Cycle-Accurate Modeling of Multicore Processors on FPGAs

by

Asif Imtiaz Khan

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

ARCHIVES
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
OCT 0 2 2013
LIBRARIES

© Massachusetts Institute of Technology 2013. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
June 28, 2013

Certified by .....
Arvind
Johnson Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chair of the Committee on Graduate Students

Cycle-Accurate Modeling of Multicore Processors on FPGAs

by

Asif Imtiaz Khan

Submitted to the Department of Electrical Engineering and Computer Science
on June 28, 2013, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

We present a novel modeling methodology which enables the generation of a high-performance, cycle-accurate simulator from a cycle-level specification of the target design. We describe Arete, a full-system multicore processor simulator, developed using our modeling methodology. We provide details on Arete's resource-efficient and high-performance implementation on multiple FPGA platforms, and the architectural experiments performed using it.

We present clear evidence that the use of simplified models in architectural studies can lead to wrong conclusions. Through two experiments performed using both cycle-accurate and simplified models, we show that on one hand there are substantial quantitative and qualitative differences in results, and on the other, the results match quite well.

Thesis Supervisor: Arvind
Title: Johnson Professor of Electrical Engineering and Computer Science

Acknowledgments

It's been a long and eventful journey, and I've met some incredible people along the way. Arvind, my research advisor, has been brilliantly insightful the last eight years that I've known him and worked with him. All my friends and colleagues in CSG have been instrumental in shaping my thinking, and I have thoroughly enjoyed our numerous discussions and collaborations. My friends, from both MIT and the Boston area, have been a wonderful source of boisterous fun, and I will always cherish the time I spent in their company.

I would like to thank Abida Aunty and Zafar Uncle, and Prof. Terry Orlando and his wife, Ann, for seeing me through some very tough times. I would never have come this far, had it not been for their kindness and generosity.

I am deeply indebted to my family who have been a great source of strength throughout this endeavor. And I am thankful to my wife, Sana, for her love and support. Our time together has brought me tremendous joy and fulfillment.

To Ammi and Baba, my parents.

Contents

1 Introduction **19**

1.1 The case for FPGA-based modeling 21

1.2 The case for cycle-accurate modeling 22

1.3 Summary of contributions 23

1.4 Document outline 25

I FPGA-based Modeling **27**

2 Functional and Timing Specifications for a Cycle-Accurate Model **29**

2.1 Introduction 29

2.2 Cycle-accurate specifications 30

2.2.1 Timed RTL (T-RTL) 31

2.2.2 Timing specifications for a processor 32

2.2.3 Target simplification vs. implementation refinement 37

2.3 Implementation refinements 38

2.4 The LI-BDN technique for writing cycle-accurate simulators 39

2.5 Summary 44

3 Fast and Cycle-Accurate Modeling of a Multicore Processor **45**

3.1 Introduction 45

3.2 Processor architecture 46

3.2.1 Core 47

3.2.2 Shared memory and cache coherence 49

3.2.3 On-chip network 53
3.2.4 Message-passing support 54
3.3 Full-system processor simulator 54
3.3.1 Simulation infrastructure 55
3.3.2 Portability across FPGA platforms 56
3.3.3 Flexibility for architectural experiments 57
3.3.4 Synthesis statistics 58
3.3.5 Performance evaluation 60
3.4 Related work 62
3.4.1 Software-based multicore simulators 62
3.4.2 FPGA-based processor simulations 63
3.5 Summary 68

4 Deterministic, Model-Cycle-Level Debugging of Synchronous Systems Modeled Asynchronously **71**

4.1 Introduction 71
4.2 Survey of debugging techniques for FPGA-based designs 73
4.2.1 System monitoring through scan chains 73
4.2.2 SCE-MI-based emulation environment 74
4.2.3 ISA-based debugging 75
4.2.4 Debugging in various asynchronous FPGA-based models . . . 75
4.3 Debugging using the LI-BDN technique 76
4.3.1 Correctness of the LI-BDN-based debugging technique 80
4.3.2 Deterministic execution 81
4.4 LI-BDN-based debugging infrastructure for a multicore processor model: A case study 85
4.5 Summary 88

II Architectural Exploration Using Cycle-Accurate Simulation **89**

5 Impact of Modeling Abstractions on the Accuracy of Single-Core Processor Simulations **91**

5.1 Introduction 91

5.2 Related work 93

5.3 Comparison of branch predictors using cycle-accurate and abstract models 96

5.3.1 Model with memory abstraction (AbsM) 97

5.3.2 Model with memory and execution abstractions (AbsME) . . . 99

5.3.3 Sampled execution of benchmarks 105

5.4 Summary 105

6 Impact of Simplified Core Models on the Accuracy of Multicore Processor Simulations **109**

6.1 Introduction 109

6.2 An experiment in the memory subsystem 111

6.2.1 Experimental setup 111

6.3 Memory experiment using the cycle-accurate core model 113

6.4 Memory experiment using the 1IPC core model 114

6.4.1 Explaining the differences in results 116

6.5 Improving the accuracy of 1IPC 120

6.5.1 Lowering the execution rate 120

6.5.2 Adding speculative instructions 123

6.5.3 Adding the full speculative path 126

6.6 Additional comments 132

6.6.1 Error scaling 132

6.6.2 Variability study 132

6.6.3 Comparison against real machines 133

6.7 Related work 136

6.8 Summary 139

7 Data Movement Control: An Architectural Experiment 141

7.1 Introduction 141

7.2 Related work 144

7.2.1 Multicore cache management 144

7.2.2 Computation migration 144

7.3 DMC hardware interface 145

7.3.1 `cpush` 145

7.3.2 `clookup` 146

7.3.3 `cmsg` 147

7.4 DMC hardware design 149

7.4.1 Correctness of `cpush` 149

7.4.2 Performance of `cmsg` 152

7.5 Implementation 152

7.5.1 Hardware 153

7.5.2 Software 153

7.5.3 Testing 153

7.6 Evaluation 154

7.6.1 Cost of `cmsg` 155

7.6.2 Thread migration 156

7.6.3 Linked lists 157

7.7 Summary 159

7.7.1 Limitations 159

8 Conclusion 161

8.1 Processor modeling on FPGAs 161

8.2 The need for cycle-accurate modeling 162

8.3 Future work 162

8.3.1 Power modeling 162

8.3.2 Combining moderate-scale cycle-accurate simulations with large-scale functional simulations 165
8.3.3 Hardware/software codesign 166

List of Figures

1-1 Performance impact of the LRU replacement policy determined using both the cycle-accurate and the 1-IPC core models. Baseline replacement policy is random. 24

2-1 Approaches to FPGA-based modeling 31
2-2 A synchronous sequential machine (SSM) 32
2-3 Specification of a shared TLB 34
2-4 Specification of a completion table 35
2-5 A refined SSM 38
2-6 Synchronous specification of a 2-read, 1-write register file module . . 40
2-7 Transforming a cycle-level specification into an LI-BDN module . . . 41
2-8 Refined LI-BDN register file module 42
2-9 Comparison of resource and timing statistics for SSM and LI-BDN implementations of a register file with 32×64-bit entries, 2 read ports and 1 write on the XUPv5 board 43
2-10 Modeling methodology 44

3-1 Architecture of a processor tile 46
3-2 Architecture of an in-order PowerPC core 47
3-3 Shared memory architecture 50
3-4 Cache state transitions 51
3-5 Home directory state transitions 52
3-6 Fully connected network topology in Arete 53
3-7 Various types of traffic supported by the on-chip network 54

3-8 Simulation infrastructure 55

3-9 A complete view of the FPGA implementation of Arete 56

3-10 Supported FPGA boards 57

3-11 Comparison of the prototype and the refined LI-BDN implementations of PowerPC on the XUPv5 board. Model parameters: 1 tile, 1 in-order 10-stage core, 64 KB 4-way associative L1 caches, 512 KB 4-way associative L2 cache, 512 MB DRAM 58

3-12 Resource utilization for the refined LI-BDN implementation of the PowerPC model and peripherals on the BEE3 board. Model parameters: 1 tile, 2 in-order 10-stage cores, 64 KB 4-way associative L1 caches, 512 KB 4-way associative L2 cache, 2 GB DRAM 59

3-13 Resource utilization for the refined LI-BDN implementation of the PowerPC model and surrounding peripherals on the XUPv5 platform. Model parameters: 1 tile, 2 in-order 10-stage cores, 64 KB 4-way associative L1 caches, 512 KB 4-way associative L2 cache, 1 GB DRAM . 60

3-14 Resource utilization for the refined LI-BDN implementation of the PowerPC model and peripherals on the ML605 board. Model parameters: 1 tile, 4 in-order 10-stage cores, 64 KB 4-way associative L1 caches, 512 KB 4-way associative L2 cache, 512 MB DRAM 60

3-15 Performance evaluation using the PARSEC benchmark suite running on top of SMP Linux. Model parameters: 4 tiles, 8 in-order 10-stage cores, 64 KB 4-way associative L1 caches, 512 KB 4-way associative L2 cache, 4 GB DRAM 61

4-1 Summary of the comparison between the LI-BDN-based debugging technique and other common debugging techniques used in FPGA-based designs 73

4-2 LI-BDN register file module with support for model-cycle-level debugging 77

4-3 FPGA-optimized LI-BDN register file module with support for debugging 78

4-4 Synchronous specification of a DRAM module with non-deterministic read latency 82
4-5 LI-BDN DRAM module with non-deterministic read latency 83
4-6 LI-BDN DRAM module with combinational reads 84
4-7 A screen shot of the debugging capabilities provided by the debugging software developed for Arete 86
4-8 Arete core with model-cycle-level debugging facilities 87
4-9 Resource and performance penalties of the debugging infrastructure in Arete. Model parameters: 1 tile, 2 in-order 10-stage cores, 64 KB 4-way associative L1, 512 KB 4-way associative L2 88

5-1 Effect of different branch prediction schemes on IPC, obtained from the ACC models. Baseline scheme is ANT. 97
5-2 Effect of different branch prediction schemes on IPC, obtained from the AbsM models. Graph in (a) is from the ACC model, added for ease of comparison. 100
5-3 Error in IPC obtained from the AbsM models with different abstraction parameters, with respect to the corresponding cycle-accurate models . 101
5-4 Effect of different branch prediction schemes on IPC, obtained from the AbsME models. Graph in (a) is from the ACC model, added for ease of comparison. 103
5-5 Error in IPC obtained from the AbsME models with different abstraction parameters, with respect to the corresponding cycle-accurate models 104
5-6 Effect of different branch prediction schemes on IPC obtained from the cycle-accurate models using sampled execution. Graph in (a) is from full execution, added for ease of comparison. 106

6-1 Publications with various core models in full-system simulators used to study the memory hierarchy or the interconnect network 110
6-2 Accurate core model 113

6-3 Impact of LRU, MRU and LNS replacement policies obtained from ACC. Baseline replacement policy is random. 115
6-4 1IPC core model 116
6-5 Impact of LRU, MRU and LNS replacement policies obtained from 1IPC. Baseline replacement policy is random. 117
6-6 Comparison of the 1IPC core model with the ACC core model (baseline) 118
6-7 Impact of LRU, MRU and LNS replacement policies obtained from 1IPC-R. Baseline replacement policy is random. 121
6-8 Comparison of the 1IPC-R core model with the ACC core model (baseline) 122
6-9 7NDH core model 123
6-10 Impact of LRU, MRU and LNS replacement policies obtained from 7NDH. Baseline replacement policy is random. 124
6-11 Comparison of the 7NDH core model with the ACC core model (baseline) 125
6-12 Impact of LRU, MRU and LNS replacement policies obtained from 7NDH-R. Baseline replacement policy is random. 127
6-13 Comparison of the 7NDH-R core model with the ACC core model (baseline) 128
6-14 10NDH core model 129
6-15 Impact of LRU, MRU and LNS replacement policies obtained from 10NDH. Baseline replacement policy is random. 130
6-16 Comparison of the 10NDH core model with the ACC core model (baseline) 131
6-17 Increase in mean error magnitude as the number of cores increases in processor simulations with various coarse-grained core models 132
6-18 Impact of LRU, MRU and LNS replacement policies obtained from ACC. Baseline replacement policy is random. Bars show the average values from sixteen application runs, while marks show the minimum and the maximum values. 134

6-19 Impact of LRU, MRU and LNS replacement policies obtained from 1IPC. Baseline replacement policy is random. Bars show the average values from sixteen application runs, while marks show the minimum and the maximum values. 135
6-20 Comparison of statistics obtained from running the same application on Arete, ARM Cortex-A9 and Core i7-965 136

7-1 Cache state transitions for DMC 150
7-2 Directory state transitions for DMC 151
7-3 Results for the memory scan benchmark. The x-axis shows the number of cache lines in a segment and the y-axis shows the average latency to the read segment from another core's L1 cache. 155
7-4 Results for the thread migration microbenchmark. The x-axis shows the number of cache lines the source core pushes to the destination core using `cpush`. 157
7-5 Results for the list microbenchmark 158

8-1 Power modeling approach 163
8-2 Statistics for register file ports 164
8-3 Combining Arete with large-scale simulators like Graphite 165
8-4 Hardware/software codesign on Arete 167

Chapter 1

Introduction

Performance modeling plays a critical role during the design cycle of a processor. It enables designers to explore and analyze architectural ideas that emerge from their knowledge, experience and intuition. To facilitate architectural exploration, simulators used for performance modeling have to be easily modifiable. To reliably assess the impact of architectural changes on processor performance, these simulators also have to model the processor architecture accurately, and run a representative set of benchmarking applications in a reasonable amount of time.

Most performance modeling is done through simulators written in C/C++. This eases the model development effort and facilitates design-space exploration. The speed of these simulators, however, has always been found lacking. As the complexity of processor designs continues to grow, the challenge of software simulation speed gets tougher to tackle.

This growing problem has been tackled in three different but complimentary ways. In the first approach, a representative subset of benchmarks is selected, based on the kind of simulation study being performed. For example, for a memory study, memory-intensive benchmarks are used. Benchmark selection is then coupled with sampling, which involves executing representative, periodic or random portions of the benchmarks on a detailed performance model with considerably low speed, and the remaining portions on a fast functional model. This approach can skew results if representative benchmarks and samples are not chosen carefully. However, there

have been many advances in this domain [1], and generally there is consensus in the community on how sampling should be done, and when it can be used acceptably. For all the experiments presented in this thesis, we used standard benchmarks, and did not employ any sampling.

The second approach makes use of faster substrates, the most obvious being multicore hosts and clusters/workstations, to simulate large multicore designs. Cycle-accurate simulations in this environment have proven to be much harder than expected. A new emerging trend is to use FPGAs for cycle-accurate simulations, as opposed to, for emulation and validation of RTL. In the last 7-8 years researchers have shown that flexible and cycle-accurate performance models can be built on FPGAs which provide 1000× performance improvement over software. *An important contribution of this thesis is to show a new way of building cycle-accurate FPGA-based performance models starting from a cycle-level specification of the target design, written in a high-level language.*

The third approach to solving the simulation speed problem is to simplify the target machine. For example, one can use a very a simple unpipelined core model when studying a large multicore processor design. The justification being that if one is studying inter-processor communication properties through the memory subsystem and the on-chip network, then perhaps the architecture of the core has minimal impact on the study. The problem is that such hunches are almost never validated. The approach is similar to using mice models for studying a biological phenomenon in human beings. No one would suggest that such a study offers any conclusive insight into human behavior unless the study is repeated on human subjects. *In this thesis we will show through concrete experiments that the use of simplified core models in multicore processor simulators leads to wrong conclusions, both quantitatively and qualitatively.*

The main conclusion of this thesis is that there is no way to get around building cycle-accurate models because, even when simplified models work, we know that only ex post facto, by conducting the same experiments on cycle-accurate models. Simulators with simplified models can save time, but only when used in conjunction

with cycle-accurate models. Furthermore, the methodology presented in this thesis (jointly developed with Muralidaran Vijayaraghavan) provides an efficient way of building cycle-accurate models on FPGAs. The methodology is efficient in the sense that both the simulator RTL for FPGAs and the RTL for ASIC-synthesis can be generated from the same source.

1.1 The case for FPGA-based modeling

Timing-accurate simulation of multicore processors on multicore hosts has proven remarkably difficult. It usually entails an exchange of timing tokens which represents an increasing overhead as the various simulator threads get out of phase. The techniques for parallel discrete event simulation (PDES), *i.e.*, how to simulate the timing of large multicore processor architectures in parallel on multicore hosts, are discussed in detail in [2]. In PDES, events are distributed among the many host cores and executed concurrently to provide the illusion of a global order. PDES techniques can be either pessimistic, requiring synchronization every time there is an ordering violation, or optimistic with speculative execution, requiring roll-back on ordering violations. In either case, the level of detail implemented in the timing model determines both its accuracy and its speed. Perhaps, for this reason very few distributed simulators model time accurately.

FPGAs, because of their "sea of gates" kind of organization, can mimic processors much more directly, avoiding many layers of interpretation necessary in any software simulator. Often, even with an order of magnitude slower implementation clock, FPGA-based simulators can outperform a simulator running on a general purpose processor. However, one has to be cautious of two things: 1) FPGAs are difficult to program, and 2) even if RTL for the processor being simulated is available (that is a big if), it is generally not suitable for simulation on FPGAs. Experience has shown that there are many hardware structures that map very well to ASICs but not to FPGAs.

HAsim is arguably the first simulator on FPGAs which was designed deliberately

to preserve cycle-accurate behavior. The methodology for constructing simulators used in this thesis, like HAsim, abstracts time in terms of enqueues and dequeues into queues, which correspond to wires in the target design. However, our tools and methodology relieves the designer of having to think in terms of queues by letting him express the design as a collection of blocks, each specified as a cycle-level state machine. This has the advantage of avoiding much tedium in design, as well as maintaining a clear cycle-level specification of the machine being simulated.

