performance gain from the increase in the number of functional units [14,21].

TABLE 14.5

Parallel Programs

Program	Command Line Arguments
FFT	-m18 -p8 -n1024 -16 -t
Jacobi	-p8 -v -s512 -i10
LU	-n512 -p8 -b16 -t
Radix	-p8 -n131072 -r16 -m524288 -t

14.5 Results

The results are relative to a benchmark run with a grid layout (see Figure 14.5) with no optical communication lines. Mixes of independent threads are first presented followed by parallel programs.

14.5.1 GroupA

The left bars in each group shown in Figure 14.7 quantify the change in the clock frequency (and therefore the performance) achieved by using the spread-out checkers layout (Figure 14.6). The middle bars include the optical communication lines from the shared front end to each of the cores. The direct communication lines allow for faster dispatch of instructions to the cores and a shorter branch mispredict penalty (the recovery is started earlier). This modest application of optical interconnect leads to an increase in performance of up to 10% for multithreaded workloads of independent applications.

The right bars combine the two techniques. The average speedup for these benchmark mixes is 35% with a maximum of 38%. The two enhancements are not completely orthogonal. The faster communication with the front end leads to enhanced utilization of the functional units, which in turn increases the baseline temperature. The increase in clock speed is therefore partly reduced.

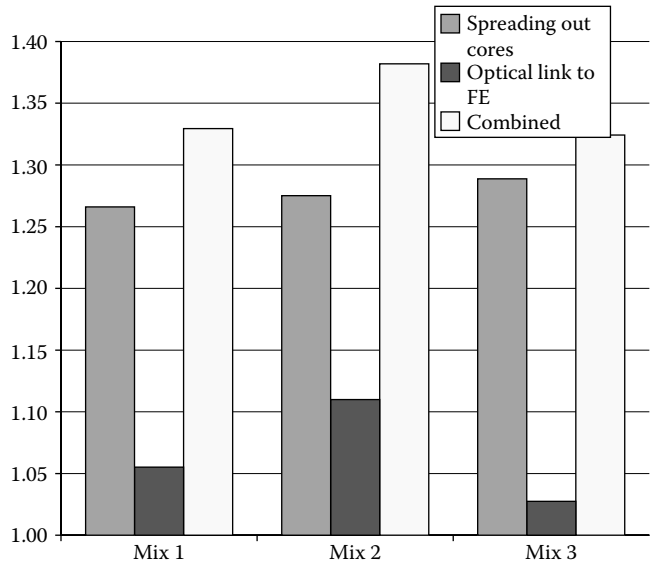


FIGURE 14.7
Speedup resulting for GroupA.

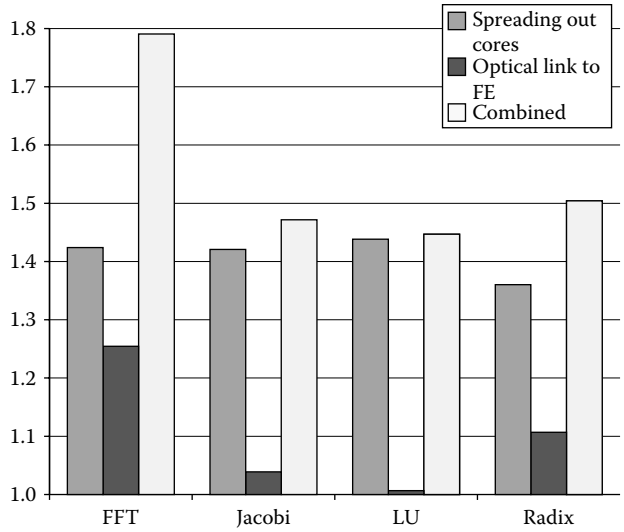


FIGURE 14.8
Speedup resulting for GroupB.

14.5.2 GroupB

The multithreaded benchmarks produce greater improvements. The left bars shown in Figure 14.8 describe improvements from spreading out the cores. The speedup is roughly 40% across all of the benchmarks.

The middle bars are obtained by adding the high-speed optical links from the front end to each core. The improvement varies depending on the nature of each benchmark, but reaches 25% for FFT.

The right bars present the results of combining the two techniques. The average speedup for these benchmarks is 55% with a maximum of 78%.

14.6 Related Work

Modeling the effects of leakage current on power dissipation and temperature at the architectural level was first studied by Sohi and Butts [32] and later by Zhang et al. [37], the former based on the BSIM3 transistor leakage model [35].

Others have investigated dynamic temperature management schemes, such as frequency, voltage, and fetch rate control [31], software scheduling behavior [28], asymmetric dual core designs [15], and a combination of these techniques [17].

Additional researchers have also considered the impact of circuit layout on temperature, such as Cheng and Kang with their iTAS simulator [10]. Investigations have also been promoted by other VLSI-based simulation research, such as Rencz et al. [27], the SISSI package [33], and others [34].