There have been many other efforts aimed at building multicore processor simulators on FPGAs. We present them in detail in Chapter 3, and describe how our modeling technique and our FPGA-based simulator, Arete [3], differs from them.

1.2 The case for cycle-accurate modeling

Besides using a faster substrate, architectural simplifications are also widely used to speed up processor simulations. To understand architectural simplifications, let us consider the following scenario. Suppose we want to study how much improvement the LRU replacement policy provides over the random replacement policy, in the shared last-level cache of a multicore processor. Whether the expense of implementing LRU is justified, depends on its quantitative benefits. One may also be interested in whether these benefits vary with each benchmark, and with the number of cores in the processor.

Of course, one generally has limited time to answer these questions in a real design setting. An accurate simulator that includes detailed models of core, memory and on-chip network, will require a lot of time and effort to build, and, even if available, will be quite slow to execute. A detailed cycle-accurate software simulator may execute at 10-100 KIPS [4] and take a few days to completely run one benchmark. Since the evaluation of replacement policies is limited to the cache, it can be argued that a detailed model of the core is not necessary because core behavior is only remotely linked to cache and network behaviors. It is indeed possible to run the same benchmark on a simplified core model in a matter of hours as opposed to days. If

one's intuition about the irrelevance of the core architecture is correct then a lot of time and effort can be saved in simulation. In this thesis we emphatically answer this question in the negative.

When we performed the replacement policy experiment using a cycle-accurate core model, the results matched our intuition completely, *i.e.*, LRU increased the cache hit rate, decreased the memory traffic and improved the overall performance, as depicted by the blue bars in Figure 1-1.

To test the supposed irrelevance of the simplified core model, we performed the replacement policy experiment using the 1-IPC core model. On 1-IPC, instructions which do not incur cache misses are executed in 1 cycle. Only stalls due to cache misses are modeled, while speculative instructions and data hazards are not modeled. Such simplified cores are used often in large multicore processor studies.

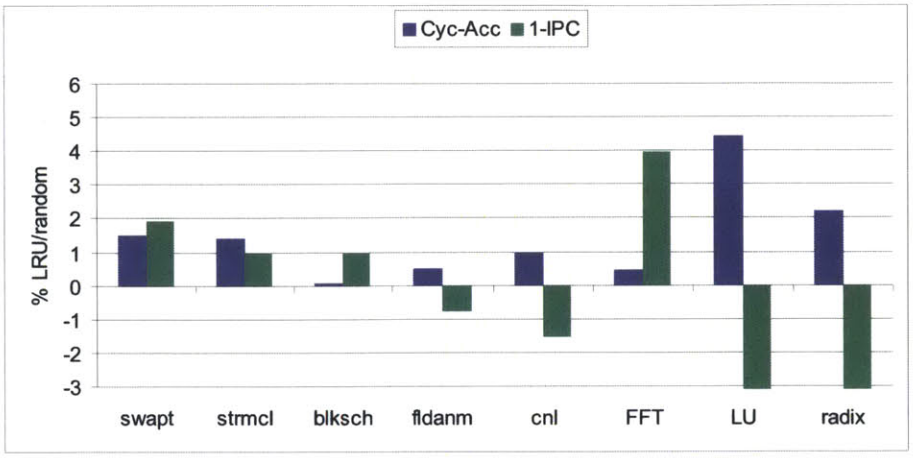
We found that when 1-IPC was used in the replacement policy experiment, the benefits of LRU over random were no longer definitive. Roughly half the benchmarks exhibited opposite trends, as depicted by the green bars in Figure 1-1. When using 1-IPC, one would not be able to conclude that LRU is better than random. It was, however, quite clear when we used the cycle-accurate model. In Chapter 6, we discuss in detail where this disparity in results comes from.

There is a large body of work which explores the use of simplified and abstract core models in processor simulations, and its impact on simulation accuracy. We describe these efforts and contrast them with our work in this domain in Chapter 5 and Chapter 6.

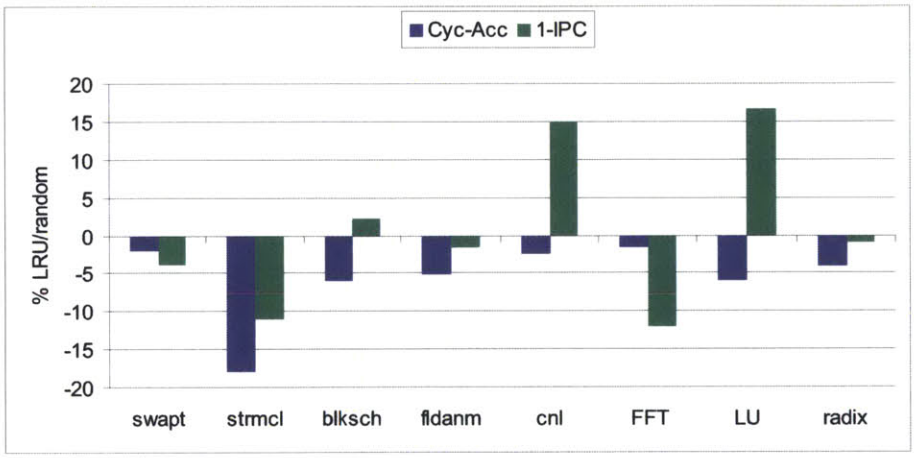
1.3 Summary of contributions

The contributions of this thesis can be divided into two main categories: FPGA-based modeling and architectural exploration using cycle-accurate simulation.

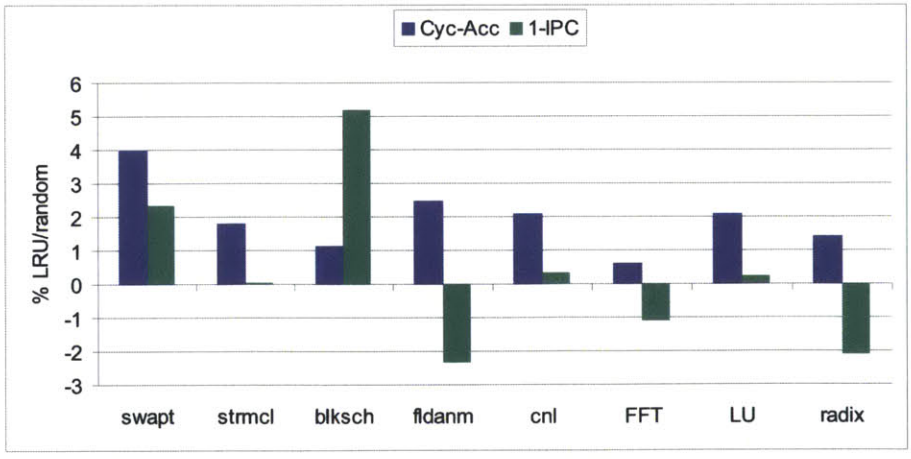
We present a modeling methodology, which starts with a cycle-level specification of the target processor design. We show how the specification can be transformed into a latency-insensitive bounded dataflow network (LI-BDN) [5] and refined to achieve



(a) Cache hit rate



(b) Memory traffic



(c) Overall performance

Figure 1-1: Performance impact of the LRU replacement policy determined using both the cycle-accurate and the 1-IPC core models. Baseline replacement policy is random.

a resource and timing efficient FPGA implementation. The specification can also be compiled into RTL for ASIC implementation and validation of the refined LI-BDN implementation.

Using our modeling methodology we built Arete [3], an FPGA-based full-system cycle-accurate multicore processor simulator with detailed core, memory and network models. Arete boots SMP Linux and runs multithreaded applications, achieving 55 MIPS performance on 8 cores. We also demonstrate its flexibility for architectural exploration and portability across FPGA platforms.

We present a general technique for building a deterministic, model-cycle-level debugging infrastructure [6], based on the LI-BDN modeling methodology. We demonstrate the technique by building a comprehensive debugging infrastructure for Arete. We show that this debugging infrastructure provides a rich set of features, while incurring small resource and performance overheads. It allows for stopping and starting any module in the processor model independently by making a novel use of the provisions of the LI-BDN methodology, and avoids complex forwarding and rollback mechanisms. It also allows us to remove the non-determinism from events such as DRAM access, network access and I/O, without keeping expensive logs.

We ask the question: Can we reliably study architectural changes in the memory hierarchy or the on-chip network of a multicore processor using a simulator that includes detailed cycle-accurate models of memory and network, but a simplified model of core? We provide empirical evidence that the use of simplified core models, such as 1-IPC, leads to conclusions that are wrong both quantitatively and qualitatively. We also give reasons for the error in results. Finally, we show that the error magnitude in such studies increases with the number of cores.

1.4 Document outline

The remaining document is organized as follows.

Part I

- Chapter 2 presents our modeling methodology. It describes the development

of a cycle-level specification for processor microarchitecture, the transformation of the specification into an LI-BDN, and its refinement to achieve an efficient FPGA implementation.

- Chapter 3 describes our efforts to build an FPGA-based cycle-accurate multicore simulator called Arete. It also presents the comprehensive simulation infrastructure included in Arete and its flexibility and portability.

- Chapter 4 presents a general technique for deterministic, model-cycle-level debugging based on LI-BDNs. It describes an application of the technique to build the debugging infrastructure for Arete.

Part II

- Chapter 5 analyzes the impact of abstract models and abstraction parameters on the accuracy of single-core processor simulations.

- Chapter 6 explores if we can reliably study architectural changes in the memory hierarchy or the on-chip network of a multicore processor using a simulator that includes detailed cycle-accurate models of memory and network, but a simplified model of core.

- Chapter 7 presents another architectural experiment, Data Movement Control (DMC), which comprises of new instructions, architectural enhancements and runtime support to enable software-based cache management and computation migration.

- Chapter 8 provides a summary of the work presented in this thesis. It also discusses some new projects in which Arete is being used. These include power modeling, improving the accuracy of 1K-core processor simulations, and hardware/software codesign.

Part I

FPGA-based Modeling

Chapter 2

Functional and Timing Specifications for a Cycle-Accurate Model[1]

2.1 Introduction

As mentioned in Chapter 1, simulation speed has always been a major issue in simulating computer systems. Even though the machines on which we simulate are getting faster or have increasing number of cores, the simulation speed cannot keep up with the ever increasing complexity and size of simulation studies that the designers want to perform. In the last few years the advent of FPGA-based simulators has changed the landscape. Projects like CMU's ProtoFlex [7], Intel-MIT's HAsim [8], UT Austin's FAST [9] and Berkeley's RAMP Gold [10] have shown that it is possible to gain one to three orders of magnitude in performance over detailed software simulators. Yet, many questions remain. For example, what target microarchitecture is being modeled by the simulators? And how difficult are FPGA simulators to write and modify as compared to software simulators?

The collective experience of the community in writing FPGA-based simulators

[1]The work presented in this chapter was jointly carried out with Muralidaran Vijayaraghavan.

shows that the RTL that is suitable for ASIC synthesis is almost never suitable for mapping on to FPGAs; it tends to make inefficient use of FPGA resources. Thus, people have devised techniques which allow an operation that is performed in one model clock cycle to take multiple FPGA clock cycles while keeping track of the model time [4, 9, 10]. We will refer to the RTL that explicitly keeps track of the model time as T-RTL for *Timed RTL*, and the RTL that does not, as D-RTL for *Direct RTL*.

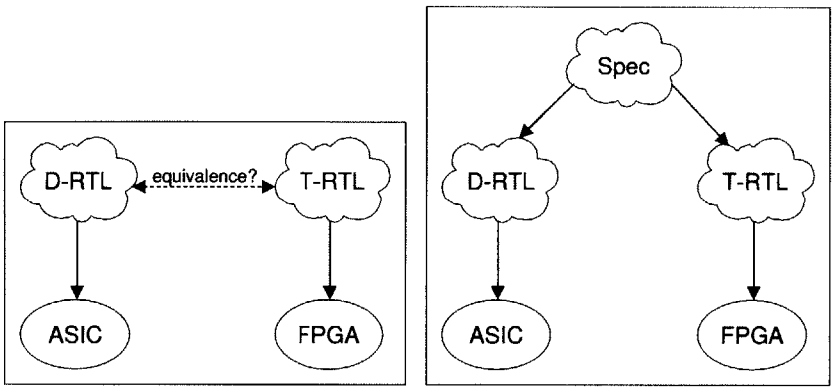
In this chapter we describe a method for writing cycle-accurate specifications of processor microarchitecture in terms of high-level cooperating synchronous sequential machines, and compile these specifications into T-RTL (Section 2.2). T-RTL can be further optimized for FPGA implementations without compromising the specifications (Section 2.3). If desired, our specifications can also be compiled into D-RTL, which can be used to synthesize an ASIC or validate T-RTL.

2.2 Cycle-accurate specifications

Intuitively, cycle-accurate specifications of a machine describes its behavior for each clock cycle. The behavior may be characterized as the values of all the machine's state elements (registers, memories, etc.) every clock cycle. Sometimes it is sufficient to consider only a subset of the state elements, *e.g.,* the program counter and the register file, in our specifications, and ignore others, *e.g.,* the pipeline registers inside a multiplier.

To give the timing specifications for a processor it is not sufficient to say that the adder takes 1 cycle, the multiplier takes 3 cycles, caches have a hit latency of 1 cycle and a miss latency of 16 cycles, etc. The designer also needs to specify which modules are pipelined, which bypass paths are present, and in case of the reorder buffer, what operations are done concurrently. This level of specification is usually available only in the D-RTL description of a machine, which is itself generated from low-level hardware description languages (HDLs), like Verilog or VHDL.

It is an accepted fact that it is tedious to write D-RTL for large systems, and also



(a) Traditional (b) Our methodology

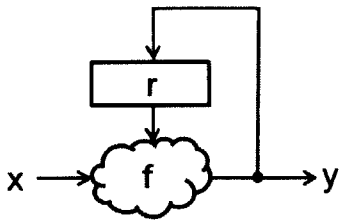
Figure 2-1: Approaches to FPGA-based modeling

that D-RTL is almost never flexible enough for the kinds of changes that designers want to make for architectural exploration. Software simulators for architectural exploration came into vogue precisely to alleviate this flexibility problem. However, the problem of cycle-accurate specifications has remained and there is a constant debate about which timing aspects are being modeled correctly or incorrectly by a given simulator.

2.2.1 Timed RTL (T-RTL)

As we said in the introduction, projects like ProtoFlex [7], HAsim [8], FAST [9] and RAMP Gold [10], have all developed fast FPGA-based processor simulators by thoroughly optimizing them for the FPGA substrate. For example, they avoid using CAM-like structures because CAMs map poorly to FPGAs. Instead, they rely heavily on FPGA-specific structures, like Block RAMs for large register arrays, and DSP slices for complex computations, such as floating point multiplications. Without such optimizations, the D-RTL for complex processor microarchitectures consumes too many FPGA resources, making it impractical to use the target processor's D-RTL directly, even if it were available. In industry, FPGA-based emulation is done using D-RTL, but it requires tremendous amount of FPGA resources. Moreover, the emulation speed is in the 1 MHz range.

For cycle-accurate simulations on FPGAs, RTL is usually written in a highly styl-



$$\begin{aligned} y(i) &= f(x(i), r(i)) \quad , i \geq 0 \\ r(i+1) &= y(i) \qquad\qquad\quad , i \geq 0 \end{aligned} \tag{2.1}$$

where f is a combinational circuit

Figure 2-2: A synchronous sequential machine (SSM)

ized manner, where one explicitly keeps track of the *model clock*, and the amount of work in a model clock cycle can take many FPGA or *implementation clock* cycles. The events in the model time are often represented as enqueue and dequeue operations in this type of RTL which we refer to as T-RTL or *Timed RTL*. Since T-RTL is significantly different from the D-RTL of the target processor, one needs to develop a notion of equivalence between the simulator and the target machine in order to establish the cycle-accuracy of the simulator (see Figure 2-1(a)). Unfortunately D-RTL is almost never available at the time of architectural exploration and most designers of cycle-accurate simulators work with informal timing specifications which are never written down explicitly.

We propose the modeling methodology illustrated in Figure 2-1(b), where we first develop the cycle-accurate specifications of the target system. These specifications can then be used to generate automatically either D-RTL for an ASIC implementation or T-RTL for an FPGA implementation. This T-RTL conforms to the specifications of the target system by construction, and it can be optimized further in a modular manner without affecting its conformity to the specifications.

2.2.2 Timing specifications for a processor

Our timing specifications are built using Synchronous Sequential Machines (SSM) which may be characterized as shown in Figure 2-2. Precise timing specifications for a complex processor can be built by specifying it as an appropriate composition of

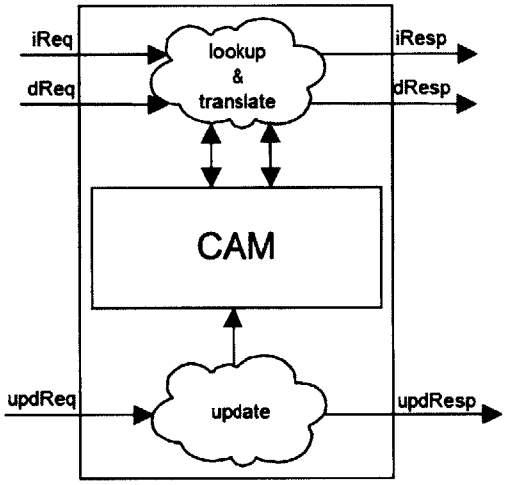
SSM modules, each corresponding to a pipeline stage or some other major block in the microarchitecture. The composition of SSMs is straightforward and results in an SSM. Our specifications are considerably easier to write than any type of RTL, and we have the tools to generate both D-RTL and T-RTL from our specifications. We demonstrate the level of detail in our specifications through two examples.

A shared TLB for instruction and data memories

We describe the specification of a TLB which is shared between instruction and data memories, and is managed by software. As shown in Figure 2-3, the TLB can have up to three simultaneous requests: an i-side address translation, a d-side address translation and a TLB update. The presence of a request is shown by an associated valid bit. Similarly, the presence of each response is also indicated by a valid bit. An address translation request returns either a hit with a page number or a miss. A TLB update request can either invalidate or update an entry and generates an acknowledgement when the operation has been completed. A description of such a TLB is given in our language in Figure 2-3.[2]

In this TLB description the response is always generated in the same cycle, and thus a response can be invalid only if the input request is invalid. However, we could have also written a different specification where it would take several clock cycles to do the lookup and the update, without changing the interface. A correct use of this module would require that a new request not be issued until the previous one has been satisfied. The user of this module should accept the response in the cycle in which it becomes available, *i.e.,* valid, otherwise the response will be lost.

[2]**Syntax notes:** Due to extensive use of `Valid/Invalid` and `Hit/Miss` signals we use the syntax of tagged-union types which are common in functional languages, and can also be expressed in C++. Thus, one can test `iReq` by writing `iReq.valid` and extract the address from a valid request by writing `iReq.virtPN`. `iResp` can be constructed by writing `Invalid` or `Valid Hit ppn` or `Valid Miss`. Another syntax point to note is that we use <= to specify a register or state update, and use := to write to an output. Such assignments can only be used at most once per clock cycle per variable.




```
interface TLB;
  Input iReq, dReq, updReq;
  Output iResp, dResp, updResp;

module TLB mkTLB {
  Reg entries[sizeTLB] (initial Invalid);

  every clock cycle {
    local iRespLocal = iReq.valid ? Valid Miss : Invalid;
    local dRespLocal = dReq.valid ? Valid Miss : Invalid;

    foreach i in [0, sizeTlb)
      if(tlb[i].valid)
        if(iReq.valid && iReq.virtPN == tlb[i].virtPN)
            iRespLocal = Valid Hit getPhysPN(tlb[i]);
        if(dReq.valid && dReq.virtPN == tlb[i].virtPN)
            dRespLocal = Valid Hit getPhysPN(tlb[i]);

    if(updReq.valid)
      foreach i in [0, sizeTlb)
        if(updReq.op == Inv)
          if(updReq.virtPN == tlb[i].virtPN)
            tlb[i] <= Invalid;
      if(updReq.op = Write)
          tlb[updReq.index] <= Valid upd.entry;

    iResp := iRespLocal;
    dResp := dRespLocal;
    updResp := updReq.valid;
  }
}
```

Figure 2-3: Specification of a shared TLB

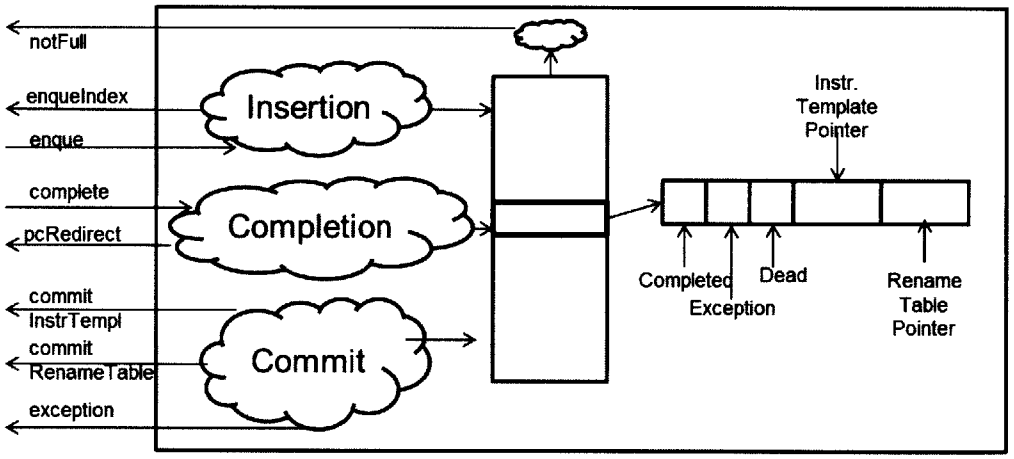


Figure 2-4: Specification of a completion table

Completion table

As a more complex example, we describe a completion table (CT) which is used in some out-of-order machines (Figure 2-4). A CT is an associative structure that has all the functionality of a traditional Reorder Buffer (ROB) except for issuing instructions to the functional units. Each valid entry in a CT corresponds to an instruction and contains `<completion bit, exception bit, dead bit, pointer to the instruction template, pointer to the rename table>`. There are the usual head and tail pointers associated with a CT where head points to the slot for the next instruction and tail points to the oldest instruction to be committed. The issue unit knows the index of the slot in the CT corresponding to each instruction template. Following operations are performed in a CT.

Insertion It is invoked by the instruction dispatch unit. Given a pair of pointers to an instruction template and a rename table, an insertion operation stores the pointer in the head slot, sets the completion, exception and dead bits to false and increments the head pointer. It also exports the recently allocated slot for the issue unit.

Completion It is invoked by a functional unit (FU) when it completes an operation. When an FU completes the operation corresponding to the instruction in the i^{th} slot, the completion and exception flags are set appropriately for the i^{th} slot in the CT. In case of a mispredicted branch, if the entry is not already marked as dead, the correct

```
interface CT
  Output notFull, enqueIndex; Input enque;
  Input complete[numCompletes]; Output pcRedirect;
  Output commitInstTempl, commitRenameTbl, exception;

module CT mkCT {
  Reg entries[sizeROB] (initial Invalid);
  Reg head, tail, numElems (initial 0);

  every clock cycle {
    local cNumElems = numElems;
    local cEntries = entries;
    notFull := numElems != sizeRob;
    enqueIndex := head;
    if(enque.valid)
      cEntries[head] = {comp: False,
                        excep: False,
                        dead: False,
                        instTemplPtr: enque.instTemplPtr,
                        renameTblPtr: enque.renameTblPtr};
      cNumElems++;
      head <= head + 1;

    local pcRedirectLocal = Invalid;
    foreach i in [0, completesNum)
      if(complete[i].valid)
        cEntries[complete[i].index].comp = True;
        cEntries[complete[i].index].excep = complete[i].excep;
        if(complete[i].misPred && !cEntries[complete[i].index].dead)
          pcRedirectLocal = Valid complete[i].newAddr;
          foreach j modulo_in (i, tail)
            cEntries[j].dead = True;
    pcRedirect := pcRedirectLocal;

    if(numElems != 0 && (cEntries[tail].comp || cEntries[tail].dead))
      commitInstTempl := Valid cEntries[tail].instTemplPtr;
      commitRenameTbl := Valid {dead: cEntries[tail].dead,
                                ptr: cEntries[tail].renameTblPtr};
      exception := cEntries[tail].dead? False : cEntries[tail].excep;
      tail <= tail + 1;
      cNumElems--;
    else
      commitInstTempl := Invalid;
      commitRenameTbl := Invalid;
      exception := False;

    numElems <= cNumElems;
    entries <= cEntries;
  }
}
```

Figure 2-4: Specification of a completion table (cont.)

program counter is sent to the fetch unit. The dead bits of all the slots from i to head are set. This operation has to be performed for each functional unit that completes in the same cycle.

Commit If the oldest entry in the CT is either complete or dead, the commit operation either commits or discards it. It exports the pointer to the instruction template for the committed instruction so that the instruction template entry can be freed. It also exports the pointer to the rename table for the committed instruction along with the dead bit that tells the rename table whether the registers written by the instruction are to be discarded or committed into the architectural state. Finally, if the instruction is not dead, but the exception flag is set, the exception is sent to the fetch unit which services it by fetching from a known interrupt handler address. The tail pointer is incremented after completing the commit operation.

2.2.3 Target simplification vs. implementation refinement

The complexity of prevalent and future systems make modeling them very difficult. Moreover, modeling every detail of a target specification can adversely affect the speed of the simulator and consume disproportionate amount of resources. In order to overcome these difficulties, often times, the target specification is simplified. Some of the common examples of target simplification are unaligned memory references and variation in DRAM latency because of access patterns.

Sometimes the changes in specification are motivated by implementation concerns. Consider the specification of a processor with a single-cycle multiplier. In the FPGA-based model of the processor, we may choose to replace the single-cycle multiplier with a 4-cycle unpipelined multiplier to reduce the resource requirements and improve the FPGA clock speed. We could make use of such a multiplier by changing the processor specification so that it can accept a 4-cycle multiplier. Changing the specification may be justified on the basis that the multiplier is used infrequently and would not affect the overall performance estimates significantly. However, changing the processor specification to tolerate a 4-cycle latency may not be as straightforward as it seems because it may make the entire specification functionally incorrect. Cycle-

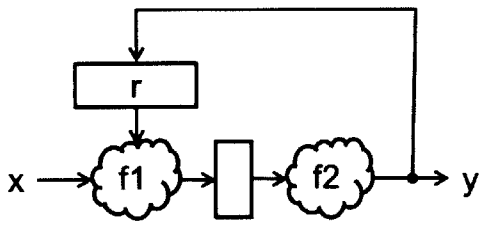


Figure 2-5: A refined SSM

accurate specifications by their very nature are quite brittle and can easily become functionally incorrect even with the smallest of changes.

Another way to replace the single-cycle multiplier with a 4-cycle multiplier is to change the implementation of the model in such a way that when the 4-cycle multiplication takes place, the rest of the model remains frozen. In this way, one can reproduce the state of the processor every model cycle by reading the value of all the registers every fourth FPGA cycle. Here, we are still simulating a processor whose specification has a 1-cycle multiplier. Only the implementation of the model is refined to take 4 cycles for every multiply operation, while keeping track of the model clock. We refer to this technique as implementation refinement and elaborate on this in the next section.

We always maintain a clear distinction between target simplifications and implementation refinements and generally do not simplify the target specifications to meet FPGA resource constraints.

2.3 Implementation refinements

As discussed in the previous section, we need a way to refine the implementation of a target specification to optimize it for the FPGA fabric, while accurately reproducing the values of the state every model cycle. In FPGA-based simulators, different modules of a simulator operate in parallel. Two modules, after refining, may take different number of FPGA cycles to simulate one model cycle.

For example, consider a refinement of Figure 2-2 where f is replaced by f_1 and f_2 where $f(x(i), r(i)) = f_2(f_1(x(i), r(i)))$ and the length of the critical path is reduced

by inserting a register between f_1 and f_2 (see Figure 2-5). If this refined implementation is used in place of the original SSM, then the rest of the circuit connected to this module must be changed to account for the 1-cycle latency. A large body of theoretical work on making such refinements has been produced in recent years (see for example, Carloni *et al.* [11], Vijayaraghavan *et al.* [12], Krstic *et al.* [13]). All these techniques essentially model the time explicitly in the circuit itself and ensure that the cycle-by-cycle behavior of the original SSM is preserved. Here, we elaborate on Vijayaraghavan *et al.* technique called Latency-Insensitive Bounded Dataflow Networks (LI-BDNs) [12].

2.4 The LI-BDN technique for writing cycle-accurate simulators

The LI-BDN technique models the timing of the SSM in terms of enqueue and dequeue operations on the input and output queues. Thus the i^{th} input and the i^{th} output in an SSM correspond to the i^{th} dequeue operation on the input queue and the i^{th} enqueue operation on the output queue, respectively. The refinement of an LI-BDN module may introduce new logic and state, but it has to preserve the timing behavior by recreating the values assumed by the input and output wires and the original module state, for each cycle of the original SSM, referred to as the model cycle. The use of LI-BDNs makes it easy to synchronize the model cycle across different modules where each module can take different FPGA cycles to simulate one model cycle. The technique also works across multiple FPGAs.

We give a brief overview of the LI-BDN technique using the example of a multi-ported register file module. We start with the cycle-level specification given in Figure 2-6 and depicted in Figure 2-7(a). The module can take in three requests simultaneously: reading of two register values, and update of one register value. The presence of the update request is indicated by an associated valid bit. If all the requests are present simultaneously, and either of the registers being read is also being

```
module regFile {
  Input rdReg1, rdReg2, upd;
  Output valReg1, valReg2;
  Reg entries[ sizeRF ] rf ( initial 0 );

  every clock cycle {
    if( upd.valid ) {
      rf[ upd.idx ] <= upd.val;
    }

    valReg1 := upd.valid && rdReg1 == upd.idx ? upd.val :
                                                rf[ rdReg1 ];
    valReg2 := upd.valid && rdReg2 == upd.idx ? upd.val :
                                                rf[ rdReg2 ];

  }
}
```

Figure 2-6: Synchronous specification of a 2-read, 1-write register file module

updated, the updated value is bypassed as the read response. Such a specification does not map well to the FPGA fabric in terms of both resources and timing.

We transform the specification into an LI-BDN so that the register array which has three ports and combinational reads can be simulated with a Block RAM which has two ports and one-cycle-latency reads. We start by attaching FIFOs to all the ports and done flags to all the output ports, as shown in Figure 2-7(b). Note that these FIFOs are in addition to the FIFOs which may be part of the synchronous specification. Now as Figure 2-7(c) depicts the `valReg1` output depends on the `rdReg1` and the `upd` inputs, which are both available. So we enqueue `valReg1` and set its done flag. We handle the `valReg2` output in the same manner. Finally, after all the outputs are enqueued and all the inputs are available, we update the Block RAM, dequeue all the inputs and reset all the done flags, as shown in Figure 2-7(d). The control logic for the LI-BDN transformation of the register file module is provided in Figure 2-8.

The conversion from a specification into an LI-BDN module is what we call the *LI-BDN transformation* of a module [5, 14]. The two requirements, that an output waits only for the inputs that it depends on, called the no-extraneous dependencies (NED) requirement, and that all the input FIFOs are dequeued when all the inputs are available and all the outputs have been produced, called the self-cleaning (SC) requirement, together guarantee the absence of deadlocks from the LI-BDN transfor-

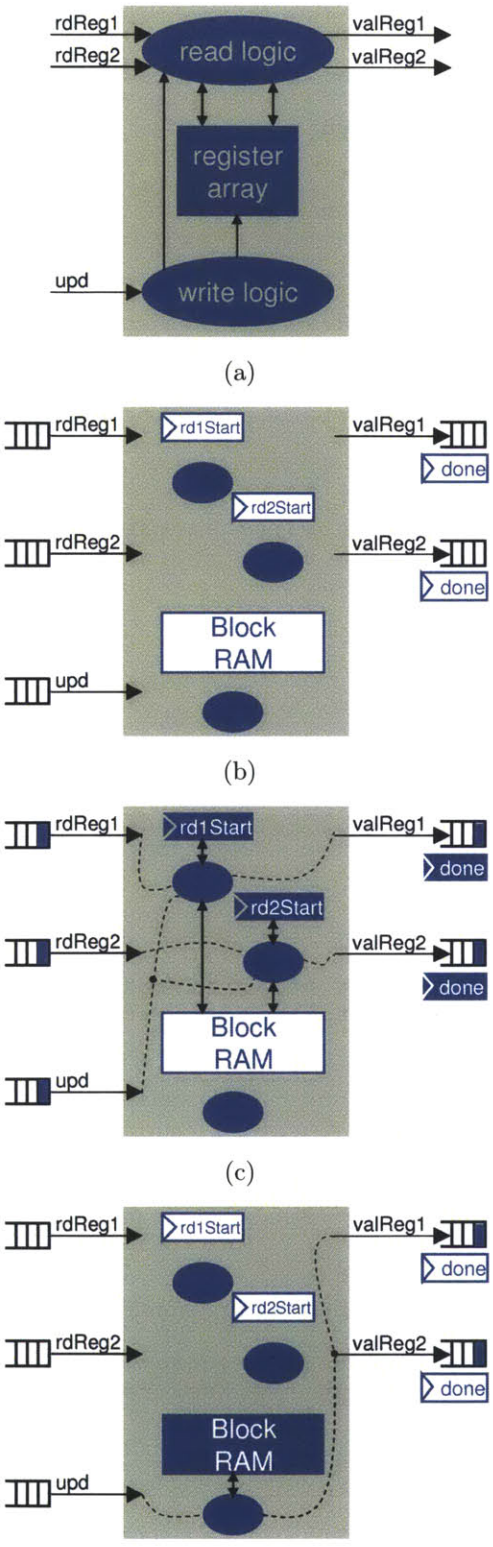


Figure 2-7: Transforming a cycle-level specification into an LI-BDN module

```
libdn regFile {
  LiBdnIn rdReg1, rdReg2, upd;
  LiBdnOut valReg1, valReg2;
  BlockRAM entries[ sizeRF ] rf ( initial 0 );
  Reg rd1Start, rd2Start ( initial False );

  rule rd1 {
    if( !valReg1.done && !valReg1.full && !rdReg1.empty
        && !upd.empty && !rd1Start )
    {
      rf.req1( Read, rdReg1.first, DontCare );
      rd1Start <= True;
    }

    if( rd1Start )
    {
      valReg1.enq( upd.first.valid && rdReg1.first == upd.first.idx ?
                   upd.first.val : rf.resp1 );
      valReg1.done <= True;
      rd1Start <= False;
    }
  }

  rule rd2 {
    if( !valReg2.done && !valReg2.full && !rdReg2.empty &&
        !upd.empty && !rd2Start )
    {
      rf.req2( Read, rdReg2.first, DontCare );
      rd2Start <= True;
    }

    if( rd2Start )
    {
      valReg2.enq( upd.first.valid && rdReg2.first == upd.first.idx ?
                   upd.first.val : rf.resp2 );
      valReg2.done <= True;
      rd2Start <= False;
    }
  }

  rule finish {
    if( valReg1.done && valReg2.done )
    {
        if( upd.first.valid )
        {
          rf.req1( Write, upd.first.index, upd.first.val );
        }
        rdReg1.deq; rdReg2.deq; upd.deq;
        valReg1.done <= False; valReg2.done <= False;
    }
  }
}
```

Figure 2-8: Refined LI-BDN register file module

	SSM	LI-BDN	Improvement
Slice LUTs	4039(5.8%)	460(0.7%)	8.78×
Slice flip flops	2240(3.2%)	839(1.2%)	2.67×
BRAMs	0(0.0%)	1(0.7%)	-
FPGA frequency	192.9MHz	229.1MHz	1.19×
FMR	1	4	0.25×
Effective frequency	192.9MHz	57.3MHz	0.30×

Figure 2-9: Comparison of resource and timing statistics for SSM and LI-BDN implementations of a register file with 32×64-bit entries, 2 read ports and 1 write on the XUPv5 board

mation.