Donald and Martonosi investigated thermal issues in SMT and CMP architectures [13], although these authors only consider steady-state temperatures and do not translate the temperature results into the effect on application performance.

In contrast to these previous research results, this work is the first to investigate the use of on-chip optical interconnects to reduce the performance gap created by increasing the physical distances between the front and back ends of the processor in order to alleviate thermal constraints.

14.7 Conclusions

With recent advances in silicon photonics, on-chip optical interconnects have become a prime candidate to alleviate a number of global communication challenges in future highly integrated microprocessors. In this chapter, the use of optical interconnects to ameliorate the increased global wire delay due to intermixing hot and cold processing units is investigated. It is shown that the selective introduction of a few optical connections can significantly enhance overall processor performance. This study has also shown that intermingling the cluster cores with the on-chip cache reduces the maximum on-chip temperature. Because the maximum temperature limits the clock speed, spreading the cores can lead to increased clock frequencies. This technique does not reduce overall power dissipation (other than the decreased leakage current due to lower on-chip temperatures) but more uniformly redistributes the dissipated power. The use of optical interconnect for long distance communication makes spreading the cores a more viable proposition in terms of maintaining high-performance levels.

In future work, the use of optical interconnect will be investigated to reduce inter-back-end communication for parallel workloads, increase link bandwidth through the use of Wave Division Multiplexing (WDM), and reduce the worst case latencies of large cache and main memory RAMs.

Acknowledgments

This research was supported by National Science Foundation grant CCR-0304574.

References

[1] Arctic Silver Incorporated. The Arctic Silver 5 Specifications. http://www.arcticsilver.com/as5.htm, 2004.

[2] C. A. Barrios, V. R. d. Almeida, and M. Lipson. Electrooptic modulation of silicon-on-insulator submicrometer-size waveguide devices. *Journal of Lightwave Technology*, 21(10):2332, Oct. 2003.

[3] C. A. Barrios, V. R. d. Almeida, and M. Lipson. Compact silicon tunable Fabry-Perot resonator with low power consumption. *IEEE Photonics Technology Letters*, 16(2):506, Feb. 2004.

[4] D. Brooks, M. Martonosi, and V. Tiwari. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 83–94, June 2000.

[5] D. Burger and T. Austin. The simplescalar toolset, version 2.0. Technical Report TR-97–1342, University of Wisconsin-Madison, June 1997.

[6] R. T. Chang, N. Talwalkar, C. P. Yue, and S. S. Wong. Near speed-of-light signaling over on-chip electrical interconnects. *IEEE Journal of Solid-State Circuits*, 38(5):834–838, May 2003.

[7] G. Chen, H. Chen, M. Haurylau, N.A. Nelson, D. H. Albonesi, P. M. Fauchet, and E. G. Friedman, "Predictions of CMOS Compatible On-Chip Optical Interconnect," *Integration, the VLSI Journal*, Volume 40, Issue 4, pp. 434–446, July 2007.

[8] G. Chen, H. Chen, M. Haurylau, N. Nelson, D. H. Albonesi, P. M. Fauchet, and E. G. Friedman. Electrical and optical on-chip interconnects in future microprocessors. In *IEEE International Symposium on Circuits and Systems*, May 2005.

[9] G. Chen, H. Chen, M. Haurylau, N. Nelson, P. M. Fauchet, E. G. Friedman, and D. H. Albonesi. Predictions of CMOS compatible on-chip optical interconnect. In *Proceedings of the IEEE/ACM International Workshop on System Level Interconnect Prediction*, Apr. 2005.

[10] Y.-K. Cheng and S.-M. Kang. A temperature-aware simulation environment for reliable ULSI chip design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(10):1211–1220, Oct. 2000.

[11] B. S. Cherkauer and E. G. Friedman. A unified design methodology for CMOS tapered buffers. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 3(1):99–111, Mar. 1995.

[12] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 55–66, Dec. 2003.

[13] J. Donald and M. Martonosi. Temperature-aware design issues for SMT and CMP architectures. In *Fifth Workshop on Complexity-Effective Design*, 2004 June.

[14] A. El-Moursy, R. Garg, S. Dwarkadas, and D. H. Albonesi. Partitioning multi-threaded processors with large number of threads. In *IEEE International Symposium on Performance Analysis of Systems and Software*, Austin, Texas, Mar. 2005.

[15] S. Ghiasi and D. Grunwald. Thermal management with asymmetric dual core designs. Technical Report CU-CS-965-03, Department of Computer Science, University of Colorado, 2003.

[16] M. Haurylau, G. Chen, H. Chen, J. Zhang, N. A. Nelson, D. H. Albonesi, E. G. Friedman, and P. M. Fauchet, "On-Chip Optical Interconnect Roadmap: Challenges and Critical Directions," IEEE Journal of Selected Topics in Quantum Electronics, Special Issue on Silicon Photonics, Vol. 12, No. 6, pp. 1699–1705, November/December 2006.