The time duration between the enqueuing of the output FIFOs and the dequeuing of the input FIFOs comprises one model cycle for the transformed module. During one model cycle, the transformed module can use any number of implementation cycles to produce the outputs or to update the state. In this manner, the model cycle is decoupled from the implementation cycle which enables an efficient implementation of the model on the desired platform while maintaining model-cycle-level accuracy.

Figure 2-9 provides a comparison of resource and timing statistics for the SSM and the LI-BDN implementations of the register file module. The FMR (FPGA to model cycle ratio) statistic listed in the table is the average number of FPGA cycles used to simulate a model cycle. Although the effective clock frequency of the LI-BDN module of the register file is one-third of the clock frequency of its SSM, typically the opposite is true. The reason is that critical path is typically present in complex logic blocks, such as multipliers and dividers. These blocks slow down the clock for the entire design. LI-BDN modules of these blocks preserve their timing behavior but implement them over many cycles, improving the overall clock frequency. Although the FMR of these LI-BDN modules is high, since multipliers and dividers are infrequently used, the overall FMR of the design remains low. The high overall clock frequency and the low overall FMR result in a higher effective frequency than that of the SSM.

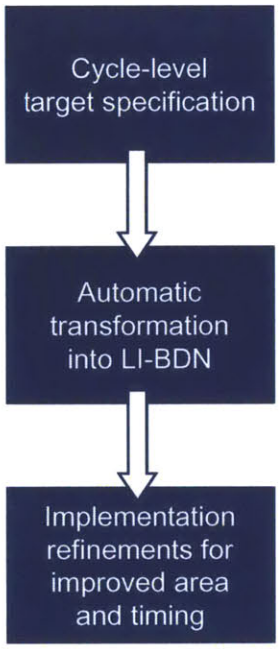


Figure 2-10: Modeling methodology

We have built a library of FPGA-optimized components which make use of Block RAMs and DSP slices which are used to implement modules such as a multi-ported register file, a Reorder Buffer or complex combinational logic like multiplication and division efficiently. Moreover, if the resulting simulator is too large to fit into a single FPGA, we partition it across different FPGAs. We create identical partitions and use LI-BDNs to preserve cycle-level behavior across them. A general technique for partitioning a large design among multiple FPGAs using latency-insensitive links has been presented by Fleming *et al.* in [15].

2.5 Summary

Figure 2-10 summarizes our modeling methodology. We start by writing a cycle-level specification of the target processor design. This specification is then compiled into an LI-BDN, which is refined to achieve an efficient FPGA implementation. We will describe our FPGA-based cycle-accurate multicore processor simulator built using this technique in Chapter 3.

Chapter 3

Fast and Cycle-Accurate Modeling of a Multicore Processor[1]

3.1 Introduction

In this chapter we present Arete, an FPGA-based cycle-accurate simulator for a multicore PowerPC architecture. We developed this simulator adhering to a cycle-level specification of the architecture. For the purpose of efficient FPGA implementation we used the LI-BDN technique [12] which helps to improve the FPGA cycle time and to reduce the FPGA resource requirements by using multiple FPGA cycles to simulate one cycle of the target architecture. We boot off-the-shelf SMP Linux and run applications such as the PARSEC [16] and the SPLASH-2 [17] benchmark suites on Arete. Our simulator is also suitable for architectural exploration. We demonstrate this by evaluating three branch prediction schemes and four cache line replacement policies, and by extending the cache coherence scheme to provide software with better control over the contents of the caches. We also show how the cycle-accurate models of core and cache hierarchy can be easily modified to create abstract models. We have ported Arete to two single-FPGA platforms (XUPv5 and ML605) and one

[1]The work presented in this chapter includes contributions from Muralidaran Vijayaraghavan and Silas Boyd-Wickizer.

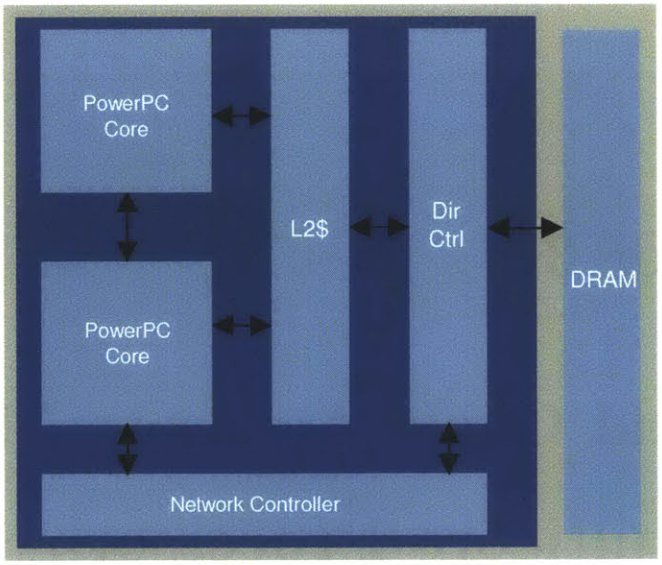


Figure 3-1: Architecture of a processor tile

multi-FPGA platform (BEE3).

To our knowledge Arete is the first cycle-accurate FPGA-based multicore processor simulator which includes both a realistic core architecture and a detailed cache coherence engine. Along with modeling this level of detail, Arete delivers high performance, *viz*, 55 MIPS while simulating eight cores on four FPGAs and up to 11 MIPS while simulating one core on one FPGA.

Chapter organization: Section 3.2 describes the architecture of the processor being modeled. Section 3.3 provides a detailed description of Arete, and provides statistics on its performance and resource utilization. Section 3.4 discusses some of the related work in the areas of multicore processor modeling and the use of FPGAs for implementing these processor models. Section 3.5 provides a summary of our work.

3.2 Processor architecture

The processor makes use of a tiled architecture where the number of tiles is a synthesis parameter that is specified according to the resources available on a particular FPGA platform. As shown in Figure 3-1, each tile is composed of a parameterized number of cores, a shared and inclusive L2 cache, a cache coherence engine and a network

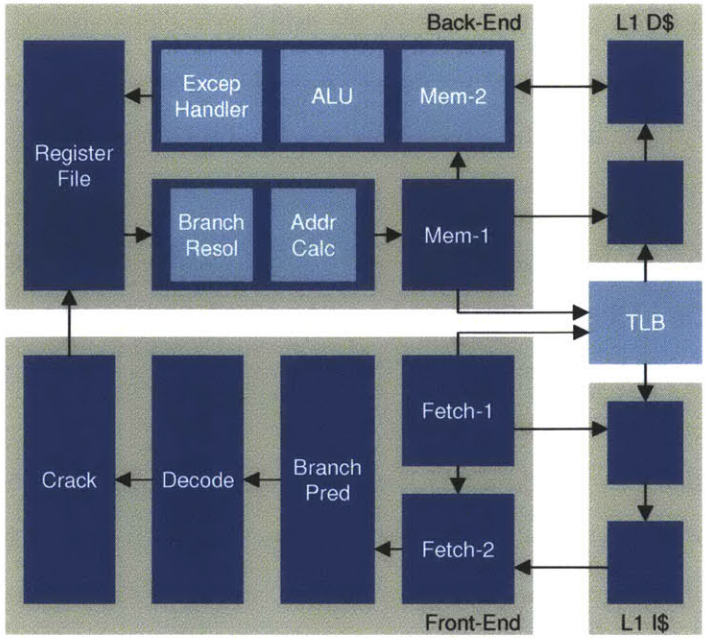


Figure 3-2: Architecture of an in-order PowerPC core

controller. Each tile directly accesses a region of DRAM memory, the size of which is platform dependent. A network layer connects all the tiles in the processor.

3.2.1 Core

The core comprises of a 64-bit, in-order PowerPC pipeline and implements the Power ISA—Embedded Environment [18]. Figure 3-2 shows the microarchitecture of the core. The pipeline is designed to provide a high degree of flexibility, and includes the following features.

(I) Pipeline stages can be split or combined without modifying the rest of the pipeline because the stages are designed to be latency-tolerant. For example, instruction decode may happen over multiple cycles, instead of one. Moreover, the two instruction fetch stages may be combined into one, if the hit path in the L1 cache is combinational.

(II) The mechanism to handle change in instruction flow allows any stage to perform branch prediction, branch resolution or exception handling.

(III) Any stage can read the register file and the various special purpose registers, but only the last stage updates them when committing instructions. Updated register values are fully bypassed, but the pipeline may still stall due to read-after-write hazards.

Each core has private instruction and data L1 caches with a pipelined hit latency of 1 model cycle. These caches are parameterized for associativity, line size, number of entries and replacement policy. The tag and data arrays of the L1 caches are implemented on block RAMs.

The core also has a shared TLB which is parameterized for number of entries, and is implemented using a combination of block and distributed RAMs. It provides multi-ported combinational access for instruction and data address translation, as well as for TLB update. It supports variable-size pages.

Pipeline description

The front-end of the core pipeline comprises of five stages. The fetch-1 stage maintains a branch target buffer (BTB). It sends the program counter (PC) to the first stage of the instruction-side L1 cache and the TLB, and updates the PC based on inputs received from the branch prediction, the branch resolution and the exception stages. The fetch-2 stage receives a single instruction from the second stage of the instruction-side L1 cache. This instruction is forwarded to the branch prediction stage. The branch prediction stage partially decodes the instruction to determine if it is a branch. In case of a branch instruction, it consults a branch history table (BHT) to predict the direction of the branch. The crack stage partially decodes the instruction to determine if it is a complex load or store. In case of a complex load or store, it divides the instruction into several simple load or store instructions and forwards the simple instructions to the decode stage one by one. The decode stage fully decodes the instruction to determine the registers it reads and modifies, and the functional unit it uses for execution.

The back-end of the pipeline also comprises of five stages. In the first stage the register file is read. The next stage determines the address for memory instructions

and the target PC for branch instructions. If either the direction or the target of the branch was mispredicted by the front-end of the pipeline, this stage resets the PC with the correct address. In case of a memory instruction, the memory-1 stage sends the virtual address to the first stage of the data-side L1 cache and the TLB. All other instructions are simply forwarded to the next stage. The execute stage sends data to the data-side L1 cache for store instructions, receives data from it for load instructions and executes all other instructions appropriately. All exceptions are also handled in this stage, *i.e.,* whenever an exception is encountered, it sets the PC to the address of the relevant exception handler. The last stage updates the register file with data computed or obtained from the cache in the execute stage.

The back-end of the pipeline is fully bypassed. However, an instruction may still stall due to RAW hazards, besides stalling because of cache misses. The address calculation stage is the only stage, besides the execute stage, which makes use of register values. So an instruction may be stalled in it, if that instruction reads a register which will be modified by an instruction either in the memory-1 stage or the execute stage.

One of the key features of the core's design is its modularity. It can support a completely different RISC ISA with appropriate modifications confined to the decode and the MMU modules.

3.2.2 Shared memory and cache coherence

Figure 3-3 shows the hierarchical structure of the shared, coherent memory architecture which forms the backbone of the multicore processor. We have designed and implemented a hierarchical, directory-based MSI protocol to provide cache coherence. Figure 3-4 provides the state transitions for cache state, while Figure 3-5 provides the state transitions for directory state. Each level of the memory hierarchy considers the next higher level as its parent, while the next lower level as its child. State (X, Y) represents a transitional state. Although we did not formally verify the coherence protocol, we tested it using an extensive suite of hand-coded microbenchmarks.

The L2 cache is inclusive and is shared by all the cores in a tile. It is parameterized

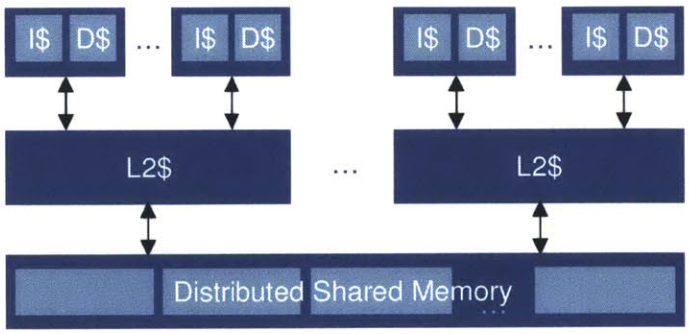


Figure 3-3: Shared memory architecture

for associativity, line size, number of entries, replacement policy and access latency. Access latency and replacement policy are runtime parameters while the rest of the parameters have to be specified before synthesis. The tag arrays and the directory state in the L2 cache are implemented on block RAMs, while the data arrays are mapped to a private region of DRAM. The coherence directory at L2 cache maintains coherence among the L1 caches to which the L2 cache is connected.

We have arranged the main memory in a distributed and shared manner where each tile has fast access to the region of main memory to which it is directly connected, but it has to traverse the network layer to access those regions which are connected to other tiles. Off-chip main memory is incorporated into Arete as an LI-BDN module. This enables us to model its access latency which is another runtime parameter of the model. DRAM latency can be fixed to a particular value or modeled as variable within a certain range. In the latter case, we do not model the variability in the target specification. Instead, we rely on the variable latency of the DRAM on the FPGA board. A private region of DRAM is used to implement the directory state in the main memory which provides cache coherence among all the L2 caches.

Just like the core, the memory subsystem is designed to be quite flexible. One can implement a new cache coherence protocol by modifying the cache coherence engine alone. Similarly, memory organization can be completely altered without modifying the rest of the system, namely the core and the on-chip network.

Current state	Request trigger	Dequeue trigger	Response trigger	Request from parent	Response to parent	Request to parent	Response from parent	Next state
M	St, data	yes						M
M	Ld	yes	data					M
M	Inv	yes			I, data			I
M				S	S, data			S
M				I	I, data			I
S	St, data	no				M		(S,M)
S	Ld	yes	data					S
S	Inv	yes			I			I
S				I	I			I
I	St, data	no				M		(I,M)
I	Ld	no				S		(I,S)
I				S				I
I				I				I
(S,M)		yes					M	M
(I,M)		yes					M, data	M
(I,S)		yes					S, data	S

Figure 3-4: Cache state transitions

Child's current state	Other children's current state	Trigger	Deq. trigger	Req. from child	Deq. req. from child	Resp. to child	Req. to child	Resp. from child	Req. to other children	Resp. to other children	Child's next state	Other children's next state
M	I	S	no				S				(M,S)	I
M	I	I	no				I				(M,I)	I
M	I							I, data			I	I
S	X = S/I	S	yes								S	X
S	X = S/I	I	no				I				(S,I)	X
S	S			M	no				I		S	(S,I)
S	I			M	yes	M					M	I
S	X = S/I							I			I	X
I	M			M	no				I		I	(M,I)
I	S			M	no				I		I	(S,I)
I	I			M	yes	M, data					M	I
I	M			S	no				S		I	(M,S)
I	X = S/I			S	yes	S, data					S	X
(M,S)	I							S, data			S	I
(M,S)	I							I, data			I	I
(M,I)	I							I, data			I	I
(S,I)	X = S/I							I			I	X

Figure 3-5: Home directory state transitions

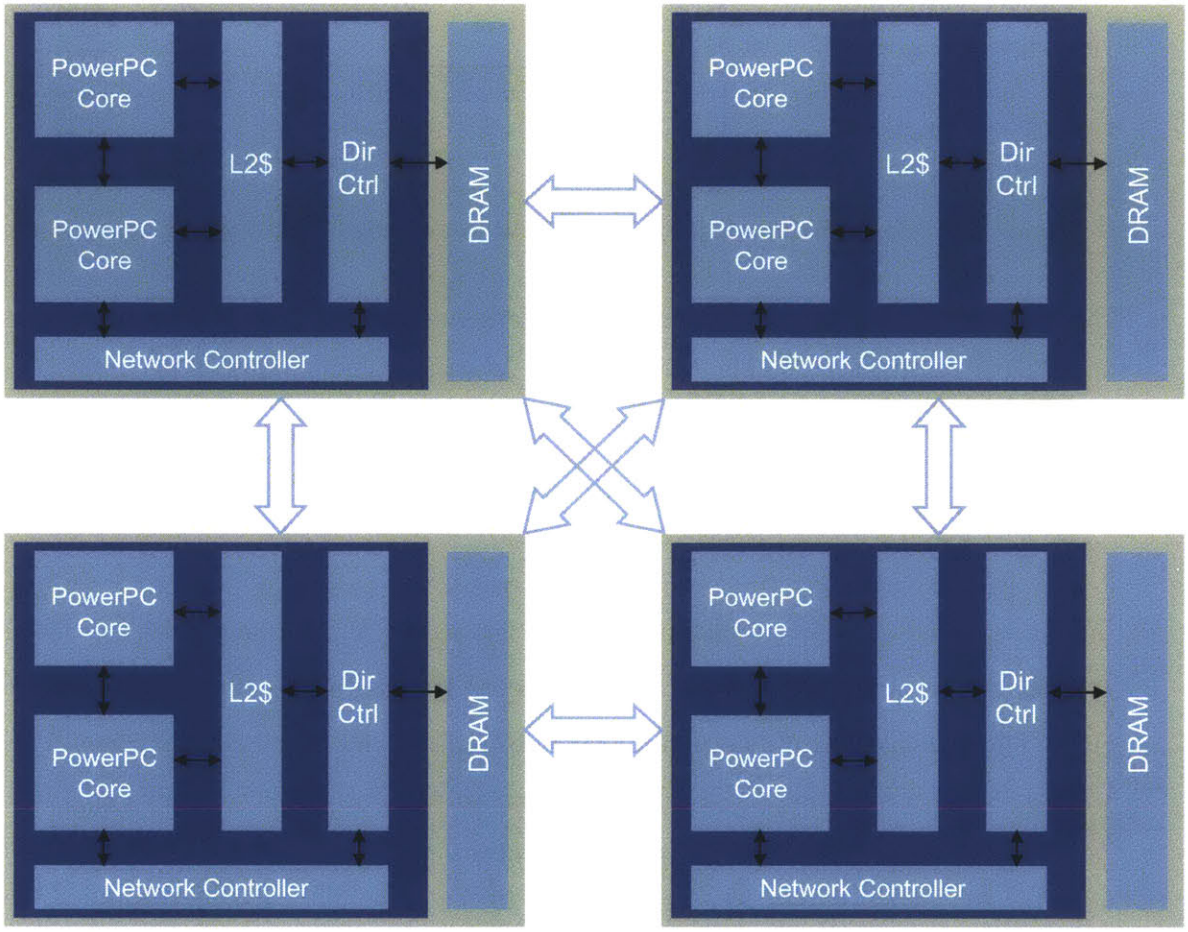


Figure 3-6: Fully connected network topology in Arete

3.2.3 On-chip network

The current implementation of the network architecture supports a bidirectional, fully-connected topology, as shown in Figure 3-6. It is parameterized for per hop latency. It is capable of handling four types of traffic: cache coherence, inter-core messaging, debugging and display, as shown in Figure 3-7. All message types are part of the processor specification. However, the debugger and the display device are only part of the FPGA platform and remain outside the specification.

All messages received by the network layer are first packetized, and then each packet is broken down into flits with parameterized bit width, before being sent across the network. We have built virtual networks for the four kinds of traffic. These virtual

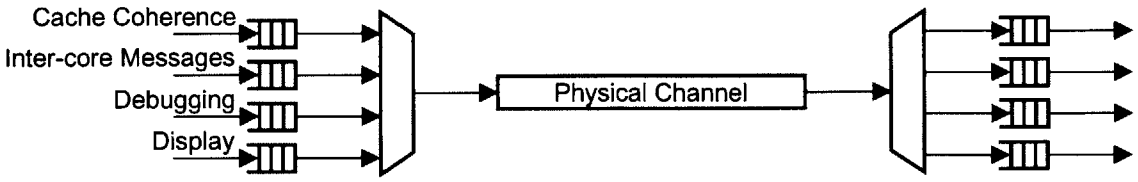


Figure 3-7: Various types of traffic supported by the on-chip network

networks include appropriate amount of buffering and utilize flow control mechanisms to ensure deadlock-freedom. The network model can be modified in isolation to support various other topologies as well as routing algorithms.

3.2.4 Message-passing support

We have added a message-passing layer to the model which allows any core in the processor to communicate with all other cores via messages defined by the Power ISA. The message-passing layer supports both unicast and multicast messages. These messages are used either by the primary core to wake up the secondary cores or by any core to cause a doorbell interrupt in another core.

3.3 Full-system processor simulator

The design and implementation of Arete provides simulation speed and accuracy along with ease of modification and portability. We started by writing a cycle-level specification of the processor, and then employed the LI-BDN technique described in section 2.4 to incorporate various implementation refinements which helped achieve an efficient FPGA implementation. In the process, we built a library of components which may be used for FPGA implementations of other models. We used Bluespec SystemVerilog (BSV) [19] to develop Arete.

In this section we outline the simulation infrastructure provided by Arete, and describe its portability to various FPGA platforms. We also describe the resource savings and performance improvements obtained from using various implementation refinements enabled by the use of the LI-BDN technique. Finally, we evaluate the

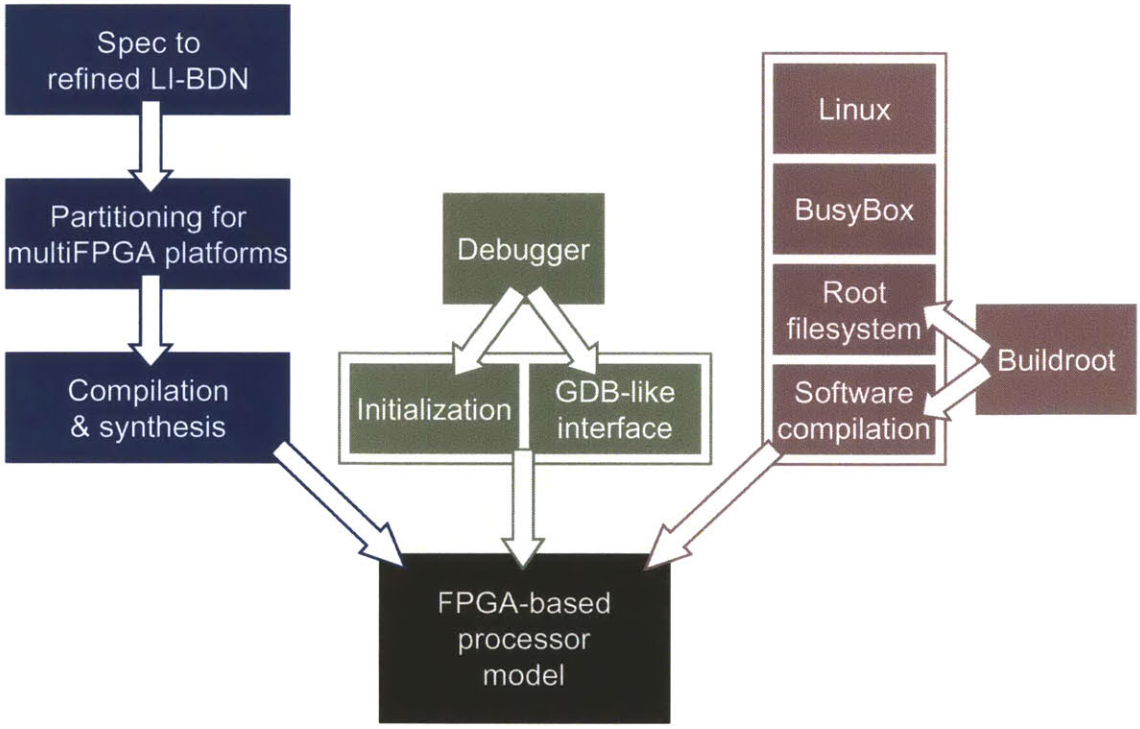


Figure 3-8: Simulation infrastructure

performance of Arete by running the PARSEC benchmark suite on top of SMP Linux.

3.3.1 Simulation infrastructure

As shown in Figure 3-8, we have strived to provide a comprehensive simulation infrastructure for architectural exploration and verification. We make use of the debugging feature enabled by the use of the LI-BDN technique to build a debugging environment for Arete. A MicroBlaze soft core runs debugging software, written in C, which provides a GDB-like interface to the user. The debugging software handles low-level model initialization and provides access to all model state during simulation. Linux 2.6.32 boots on Arete and we use Buildroot [20] to generate a cross-compilation toolchain for the PowerPC architecture, and a root filesystem. We also run the BusyBox package [21] which provides many common UNIX utilities.

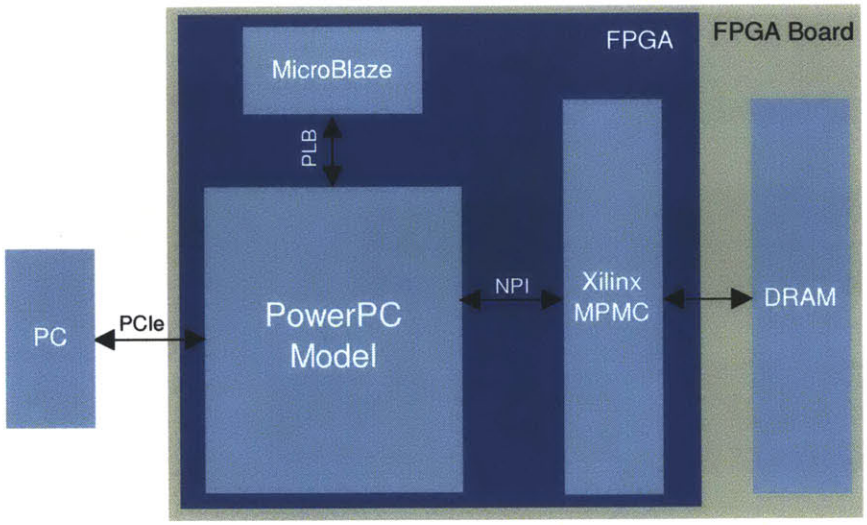


Figure 3-9: A complete view of the FPGA implementation of Arete

3.3.2 Portability across FPGA platforms

As shown in Figure 3-9, the model communicates with three external resources: a Xilinx multi-ported memory controller (MPMC) which provides access to DRAM, a MicroBlaze soft core which runs debugging software, and a PC which provides access to a text terminal. For a particular FPGA platform, we wrap the interfaces to the three resources in order to present latency-insensitive, request-response interfaces to the model. We have ported Arete to three FPGA boards: XUPv5, ML605 and BEE3, which are shown in Figure 3-10. This portability does not require any modifications to the design of the model; one only needs to specify appropriate values of certain parameters before synthesis.

When porting a model to a multi-FPGA platform several issues arise. One of the main issues is that the model has to be explicitly partitioned, and a different configuration file has to be generated for each FPGA, which can become tedious. We have made use of functionally-identical partitions and a distributed protocol for assigning identifiers to each partition. Together they enable one configuration file to program all the FPGAs with tremendous compute savings during compilation and synthesis.

Another issue is that implementing a model on multiple FPGAs can alter its timing

Figure 3-10: Supported FPGA boards

behavior. For example, when a path that is modeled to be two cycles long, originates on one FPGA and terminates on another, it might require six FPGA cycles. We, however, are able to preserve the timing behavior at the model-cycle-level through the use of the LI-BDN technique.

These features are similar to those developed in earlier projects. BORPH [22] is an operating system designed for FPGA-based reconfigurable computers. It augments the Linux kernel with hardware processes which are hardware designs that run on FPGAs, but behave like normal user programs. In order to allow hardware processes to communicate with the rest of the system, BORPH provides them standard system services, such as file system access. Similarly, LEAP [23] provides a set of device abstractions, communication mechanisms and useful services across many FPGA platforms. The goal is to facilitate application development on FPGAs by providing a standardized platform architecture.

3.3.3 Flexibility for architectural experiments

Due to our platform's modularity and parameterization, we were able to conduct various architectural experiments on Arete with moderate effort. The design, verification and evaluation of three branch prediction schemes and four cache line replacement policies each required only 2 man-days worth of work. A significant overhaul of the cache coherence protocol to support software management of caches was carried out in 30 man-days. Moreover, detailed, cycle-accurate models of core and memory were transformed into simplified models in 5 man-days.

The simulation platform has been used in two class projects in the graduate-

	Prototype	LI-BDN	Improvement
LUTs	105104	24153	4.35×
Flip flops	638678	16165	39.51×
Block RAMs	0	43	-
DSP slices	12	12	1.00×
FPGA frequency (MHz)	4.8	110	22.92×
FMR	1	9	0.11×
Effective frequency (MHz)	4.8	12.2	2.54×

Figure 3-11: Comparison of the prototype and the refined LI-BDN implementations of PowerPC on the XUPv5 board. Model parameters: 1 tile, 1 in-order 10-stage core, 64 KB 4-way associative L1 caches, 512 KB 4-way associative L2 cache, 512 MB DRAM

level course, Complex Digital Systems (MIT 6.375). It has also been adopted by our collaborators at IBM Research and Barcelona Supercomputing Center in their research.

3.3.4 Synthesis statistics

In section 2.4 we described how a refined LI-BDN implementation of a cycle-level specification can achieve both higher performance and reduced resource utilization on FPGAs. To gauge the impact of the use of the LI-BDN technique and implementation refinements on Arete, we synthesized both the cycle-level specification of a single-core processor model, that we call the prototype, and its transformed and refined LI-BDN counterpart. The LI-BDN version of the model included such implementation refinements as the 5-ported register file being simulated by a dual-ported block RAM, and complex combinational logic with long critical path being simulated by its multi-cycle counterpart.

Figure 3-11 shows the comparison between the two implementations. The refined LI-BDN implementation uses a fourth of the LUT resources consumed by the prototype and provides a twenty times speedup in the FPGA clock speed. The FMR (FPGA to model cycle ratio) statistic listed in the table is the average number of FPGA cycles used to simulate a model cycle. As mentioned before, the multi-cycle

	LUTs	Flip flops	Block RAMs	DSP slices
Branch prediction	357	611	1	0
Decode	1016	392	0	0
ALU	11134	4426	0	12
L1 I-cache	1982	1795	20	0
L1 D-cache	2923	2203	20	0
TLB	2330	896	1	0
Miscellaneous	5165	6137	1	0
PowerPC core	24907	16460	43	12
L2 cache	4597	5407	24	0
Directory controller	3238	3674	0	0
Network layer	5653	6816	2	0
Peripherals	5980	7207	8	0
Overall	69282	56024	120	24
Utilization	71%	57%	56%	18%

Figure 3-12: Resource utilization for the refined LI-BDN implementation of the PowerPC model and peripherals on the BEE3 board. Model parameters: 1 tile, 2 in-order 10-stage cores, 64 KB 4-way associative L1 caches, 512 KB 4-way associative L2 cache, 2 GB DRAM

implementation of complex combinational logic is infrequently used. This results in an FMR of 9 for the LI-BDN implementation, even though it takes up to 32 FPGA cycles to simulate some combinational logic. The low FMR allows the LI-BDN implementation to provide a 2.5× improvement in performance over the prototype.

Figure 3-12 provides a detailed breakdown of the resources used by the various modules in the refined LI-BDN implementation of a dual-core processor model on one FPGA chip of the BEE3 board [24]. The first section of the table lists the major components of the processor core, while the second section lists the components of the tile. Figure 3-13 and Figure 3-14 provide the synthesis statistics for Arete on the XUPv5 board and the ML605 board, respectively.

The dual-core processor implemented on the BEE3 board has a higher resource utilization than that on the XUPv5 board, because it includes the coherence directory

	LUTs	Flip flops	Block RAMs	DSP slices
PowerPC core	24897	16458	43	12
L2 cache	3173	3319	24	0
Peripherals	1637	2354	0	0
Overall	54604	38589	110	24
Utilization	78%	55%	74%	37%

Figure 3-13: Resource utilization for the refined LI-BDN implementation of the PowerPC model and surrounding peripherals on the XUPv5 platform. Model parameters: 1 tile, 2 in-order 10-stage cores, 64 KB 4-way associative L1 caches, 512 KB 4-way associative L2 cache, 1 GB DRAM

	LUTs	Flip flops	Block RAMs	DSP slices
PowerPC core	24180	16254	43	12
L2 cache	4215	5243	30	0
Peripherals	2454	3056	0	0
Overall	103389	73315	202	48
Utilization	68%	24%	49%	6%

Figure 3-14: Resource utilization for the refined LI-BDN implementation of the PowerPC model and peripherals on the ML605 board. Model parameters: 1 tile, 4 in-order 10-stage cores, 64 KB 4-way associative L1 caches, 512 KB 4-way associative L2 cache, 512 MB DRAM

and logic for main memory, the network model, and the inter-FPGA links, which are absent on the XUPv5 board. Since the Virtex-5 FPGAs on the BEE3 board are much larger than that on the XUPv5 board, the utilization ratios are not very different. The ML605 board has a Virtex-6 FPGA, and the quad-core processor implemented on it has the lowest utilization ratios.

3.3.5 Performance evaluation

We implemented an 8-core processor model on the BEE3 board, where each FPGA chip was programmed to simulate one tile of the processor. We ran a subset of the

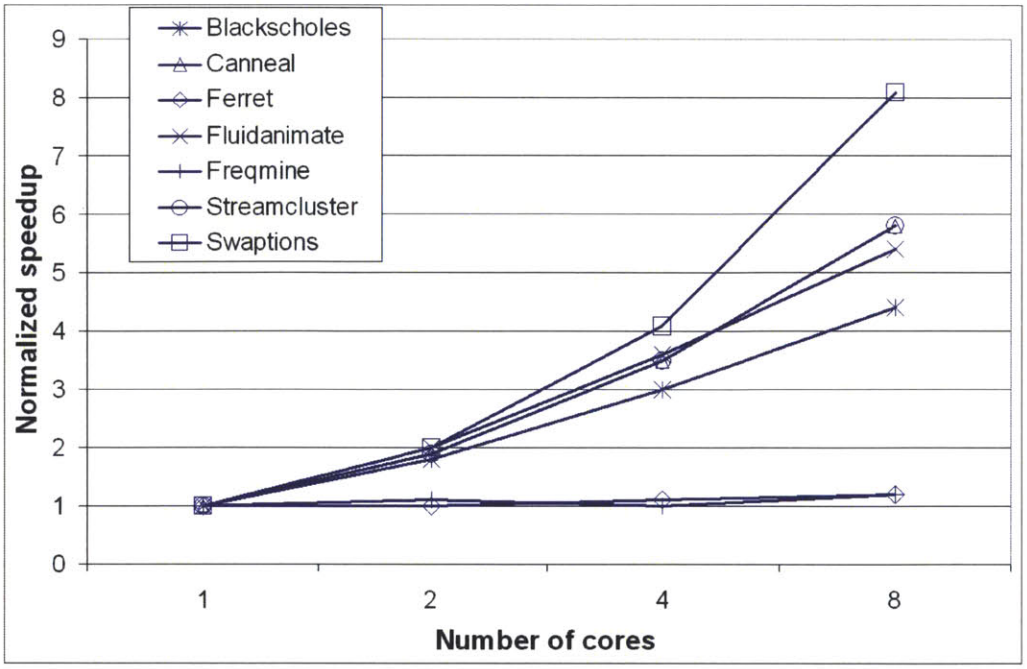


Figure 3-15: Performance evaluation using the PARSEC benchmark suite running on top of SMP Linux. Model parameters: 4 tiles, 8 in-order 10-stage cores, 64 KB 4-way associative L1 caches, 512 KB 4-way associative L2 cache, 4 GB DRAM

PARSEC benchmark suite on top of SMP Linux, and calculated the performance using counters built into the model. Figure 3-15 shows the speedup for the various benchmarks as the number of allocated cores increases from 1 to 8. For each benchmark, the speedup is normalized with respect to single-core performance. Ferret, which is very communication intensive, and Freqmine, which is parallelized with OpenMP, exhibit almost no speedup. The remaining benchmarks are parallelized using the Pthreads library, and scale between 4× to 8× from 1 to 8 cores. When all the 8 cores were allocated, the processor model was able to achieve a performance of 55 MIPS on average.