[17] M. Huang, J. Renau, S. Yoo, and J. Torrellas. The Design of DEETM: A framework for dynamic energy efficiency and temperature management. *Journal of Instruction-Level Parallelism*, 3, Oct. 2001.

[18] W. Huang, M. R. Stan, K. Skadron, K. Sankaranarayanan, S. Ghosh, and S. Velusamy. Compact thermal modeling for temperature-aware design. In *Proceedings of the 41st IEEE/ACM Design Automation Conference*, June 2004.

[19] R. E. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, pages 24–36, Mar./Apr. 1999.

[20] N. S. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. S. Hu, M. J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12):68–75, Dec. 2003.

[21] F. Latorre, J. González, and A. González. Back-end assignment schemes for clustered multithreaded processors. In *Proceedings of the 18th Annual ACM International Conference on Supercomputing*, pages 316–325, June 2004.

[22] S. Lee. How to select a heat sink. *Electronics Cooling*, 1(1), June 1995.

[23] A. Liu, R. Jones, L. Liao, D. Samara-Rubio, D. Rubin, O. Cohen, R. Nicolaescu, and M. Paniccia. A high-speed silicon optical modulator based on a metal-oxide-semiconductor capacitor. *Nature*, 427:615–618, Feb. 2004.

[24] R. Lytel, H. L. Davidson, N. Nettleton, and T. Sze. Optical interconnections within modern high-performance computing systems. *Proceedings of the IEEE*, 88(6):758–763, June 2000.

[25] D. A. B. Miller. Rationale and challenges for optical interconnects to electronic chips. *Proceedings of the IEEE*, 88(6):728–749, June 2000.

[26] J. Oh, J. Campbell, S. G. Thomas, S. Bharatan, R. Thoma, C. Jasper, R. E. Jones, and T. E. Zirkle. Interdigitated Ge p-i-n photodetectors fabricated on a Si substrate using graded SiGe buffer layers. *IEEE Journal of Quantum Electronics*, 38(9):1238–1241, Sept. 2002.

[27] M. Rencz, V. Szekely, A. Poppe, and B. Courtois. Friendly tools for the thermal simulation of power packages. *International Workshop on Integrated Power Packaging, 2000*, pages 51–54.

[28] E. Rohou and M. D. Smith. Dynamically managing processor temperature and power. In *Proceedings of the 2nd Workshop on Feedback-Directed Optimization*, Nov. 1999.

[29] S. Sair and M. Charney. Memory behavior of the SPEC2000 benchmark suite. Technical report, IBM T. J. Watson Research Center, Oct. 2000.

[30] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors*. 2003.

[31] K. Skadron, M. R. Stan, K. Sankaranarayanan, W. Huang, S. Velusamy, and D. Tarjan. Temperature-aware microarchitecture: Modeling and implementation. *ACM Transactions on Architecture and Code Optimization*, 1(1):94–125, Mar. 2004.

[32] G. S. Sohi and J. A. Butts. A static power model for architects. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pages 191–201, Dec. 2000.

[33] V. Szekely, A. Poppe, A. Pahi, A. Csendes, G. Hajas, and M. Rencz. Electro-thermal and logi-thermal simulation of VLSI designs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 5(3):258–269, Sept. 1997.

[34] K. Torki and F. Ciontu. IC thermal map from digital and thermal simulations. In *Proceedings of the 2002 International Workshop in THERMal Investigations of ICs and Systems*, pages 303–308, Oct. 2002.

[35] University of California, Berkeley. *BSIM3v3.2.2 Manual*, 1999.

[36] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[37] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. R. Stan. Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects. Technical Report CS-2003-05, Department of Computer Science, University of Virginia, Mar. 2003.

Index

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

COMPUTER ENGINEERING

Unique Chips and Systems

While integrated circuits enable technology for the modern information age, computing, communication, and network chips fuel it. As soon as the integration ability of modern semiconductor technology presents opportunities, issues in power consumption, reliability, and form-factor present challenges. The demands of emerging software applications can only be met with unique systems and chips. Drawing on contributors from academia, research, and industry, ***Unique Chips and Systems*** explores specific approaches to designing future computing and communication chips and systems.

The book focuses on specialized hardware and systems as opposed to general-purpose chips and systems. It covers early conception and simulation, mid-development, application, testing, and performance. The chapter authors introduce new ideas and innovations in unique aspects of chips and system design, then go on to provide in-depth analysis of these ideas. They explore ways in which these chips and systems may be used in further designs or products, spurring innovations beyond the intended scopes of those presented. International in flavor, the book brings industrial and academic perspectives into focus by presenting the full spectrum of applications of chips and systems.

51741

ISBN 1-4200-5174-1

90000

9 781420 051742

CRC Press
Taylor & Francis Group
an **informa** business
www.taylorandfrancisgroup.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
270 Madison Avenue
New York, NY 10016
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

www.crcpress.com