3.4 Related work

3.4.1 Software-based multicore simulators

Many software-based multicore simulators have been developed in recent years. Rsim is a discrete event-driven simulator written in C++ and C [25], and provides detailed models of out-of-order superscalar processors connected via coherent shared memory. It does not run an operating system and only models user-level activity of applications. Simics [26] is a popular commercial functional simulator which, on the other hand, can boot an operating system and run applications on top of it. Simics can be coupled with detailed execution-driven performance models like Gems [27], and M5 [28]. Gems and M5 provide accurate models of the memory hierarchy and the on-chip network for a multi-core system allowing detailed evaluation of these components. Garnet [29] is one such accurate model of the on-chip network which uses the Gems framework. COTSon [30] is another multicore simulator framework based on AMD's SimNow [31] which is a JIT-based dynamically-translating emulator. COTSon runs an operating system and applications on top of it. The MPARM SystemC framework [32] is a complete system-level simulator, and includes cycle-accurate cores, complex memory hierarchies and bus-based interconnection mechanisms. A linux port for MPARM is underway. BigSim [33] is another multi-core simulator which simulates a distributed memory as opposed to the shared memory model that we simulate. All of the above simulators are at least an order of magnitude slower than the FPGA-based Arete.

A recent multicore processor simulator called Graphite [34] targets systems with thousands of cores. It relaxes cycle-accuracy to attain a higher simulation speed ranging in tens of MIPS. Unlike Arete, Graphite is not a full system simulator, and it does not run an operating system. ZSim [35] is another recent effort aimed at simulating thousands of cores which improves on both the accuracy and speed of Graphite by combining instruction-driven timing models of the core with event-driven timing models of the memory subsystem.

3.4.2 FPGA-based processor simulations

FPGA-based performance modeling of multicore processor architectures was kick-started by the Research Accelerator for Multiple Processors (RAMP) project [36] in 2005. The focus of the project was to explore the role that FPGAs can play in accelerating computer architecture research. The RAMP project brought together many collaborators from both industry and academia with the common goal of sharing ideas, techniques and infrastructure for FPGA-based research.

In the early phase of the RAMP project, many teams implemented the RTL of various processor designs, developed for ASIC implementation, on FPGAs. The goal was to quickly come up with a large multicore design implemented across an array of FPGA chips. This effort brought the realization that the RTL developed for ASIC implementation is not very well suited to the FPGA fabric. The key idea that emerged from this experience was that model time should be separated from FPGA time in order to improve simulation speed and reduce resource utilization on FPGAs.

Our effort to build Arete was based on this idea, and so were RDL, FAST, ProtoFlex, HAsim and RAMP Gold. We first give an overview of these projects and contrast their modeling approach with ours. We then describe some of the other work in FPGA-based modeling of multicore processors.

RAMP Design Framework (RDF)

The goal of RDF [37] was to enable high-performance simulation and emulation of large-scale, massively-parallel systems on a wide variety of FPGA platforms, and to enable a large community of users to cooperate and build a useful library of inter-operable hardware models. RDF is designed to support a wide range of accuracy with respect to timing, from cycle-accurate simulations to purely functional emulations.

In RDF, a target model is a collection of loosely coupled units communicating using latency-insensitive protocols. All communication between units is via messages sent over unidirectional point-to-point FIFO channels. Channels are strictly typed with respect to the messages they can carry, but messages can be fragmented during

transmission, to enable flexibility in implementation. To model channel latency and bandwidth, credit-based flow control is used.

RDF channels require that the units in the target model communicate with each other using FIFO interfaces. Moreover, credit-based flow control is convenient when communicating units are separated by long latencies, but it is excessive when units are closely coupled.

ProtoFlex

The main idea behind ProtoFlex [7] from Chung *et al.* is to accelerate SMARTS-style simulations [1]. In this style of simulations only a few small samples extracted from a large benchmarking application are run on a detailed processor model. The remaining application is run on a functional simulator which does not keep track of the timing of the target processor design. Since the functional simulator executes orders of magnitude more application code than the detailed processor model, the functional simulator can become the performance bottleneck.

Chung *et al.* perform the functional simulation of multicore processors on FPGAs to obtain higher performance. Their FPGA-based functional model is then coupled with a timing model under SMARTS-style simulation.

One of the key observations in the ProtoFlex work is that there is a class of instructions whose implementation in the functional model is very inefficient, both in terms of FPGA resources and timing. These instructions, however, are very rarely executed. Chung *et al.* decided to split the functional model between FPGA and software. Whenever the FPGA-based functional model detects one of these instructions, it migrates the model state and execution to a host PC. Once the software-based functional simulation of the instruction completes, model state and execution transfers back to the FPGA. Although the migration between FPGA and software is quite expensive, since it occurs very infrequently, its impact on overall simulation speed is negligible.

Another contribution made in this work is time-multiplexing of the FPGA-based functional model. Chung *et al.* observed that when functional modeling migrates to

software, the FPGA-based functional model remains idle. In order to improve the utilization of their FPGA-based model, they decided to implement multithreading. They also observed that the functional model of the processor pipeline on FPGA could be greatly simplified if each pipeline stage executes a different thread. Thus eliminating the need for hazard detection and stall logic. They implemented a functional model which executes 16 threads simultaneously.

FPGA-Accelerated Simulation Technologies (FAST)

In FAST [9], Chiou *et al.* make the opposite placement decision for the functional and timing partitions of the simulator than ProtoFlex. FAST uses a software-based functional emulator to generate an instruction stream which is fed to an FPGA-based timing model that combines cycle-level timing information with the stream. Using this approach, Chiou *et al.* have also developed a high-performance multicore simulator [38].

The functional emulator in FAST is based on QEMU [39], and includes support for check-points and rollback. This allows the timing model to redirect the functional emulator whenever the instruction stream diverges from the correct execution path, as determined by the timing model. This approach differs from traditional software-based timing-directed simulators in that the long redirection latency between the FPGA and the host PC means that the functional emulator cannot stall for feedback from the timing model after executing every single instruction. Instead, FAST uses a speculative functional emulator which produces an instruction stream along a path that it predicts the timing model will take.

HAsim

HAsim [8] is a framework for building FPGA-based multicore performance models using a technique called A-Ports [4]. A-Ports are communication primitives that enable asynchronous modeling of synchronous systems using FIFO queues. A-Ports also include a variable-length shift register for modeling the latency of modules that comprise the synchronous system. This modeling technique was the source of inspiration

for the LI-BDN modeling methodology that we used to build Arete.

Using HAsim, Pellauer *et al.* developed a multicore simulator on FPGAs with detailed core and network models [40]. Making use of time multiplexing, the largest system that they have demonstrated on a single FPGA chip comprises of 16 cores, but the cache model lacks support for cache coherence. In a more recent attempt [15], Fleming *et al.* mapped HAsim to two FPGAs and scaled it to 128 cores. HAsim is also a functional-timing partitioned simulator, but unlike ProtoFlex and FAST, both partitions are accelerated on the FPGA substrate.

RAMP Gold

Tan *et al.* 's RAMP Gold [10] is also a partitioned, time-multiplexed simulator. Like HAsim, RAMP Gold places both the timing and functional partitions on the FPGA fabric. The aim of RAMP Gold was to study the scaling of the cache hierarchy in very large multicore processor designs. To accomplish this goal, Tan *et al.* implemented only one core on the FPGA in the functional partition. The core stalls only on cache misses and its design is thoroughly optimized for the FPGA fabric. In the timing partition of RAMP Gold, the functional core model is used to simulate a 64-core shared-memory processor architecture.

The limitations of this work include the inability to model core architectures which involve branch prediction or out-of-order execution. Although a detailed memory model is included in the timing partition, it does not model cache coherence. Cache coherence is ensured by sharing the level-1 caches among the 64 cores. The network model comprises of a magic crossbar.

Other FPGA-based multicore processors

Many projects made use of the MicroBlaze softcore, the MIPS softcore or the PowerPC hardcore found on FPGAs to build large multicore processors with detailed memory and network models. These projects include Liberty [41], RAMP Blue [42], ATLAS [43], Beehive [44], Heracles [45], Beefarm [46], and FPGA-based MPSoC emulation frameworks from Valle *et al.* [47] and Nava *et al.* [48]. We provide an

overview of a few of these projects below.

Liberty [41] is originally a software simulator designed for implementation on a parallel host by making use of barrier synchronization. Penry *et al.* 's work allowed the migration of the software threads of the simulator to a PowerPC hardcore on a Xilinx Virtex-IIPro FPGA. Additional logic was implemented around the PowerPC core to correctly integrate it with Liberty's parallel task scheduler, and to stall it when no thread was available. The thread executed much faster on the PowerPC core, and the approach demonstrated that large speedups could be gained from such a partitioning.

RAMP Blue [42] connected multiple BEE2 boards, each of which contains multiple FPGAs. Each FPGA was programmed to implement several MicroBlaze softcores. The MicroBlaze cores were connected with a network that supported both shared-memory and message-passing. Although RAMP Blue achieved several orders of magnitude higher performance than software simulators, it did not model a realistic target processor architecture.

ATLAS [43], also known as RAMP Red, is a multicore processor which was implemented on the BEE2 board and included support for hardware transactional memory. Similar to Liberty, ATLAS used the PowerPC hardcore found on some FPGA chips. The PowerPC core was augmented with a transactional memory component. The ATLAS project also demonstrated several orders of magnitude higher simulator performance compared to software.

The Beehive processor [44] is an experimental many-core computer implemented on a single FPGA on the XUPv5 board. The processor cores were based on a new RISC ISA which was designed to be easily understood and readily modifiable. The cores included private, split level-1 caches which lacked cache coherence. A ring network connected all the cores to each other and to main memory.

Discussion

Arete differs from earlier FPGA-based simulators in that Arete was developed using a cycle-level specification of the target processor design. This specification was trans-

formed into FPGA-optimized RTL using the LI-BDN technique which makes Arete cycle-accurate by construction. Moreover, we maintain a clear distinction between target simplifications and implementation refinements described in Section 2.2.3.

3.5 Summary

We have presented a fast and cycle-accurate simulator for a multicore PowerPC architecture. The simulator accurately models a shared memory subsystem which includes a cache coherence engine. We are able to run off-the-shelf SMP Linux along with several applications. We have also ported the simulator to several FPGA platforms with both single and multiple FPGAs. The simulator is highly parameterized and modular, and we have demonstrated its flexibility by performing various architectural experiments with moderate effort.

We employed some novel ideas to provide a user-friendly simulation infrastructure.

(I) A distributed debugging environment using the LI-BDN technique enables us to independently freeze any module in any model cycle. We provide its details in Chapter 4.

(II) Functionally-identical partitions and a distributed protocol for assigning identifiers makes it possible to use one configuration file for all the FPGAs in a multi-FPGA platform.

FPGA-based modeling has come a long way in the past few years. Although it offers substantially higher simulation speed than software, a few key issues have prevented its widespread adoption for architectural research. We have addressed these issues in the design and development of Arete.

(I) Programmability: FPGAs are typically programmed in low-level RTL languages like Verilog or VHDL. Designing a large and complex system in RTL requires a tremendous effort. Moreover, these designs are very inflexible for architectural exploration. These issues are mitigated by the use of a high-level specification language and BSV.

(II) Resource management: Unlike software simulators, FPGA-based simulators have hard resource constraints. To meet these constraints, one has to either time-multiplex the limited resources or map the system to multiple FPGAs. Both approaches can result in a loss of efficiency if the cycle-by-cycle timing behavior of the implemented design has to be preserved. The LI-BDN technique provides a much more efficient solution because it decouples implementation from specification, and only preserves the timing behavior of the specification.

(III) Interfacing with off-chip memory or host PC: These interfaces tend to be quite complicated and ill-documented. We have minimized this problem by wrapping these low-level interfaces with split-transaction (send/receive) interfaces. We have done this to port Arete to the three FPGA boards that are being commonly used for academic research.

Chapter 4

Deterministic, Model-Cycle-Level Debugging of Synchronous Systems Modeled Asynchronously[1]

4.1 Introduction

As designs of digital systems continue to become more complex, designers are increasingly adopting FPGAs for both performance modeling and rapid prototyping. The FPGA fabric allows designers to exploit the inherent parallelism in these systems, and delivers a tremendous performance improvement over software. As mentioned in Chapter refchap:Spec, this adoption comes at a price. Parts of the target system being modeled or prototyped often do not map well to the structures in the FPGA fabric, in terms of both resources and timing. And the solution to the problem is to implement the synchronous behavior of the target system in an asynchronous manner on the FPGA, decoupling the model cycles from the FPGA cycles.

Debugging is an integral part of the design effort. A comprehensive debugging infrastructure needs to provide model-cycle-level access to all the pertinent state in the system. Providing such low-level access is not straight-forward in such asynchronous

[1]The work presented in this chapter includes contributions from Muralidaran Vijayaraghavan.

implementations of synchronous systems as described above. Taking a snapshot of the state in a certain FPGA cycle is possible, but the snapshot may contain values of state elements from different model cycles. Either the lagging modules have to be advanced or the hastening modules have to be rolled back in order for the state snapshot to reconcile to a particular model cycle.

At a high level, a designer should be able to issue a `stop(modelCycle n, state S)` command, which will freeze the entire system in model cycle `n` and provide the values of all the state elements included in vector `S`. A `start(state S)` command will also be needed to resume the operation of the asynchronous implementation with the state elements initialized to the values specified in vector `S`.

Parallel systems with inherent non-determinism, such as multicore processors running parallel applications, offer yet another challenge for debugging. A large body of work [49, 50, 51, 52] exists that strives to achieve deterministic execution. To circumvent the non-determinism in the system, these solutions have to keep a log of all the non-deterministic events, the performance and resource overheads of which can be prohibitive.

In this chapter we present 1) a technique for building a deterministic model-cycle-level debugging infrastructure, based on the LI-BDN modeling methodology, and 2) an application of the technique to build a comprehensive debugging infrastructure for Arete [3], which is an FPGA-based multicore processor simulator.

We show that the debugging infrastructure in Arete provides a rich set of features, while incurring small resource and performance overheads. It allows for stopping and starting any module in the processor model independently by making a novel use of the provisions of the LI-BDN methodology, and avoids complex forwarding and rollback mechanisms. It also allows us to remove the non-determinism from events such as DRAM access, network access and I/O, without keeping expensive logs.

Chapter organization: Section 4.2 presents the various debugging techniques used in FPGA-based models and prototypes. Section 4.3 describes how deterministic model-cycle-level debugging can be implemented using LI-BDNs. Section 4.4 discusses the debugging infrastructure in Arete, a multicore processor simulator, and

Debugging support	Clock control type	Deterministic execution support	Resource-performance overhead
Scan chains	None	No	Substantial
SCE-MI-based	FPGA	No	Substantial
ISA-based	None	No	Moderate
Asynchronous models	Model[a]	Possible	Substantial
LI-BDN-based	Model	Yes	Minimal

[a]requires a forwarding or rollback mechanism

Figure 4-1: Summary of the comparison between the LI-BDN-based debugging technique and other common debugging techniques used in FPGA-based designs

presents statistics on its resource and performance overheads. Section 4.5 provides a summary of our work.

4.2 Survey of debugging techniques for FPGA-based designs

In this section we discuss some of the common debugging techniques used in FPGA-based designs. Figure 4-1 provides a summary of the comparison between these techniques and the technique based on the LI-BDN methodology that we present in this chapter.

4.2.1 System monitoring through scan chains

System monitoring solutions based on scan chains [53, 54, 55, 56] are perhaps the most widely used tools for debugging FPGA-based designs. They integrate logic analyzers and other test and measurement cores with the target design on FPGA. A remote graphical user interface communicates with these cores, and provides the designer with a logic analyzing solution.

In ChipScope [53], for example, the designer generates integrated logic analyzer (ILA) cores for all the modules in his design that he wishes to monitor. The ILA cores are customizable and include logic for detecting trigger events. They also include logic

for capturing and storing data using on-chip Block RAMs. An integrated controller (ICON) core is then used to provide communication between all the ILA cores and the software running on a host PC. The communication takes place over the JTAG boundary scan port of the FPGA. The ILA cores and the ICON core can be integrated into the design at either the HDL-source-code-level or the synthesized-netlist-level.

Although these tools provide some very useful features for debugging synchronous designs, they lack control over both the FPGA and the model clocks. Moreover, the monitoring cores are synchronous and use the FPGA clock. To use these tools for debugging synchronous designs implemented in an asynchronous manner, the designer would have to develop forwarding and rollback mechanisms to be able to construct model time accurately. He would also have to develop some means of operating the monitoring cores using the model clock. These tools do not include support for deterministic execution, and the cost of comprehensively monitoring large designs may be prohibitive.

UltraSOC [57] and ARM CoreSight [58] are similar debugging tools targeted towards SoCs.

4.2.2 SCE-MI-based emulation environment

An emulation environment based on the SCE-MI standard, such as Bluespec emVM [59], comprises of an FPGA configured with a hardware design and a host PC running the emulation console. The FPGA and the host PC are connected by a physical link such as PCIe, ethernet, RS-232, *etc.* The emulation console communicates with the components of the hardware design through implementation-independent transactors. These transactors allow the designer to start and stop the FPGA clock. They also allow control over the hardware design in the form of reset and various testing and debugging tasks. Probing functionality such as waveform viewing is also often provided.

Although the provided set of monitoring and debugging features is not as extensive as that in ChipScope, SCE-MI based emulation environments provide the ability to freeze the entire synchronous design in any FPGA cycle. However, for debugging

synchronous designs implemented in an asynchronous manner, the designer would face the same set of challenges as he would with ChipScope. A lack of support for deterministic execution and substantial resource and performance overheads also limit the appeal of such tools.

4.2.3 ISA-based debugging

When using FPGAs for modeling processors, designers also have the option of implementing the debugging facilities prescribed in the ISA. These facilities enable debugging functions, such as reset, instruction and data breakpoints, and single-stepping of programs. They generally consist of debug control and status registers, address and data value comparison registers, and a debug interrupt. Whenever a debug event takes place, it raises a debug exception (if enabled by setting the appropriate bits in the control register). A debug interrupt handler routine is then invoked which performs the appropriate debug operation.

ISAs also include instructions, such as Debugger Notify Halt (DNH) in the Power ISA [18], which cause the processor to stop fetching and executing instructions, and allow the processor to be managed by an external debugging facility. Such a facility is allowed to access processor resources and control its execution.

Although ISA prescribed debugging facilities provide fine-grained control over the processor's resources, they may be quite difficult to implement. For instance, implementing a precise debug interrupt in an out-of-order processor can be quite cumbersome.

4.2.4 Debugging in various asynchronous FPGA-based models

Various FPGA-based simulators that rely on asynchronous modeling of synchronous designs, such as ProtoFlex [7], UT-FAST [9], RAMP Gold [10] and HAsim [8], implement debugging facilities in an ad-hoc manner. They need to implement forwarding and rollback mechanisms in order to achieve model-cycle-level debugging. In

ProtoFlex, printf-like statements are added to the generated RTL to provide monitoring during software simulation of the RTL. The Connectors in UT-FAST include support for triggering, logging of traces and user-specified aggregation. RAMP Gold embeds a microcode injector into the functional processor pipeline for debugging and simulation control. HAsim provides a distributed mechanism for model-cycle-level control which involves waiting for all the A-Ports to become balanced, at which point all the modules are in the same model cycle.

4.3 Debugging using the LI-BDN technique

The major requirement for debugging a large and complex model is to have the ability to freeze it in a particular model cycle so that a precise snapshot of all the state can be obtained. This requirement gets quite tricky when the synchronous specification of a target design is decoupled from its platform-specific implementation. A designer typically requires the values of the state during a particular model cycle as opposed to the implementation cycle. Even if the entire implementation is frozen during a particular implementation cycle, various asynchronous modules in the implementation have to either rollback or advance so that the entire design converges to a particular model cycle. Such an ability is similar to taking a snapshot of the architectural state of an out-of-order processor for precise exceptions.

We present a novel technique, based on the LI-BDN theory, for freezing an asynchronous implementation of a synchronous design during a particular model cycle. The technique does not involve forwarding or rollback of modules. Instead, we make use of the property that a model cycle of an LI-BDN module completes only when all the outputs have been enqueued, all the inputs are available and have been dequeued, and all the state elements have been updated.

As shown in Figure 4-2, we introduce a new input, `proceed`, to the LI-BDN register file module from Figure 2-7(b). This new input does not alter the specification of the register file module in any way as it is completely ignored. We also add debugging logic, and `debugReq` and `debugResp` FIFOs to the module. An external debugger can

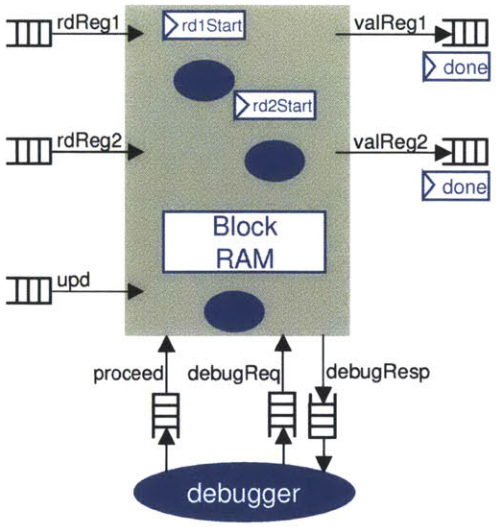


Figure 4-2: LI-BDN register file module with support for model-cycle-level debugging

freeze the module in model cycle n by enqueuing a *Normal* token $n-1$ times into the `proceed` input FIFO and enqueuing a *Debug* token the n^{th} time. Once the module receives a *Debug* token, it enters the debug mode and waits for debug commands that are sent through the `debugReq` FIFO. The debug commands can either read or update the Block RAM. Responses for Block RAM read requests become available one cycle after the request is made, and are sent back to the external debugger through the `debugResp` FIFO. When the external debugger sends a *Finish* command, the module leaves the debug mode, updates the Block RAM, dequeues all the LI-BDN input FIFOs, resets all the done flags and proceeds onto the next model cycle. The highlighted code in Figure 4-3 shows the debugging logic added to the LI-BDN register file module. Only the parts that deal with `rf` are specific to the register file module, the rest can be added to any LI-BDN module for debugging.

Although sending a token to every LI-BDN module on every model cycle is expensive, because very few modules contain state that needs to be accessed for debugging, the area overhead of our debugging technique remains quite modest, as we will show in Section 4.4. Moreover, since the debugging facility is not on the critical path, its performance overhead is negligible.

```
libdn regFile
{
  LiBdnIn rdReg1, rdReg2, upd, proceed;
  LiBdnOut valReg1, valReg2;
  FifoIn debugReq;
  FifoOut debugResp;
  BlockRAM entries[ sizeRF ] rf ( initial 0 );
  Reg rd1Start, rd2Start ( initial False );

  rule rd1
  {
    if( !valReg1.done && !valReg1.full && !rdReg1.empty
        && !upd.empty && !rd1Start )
    {
      rf.req1( Read, rdReg1.first, DontCare );
      rd1Start <= True;
    }

    if( rd1Start )
    {
      valReg1.enq( upd.first.valid && rdReg1.first == upd.first.idx ?
                   upd.first.val : rf.resp1 );
      valReg1.done <= True;
      rd1Start <= False;
    }
  }

  rule rd2
  {
    if( !valReg2.done && !valReg2.full && !rdReg2.empty &&
        !upd.empty && !rd2Start )
    {
      rf.req2( Read, rdReg2.first, DontCare );
      rd2Start <= True;
    }

    if( rd2Start )
    {
      valReg2.enq( upd.first.valid && rdReg2.first == upd.first.idx ?
                   upd.first.val : rf.resp2 );
      valReg2.done <= True;
      rd2Start <= False;
    }
  }
```

Figure 4-3: FPGA-optimized LI-BDN register file module with support for debugging

```
  rule finish
  {
    if( valReg1.done && valReg2.done && !proceed.empty )
    {
      if( proceed.first == Normal )
      {
        if( upd.first.valid )
        {
          rf.req1( Write, upd.first.index, upd.first.val );
        }
        rdReg1.deq; rdReg2.deq;
        upd.deq;
        proceed.deq;
        valReg1.done <= False; valReg2.done <= False;
      }

      else if( !debugReq.empty )
      {
        if( debugReq.first.type == Read && !debugResp.full )
        {
          if( !rd1Start )
          {
            rf.req1( Read, debugReq.first.index, DontCare );
            rd1Start <= True;
          }
          else
          {
            debugResp.enq( rf.resp1 );
            rd1Start <= False;
          }
        }
        else if( debugReq.first.type == Write )
        {
          rf.req1( Write, debugReq.first.index, debugReq.first.val );
        }
        else if( debugReq.first.type == Finish )
        {
          if( upd.first.valid )
          {
            rf.req1( Write, upd.first.index, upd.first.val );
          }
          rdReg1.deq; rdReg2.deq;
          upd.deq;
          proceed.deq;
          valReg1.done <= False; valReg2.done <= False;
        }
        debugReq.deq;
      }
    }
  }
}
```

Figure 4-3: FPGA-optimized LI-BDN register file module with support for debugging (cont.)

4.3.1 Correctness of the LI-BDN-based debugging technique

An LI-BDN module obtained through the transformation discussed in Section 2.4 has the following properties.

1. It simulates a model cycle by first producing all the outputs once (in an order determined by the availability of the inputs), followed by the firing of the `finish` rule.

2. It can take multiple implementation cycles to produce an output or to fire the `finish` rule.

3. The output rules can fire concurrently, but cannot fire in parallel with the `finish` rule. The `finish` rule acts as a barrier and prevents output rules from refiring before the model cycle is completed.

4. The LI-BDN transformation of any synchronous specification is fully automated, and we assume that it is correct.

We establish the correctness of the LI-BDN-based debugging technique through the following arguments.

1. The additional input, `proceed`, introduced for debugging, only affects the firing of the `finish` rule. The `finish` rule waits for a `proceed` token to arrive before firing, which can result in a prolonged model cycle. The latency of an LI-BDN module, *i.e.,* the number of implementation cycles consumed to simulate a model cycle, can be varied without affecting the correctness of the module.

2. The external debugger has to enqueue a `proceed` token, either *Normal* or *Debug*, for every model cycle. This ensures that every model cycle completes and forward-progress is made.

3. `proceed` is added to every LI-BDN module containing model state which needs to be monitored for debugging. Even though communicating LI-BDN modules may consume different number of implementation cycles to simulate the same

model cycle, `proceed` in a particular module remains independent of others, and it is consumed when the finish rule in its module is fired.

4. The additional FIFOs, `debugReq` and `debugResp`, are *out-of-band* communication links that remain outside the scope of the LI-BDN. The debug commands delivered by the `debugReq` FIFO are only serviced when the module is in the debug mode, which can only be activated after all the outputs are enqueued, but before the model state is updated. The LI-BDN resumes normal operation upon leaving the debug mode.

5. The debugging logic does not introduce any new behaviors into the target design being modeled. It only allows for reading and writing of model state when the LI-BDN module is in the debug mode.

Debugging logic, as in the case of the register file example above, can be introduced into an LI-BDN module such that it remains completely disjoint from the LI-BDN control logic in the module. This is evident from the highlighted code in Figure 4-3.

4.3.2 Deterministic execution

There are many sources of non-determinism in complex, parallel systems such as the randomness of the DRAM access latency. This complicates the debugging further by prohibiting deterministic replays. The use of LI-BDNs in modeling provides an opportunity to suppress the non-determinism. In the case of DRAMs, the access latency can be fixed to any desired value. This is possible because the LI-BDN can utilize different numbers of FPGA cycles to simulate different model cycles, and accommodate the randomness appropriately. The enqueuing of the output FIFOs (or the dequeuing of the input FIFOs) can happen when the non-deterministic event has taken place.

As an example, we will show how a DRAM module with a non-deterministic read latency is converted into an LI-BDN module with a deterministic read latency, *viz*, a combinational read. Figure 4-4 presents the synchronous specification of a memory

```
module memory
{
  Input req;
  Output resp;
  DRAM dram;

  every clock cycle
  {
    dram.req = req;
    resp = dram.resp;
  }
}
```

Figure 4-4: Synchronous specification of a DRAM module with non-deterministic read latency

module that uses a DRAM. Both `req` and `resp` have associated valid bits. The system which uses this module makes `req` valid for only one cycle, consumes `resp` in the cycle in which `resp` is valid, and does not make another valid `req` until it receives a valid `resp`. `dram` has a non-deterministic response time, and produces a response, `dram.resp`, which is valid for only one cycle.

Figure 4-5 presents the LI-BDN module of the non-deterministic memory module shown in Figure 4-4. It increments model time irrespective of the validity of `dram.resp`. `tempResp` ensures that a valid `dram.resp` is not dropped. In this case, every model cycle is simulated in two implementation cycles.

Figure 4-6 presents the LI-BDN memory module with combinational reads. If `req` is valid in a model cycle, the LI-BDN module sends it to `dram`, and waits, without incrementing the model cycle, until `dram.resp` becomes valid. When `dram.resp` becomes valid, the LI-BDN enqueues it into `resp` and completes the model cycle by dequeuing `req`. This LI-BDN module may consume a varying number of implementation cycles, depending on the `dram` latency, to simulate difference model cycles. Even though `dram` can take a non-deterministic number of FPGA cycles to produce a valid response, model cycles are incremented deterministically leading to deterministic execution.

The two LI-BDN modules presented in Figures 4-5 and 4-6 model different memory modules, but both fulfill the NED and SC requirements, and are deadlock-free.

```
libdn memory
{
  LiBdnIn req;
  LiBdnOut resp;
  DRAM dram;
  Reg tempResp ( initial Invalid );

  rule tempRule
  {
    if( !tempResp.valid )
    {
      tempResp <= dram.resp;
    }
  }

  rule respRule
  {
    if( !resp.done && !resp.full )
    {
      resp.enq( tempResp );
      resp.done <= True;
      if( tempResp.valid )
      {
        tempResp <= Invalid;
      }
    }
  }

  rule finish
  {
    if( resp.done && !req.empty )
    {
      dram.req = req.first;
      req.deq;
      resp.done <= False;
    }
  }
}
```

Figure 4-5: LI-BDN DRAM module with non-deterministic read latency

```
libdn memory
{
  LiBdnIn req;
  LiBdnOut resp;
  DRAM dram;
  Reg start ( initial False );

  rule respRule
  {
    if( !resp.done && !resp.full && !req.empty && !start )
    {
      if( req.first.valid )
      {
        dram.req = req.first;
        start <= True;
      }
      else
      {
        resp.enq( Invalid );
        resp.done <= True;
      }
    }
    if( dram.resp.valid )
    {
      resp.enq( dram.resp );
      resp.done <= True;
      start <= False;
    }
  }

  rule finish
  {
    if( resp.done )
    {
      req.deq;
      resp.done <= False;
    }
  }
}
```

Figure 4-6: LI-BDN DRAM module with combinational reads

4.4 LI-BDN-based debugging infrastructure for a multicore processor model: A case study

Using the LI-BDN-based debugging methodology described in Section 4.3, we built a comprehensive debugging facility for Arete [3], which is an FPGA-based cycle-accurate multicore simulator. Arete may be implemented as a distributed multicore simulator on a multi-FPGA platform, which requires the debugging infrastructure to be implemented in a distributed manner. We make use of the tiled microarchitecture of the processor to partition the model among various FPGAs in such a way that only one configuration file can be used for all the FPGAs. This enables a simple replication of the debugging facilities, but complicates the design of the controller.

The distributed debugging facilities in FPGA are controlled by a software running on a MicroBlaze soft core. The MicroBlaze core communicates with the model through the PLB. The software presents a GDB-like interface to the user. Its features include model initialization, break points, single-stepping, access to processor state such as program counter, general purpose registers (GPRs), special purpose registers (SPRs), TLB array, and data and tag arrays in caches, and access to performance counters which include model cycles, FPGA cycles, instructions, stalls due to data and control hazards, and cache hits and misses. Figure 4-7 presents a screen shot of the debugging capabilities provided by the debugging software developed for Arete.

Figure 4-8 shows the debugging facilities incorporated into a core in Arete. These FPGA-based facilities include logic and state for

- model initialization, which may be done in a distributed manner on a multi-FPGA platform, and requires assigning a unique identifier to each of the identical model partitions,
- distribution and accumulation of debugging and performance information from various tiles, cores and modules,
- instruction address, data address and model cycle comparisons for freezing Arete in a particular model cycle.

```
The following commands can be issued at any time
    freeze
The following commands operate only when the system is paused
    help
    exit
    resume
    getResume
    getPc
    setPc <addr>
    setTlb <index> <data>
    getTlb
    getEpochs
    getModelCycles
    setModelCycles <n>
    getHostCycles
    setHostCycles <n>
    getReg <index>
    setReg <index> <data>
    getMas
    getTb
    getInstCount
    setInstCount <n>
    getPrivInstCount
    getMispreds
    getExceps
    getWrngPaths
    getRAWStalls
    getIStats
    getDStats
    getL2Stats
    getBreakAddr
    setBreakAddr <addr>
    removeBreakAddr
    getBreakCount
    setBreakCount <n>
    removeBreakCount
    step <n>
    setTID
    setCore0
    set2Cores
    set4Cores
    set8Cores
    getMem <addr>
    setMem <addr> <data>
```

Figure 4-7: A screen shot of the debugging capabilities provided by the debugging software developed for Arete

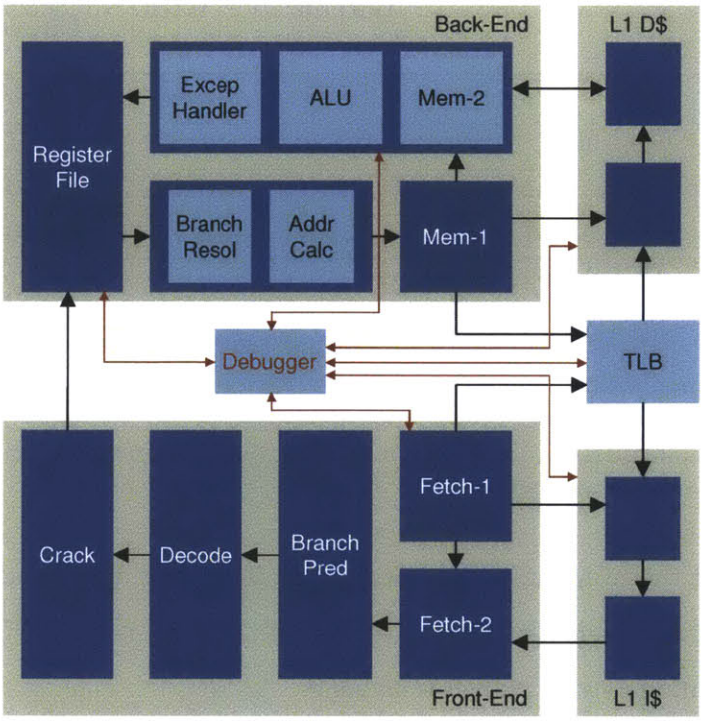


Figure 4-8: Arete core with model-cycle-level debugging facilities

The use of the LI-BDN modeling methodology in building Arete also enables us to provide deterministic execution of parallel applications on the multicore processor model. We make the observation that the three sources of non-determinism in Arete are memory, on-chip network, and external inputs. We transform the DRAM along with the memory controller, and the on-chip network (implemented in a distributed manner on multiple FPGAs) into LI-BDN modules. This allows us to fix their latencies in the manner described in Section 4.3.2. We deal with external inputs by freezing the model whenever the program expects such an input. This ensures that the external input is always received in the same model cycle. Both of these techniques have very low resource and performance overheads, and help to avoid keeping expensive logs of non-deterministic events.

Figure 4-9 shows the minimal overhead of including the deterministic model-cycle-level debugging facility in Arete. It causes an increases of 5% in resource utilization, and reduces FPGA clock frequency by 6%. As described in the register file example in Section 4.3, the debugging facility requires limited additional state and logic

	Without debugging	With debugging	Change
LUTs	61155	64154	+5%
Flip flops	49359	51331	+4%
Block RAMs	111	111	0%
DSP slices	24	24	0%
FPGA clock frequency (MHz)	125	117	−6%

Figure 4-9: Resource and performance penalties of the debugging infrastructure in Arete. Model parameters: 1 tile, 2 in-order 10-stage cores, 64 KB 4-way associative L1, 512 KB 4-way associative L2

resources, and the overhead is expected to scale linearly with model size. Moreover, the debugging facility has no impact on the average number of FPGA cycles required to simulate a model cycle, which remains 9.

4.5 Summary

In this chapter we presented a debugging technique based on the LI-BDN modeling methodology. The technique facilitates deterministic model-cycle-level debugging, while avoiding both forwarding or rollback mechanisms for model-cycle-level control, and logging of non-deterministic events for deterministic replay. We used the technique to build the debugging infrastructure for Arete, which is an FPGA-based cycle-accurate multicore simulator. The debugging infrastructure provides a rich set of features, while incurring small resource and performance overheads.

Part II

Architectural Exploration Using Cycle-Accurate Simulation

Chapter 5

Impact of Modeling Abstractions on the Accuracy of Single-Core Processor Simulations

5.1 Introduction

Suppose we want to evaluate three different branch prediction schemes in the context of an in-order processor pipeline. A more sophisticated prediction scheme may provide a higher rate of instructions per cycle (IPC), but at the cost of some chip area and power consumption. So a proper evaluation requires a quantitative cost-benefit analysis. One also has to realize at the onset that any such study is constrained by time and resources. For example a company may assign two engineers and give them three to six months to conduct the study. In this chapter we focus on quantitatively estimating the benefit of each branch predictor and ignore the question of cost.

Such studies are typically performed by running a suite of benchmark programs using software simulators of the proposed architecture. Often a simulator for one of the closely related machines is available and the experimenter repeatedly modifies it to incorporate architectural features to be studied. Some of the perennial questions in any such study are:

1. Is the simulator detailed and accurate enough for what one wants to study? For example, for studying branch predictors one must be able to study the effect of instructions executed on the mispredicted path.

2. Is the simulator flexible enough so that it can be modified to study each alternative design? For example, for our study the simulator must be modifiable to incorporate any of the three branch predictors.

3. Is the simulator fast enough so that the benchmarks of interest can be run to completion in a reasonable amount of time?

4. Does the simulator have the capacity (*e.g.,* memory) to run the benchmarks on the data sets of interest?

Some published architectural studies are conducted using simulators that abstract away many details of the cycle-accurate models. Though many researchers have pointed out the dangers of *unvalidated* simulation models (see, for example [60, 61, 62, 63, 64, 65]), validation of simulation results against real machines or cycle-accurate models is quite uncommon in published literature.

In this chapter, we quantitatively evaluate the accuracy of two abstract architectural models used for estimating the performance of three branch prediction schemes. We modified our base cycle-accurate simulator, Arete [3], for three different branch predictors. These are 1. an always not-taken predictor (the ANT scheme); 2. a 2-bit branch direction predictor (the BHT scheme); and 3. a branch predictor with a branch target buffer in addition to the direction predictor (the BTB scheme). The architectural abstractions we studied were a) one where the memory hierarchy is replaced by a one-level memory combined with a statistical model parameterized by the estimated number of cache misses (the AbsM model); and b) one where the back-end (execute part) of the processor pipeline is replaced by a single stage and a statistical model is used to inject stalls due to data hazards (the AbsME model).

As architects we would expect the BTB scheme to perform better than the BHT scheme and the BHT scheme to perform better than the ANT scheme, though it would

be difficult to guess by how much. Similarly, we would expect the AbsM model to be more accurate than the AbsME model, provided we can find the right parameters to plug into the abstract models. Can we estimate the cache miss rates and pipeline stalls for each of the architectures without its cycle-accurate model? Do we expect the cache miss rates or pipeline stalls to be different for each branch predictor? The answers to these questions depend upon the behavior of the instructions executed on the mispredicted path. For meaningful conclusions to be drawn from our architectural study, the errors in predictions using abstract models should be significantly less than the quantitative differences predicted by our abstract model studies.

Our study shows that the AbsM model was highly accurate and captured the impact of changing the branch predictors correctly, both quantitatively and qualitatively. However, when the execution pipeline abstraction was added (the AbsME model), the accuracy of the model dropped considerably, so much so that one may conclude there was no significant advantage of the BHT scheme over the ANT scheme. The parameters used in the abstract models (like the cache hit rate, the number of stalls in the pipeline due to RAW hazards, etc.) have a big impact on the accuracy of the abstract models. Finally, we argue that to validate abstract architectural models, it is essential to build cycle-accurate models and it is practical to do so.

Chapter Organization: Section 5.2 discusses some of the related work. Section 5.3 discusses the memory and execute/stall abstractions that we employed in our abstract models in greater detail. It also examines the accuracy of the abstract models with respect to the cycle-accurate models. Section 5.4 provides a summary of our findings.

5.2 Related work

There is a substantial body of work to improve the speed of software simulators without compromising the accuracy of performance estimates. Yi *et al.* discuss various simulation methodologies that have been developed to reduce the number of instructions of a benchmark that a simulator needs to execute to predict the performance for the full run of the benchmark [63]. The three common techniques are: 1) re-

duced input-set simulation where a smaller but supposedly representative input set is used for the benchmarks, 2) truncated execution where the execution is stopped after running a fixed number of instructions, and finally 3) the sampling techniques where performance is estimated by running randomly or periodically sampled instructions on the benchmarks. Their paper concludes that the sampling technique is both the fastest and the most accurate. Our studies confirm the validity of the sampling technique.

Papers [60, 61, 62] describe how abstractions in simulators can reduce the accuracy of the performance estimates. Desikan *et al.* [60] compared the Sim-Alpha simulator, which is an out-of-order simulator implemented according to the specification of Alpha 21264 microarchitecture, against a real Compaq DS-10L workstation. They recommend the use of microbenchmarks to calibrate the simulators against real machines. To avoid errors due to incorrect parameters, they recommend using cache hit ratio, etc. from published documents or from real machines. Our studies confirm that the choice of parameters affects the accuracy of predictions made by the abstract models substantially.

Cain *et al.* [61] again argue the need for high-precision (*i.e.,* non-abstract) models and actual workloads in order to correctly predict performance of real designs. They show that OS and I/O effects drastically impact the accuracy of performance predictions. But Cain *et al.* also assert that simulating instructions on the mispredicted paths is largely unimportant for performance predictions. One cannot take this assertion literally if the goal is to study various branch-prediction schemes. Indeed our results show that inaccurate modeling of the behavior of mispredicted instructions is the main reason for the decrease in accuracy of abstract models.

Bose *et al.* [62] argue that detailed simulation is expensive and not plausible (This was probably a true assertion given the simulation technology of the time when their paper was published). Instead they say that one should build abstract simulators and these simulators must be calibrated against existing real machines by running microbenchmarks that target specific portions of the machines such as cache behaviors, loop executions, etc. Again the issue with this approach is that

the new machine to be designed is going to be different from any existing machine thus invalidating the calibration parameters obtained from the existing machines. We show that even the slightest variation in parameters (obtained from a different but close enough architecture) can cause huge inaccuracies in simulation.

Black *et al.* [64] give an instructive overview of simulation techniques. They identify the three basic kinds of errors: modeling errors (the errors in simulator code), specification errors (the specification of the target architecture is erroneous resulting in an erroneous simulator) and abstraction errors (errors that creep in because of modeling a system with insufficient detail). They describe the design process used by microarchitects, where they start out with a crude simulation model and systematically refine it, adding more features and details, and fixing bugs in the process, in order to create a detailed simulator of the target architecture. Our work supports most of the assertions made by Black *et al.* . They also claim that model refinement does not improve the accuracy of the model monotonically which is confirmed by our study.

Similar to Black *et al.* , Skadron *et al.* [65] give an overview of the state of affairs in simulation technologies and provide recommendations on how computer architecture evaluation techniques can be improved. They argue that designing benchmarks, both micro and macro, is paramount for accurately predicting the performance of the system being designed. They also claim that analytical modeling techniques are not very well studied and that these will become important given the trend in computer architecture to move towards multi-core architectures which will further slow down the already slow simulators. They also say that the impact of abstractions on the accuracy of performance projections is not well understood and several studies have to be done in order to classify specific abstractions as good or bad.

We argue that detailed cycle-accurate simulations are necessary to validate abstract models. Our solution is to build cycle-accurate models on FPGAs. This not only enables validation of abstract models but also offers huge gains in simulation speeds of cycle-accurate models. The technology for building cycle-accurate simulators on FPGAs is improving and the library of components for building such simu-

lators is growing. We believe that in the next few years it will become as easy to build cycle-accurate simulators on FPGAs as it is to build cycle-accurate software simulators.

5.3 Comparison of branch predictors using cycle-accurate and abstract models

We evaluated the three branch prediction schemes (ANT, BHT and BTB) for the in-order PowerPC pipeline by building three architectural variants of the cycle-accurate processor model (the ACC models) on the XUPv5 FPGA platform with the following configuration.

Tiles	1×
Cores	1×, in-order, 10-stage PowerPC
L1 I-cache	1×, private, 64 KB, 4-way set-associative, 64B blocks, 1 cycle pipelined hit latency
L1 D-cache	1×, private, 64 KB, 4-way set-associative, 64B blocks, 1 cycle pipelined hit latency
L2 cache	1×, shared, inclusive, 512 KB, 4-way set-associative, 64B blocks, 32 cycle pipelined hit latency
Main memory	1×, 512 MB, 256 cycle latency

Figure 5-1 shows the IPCs for each of these models while running an off-the-shelf 32-bit Linux kernel and 5 benchmarks selected from the SPECINT2000 suite. The ANT scheme is used as the baseline. These statistics show that the BHT scheme provides a 9% to 21% improvement in IPC over the ANT scheme while the BTB scheme provides an additional 1% to 15% improvement. This is the type of performance that we would expect if we actually built these machines.

In the rest of this section we want to study how close we could have come to this conclusion using abstract models. To reiterate, the reasons for considering abstract models are that they may be easier to implement and may run faster, especially

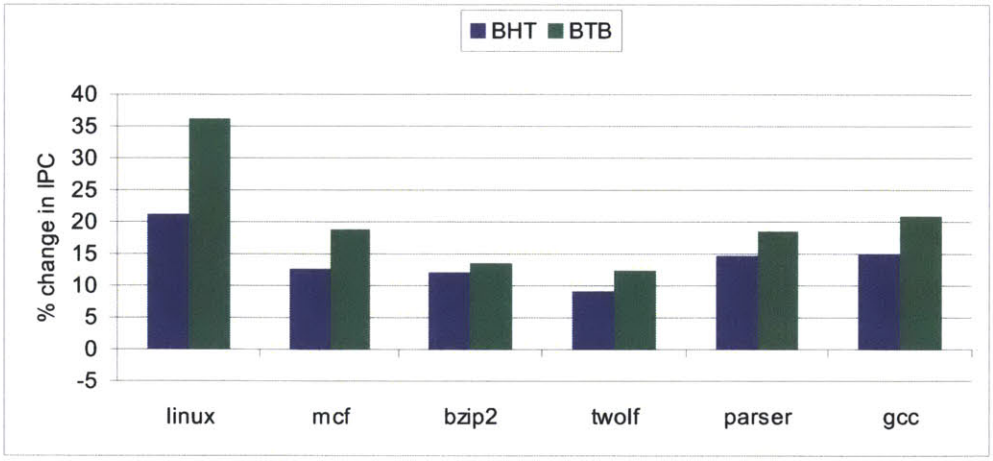


Figure 5-1: Effect of different branch prediction schemes on IPC, obtained from the ACC models. Baseline scheme is ANT.

in software. For each of the two abstract models, we will first provide its detailed description and then exercise it using parameters obtained from the cycle-accurate models. Finally, we will quantify the error in results obtained from it. At the end of the section we will discuss a study based on sampled execution of benchmarks.

5.3.1 Model with memory abstraction (AbsM)

In this model all cache accesses are serviced directly by a flat memory model but a certain percentage of accesses are treated as cache misses and charged a longer latency. We divide the overall number of cache requests (at both L1 and L2) into chunks of 1000, and we treat x of these requests, chosen randomly, as misses, where x is the average number of cache misses per one thousand requests obtained from the cycle-accurate model. For those requests which are treated as cache misses, the corresponding responses are not supplied until cycles equal to the appropriate cache latency have passed. We justify the use of this abstraction through the following argument.

We are modeling an in-order pipeline, with blocking caches. If the target machine and the abstract model fetch the same instruction stream (including both correctly and incorrectly predicted instructions), then it does not matter which instructions are

penalized for cache misses (including both data and instruction cache misses). This is because the cache miss latency simply gets added to the overall number of cycles required to run an application. This assumption holds true only if both the abstract model and the actual target specification always fetch the same instruction stream. It fails when a stall in the pipeline due to a data cache miss delays the update of the branch predictor resulting in the abstract model fetching a different instruction from that fetched by the target specification. We believe this to be a second-order effect, which should not have much impact on the accuracy of the abstract model.

Obtaining parameter values

When using abstract models, one of the challenges is to obtain accurate values for the various parameters. Typically, these values are obtained from a real processor. They are then tweaked to match the performance of the abstract model with that of the real processor. When the abstract model is used to carry out an architectural study, these parameter values may negatively impact the accuracy of the study, because they may have exhibited variation if the study were performed on the real processor (which is not possible). However, if the study is performed on a cycle-accurate model, accurate parameter values can be obtained from each architectural variation of the model.

For our experiment we use three sets of parameter values; the first two representing the typical case, and the last representing the accurate case.

1. We assume that the cycle-accurate model with the BHT scheme is a real processor, and the parameter values we obtain from it are used in all the architectural variations of the abstract model.

2. Next, we assume that the cycle-accurate model with the ANT scheme is a real processor, and the parameter values we obtain from it are used in all the architectural variations of the abstract model.

3. Finally, we obtain the parameter values from each of the three variations of the cycle-accurate model, and use them in the corresponding variation of the

abstract model.

Comparison of results

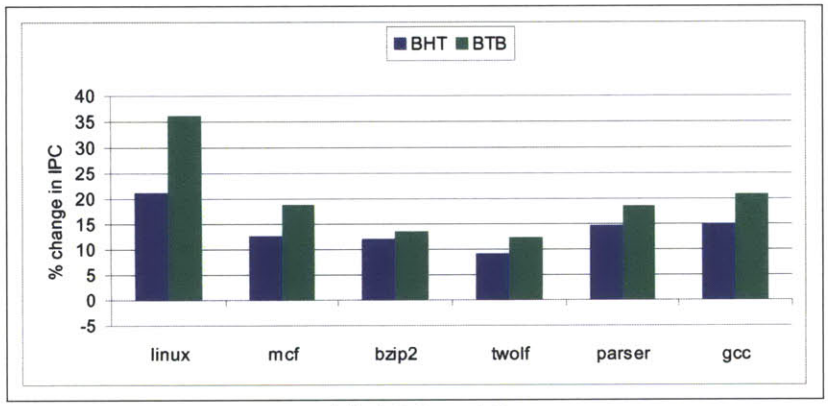
We now compare the performance statistics obtained from the AbsM model against those obtained from the cycle-accurate model. Figure 5-2 shows the change in IPC demonstrated by AbsM for each of the three sets of parameter values. All three graphs are quite similar, and we can conclude that the increase in IPC after adding the BHT is quite substantial compared to the increase in IPC after adding the BTB. This observation matches quite well with that obtained from the cycle-accurate model.

Figure 5-3 provides the quantitative error in IPC values obtained from the AbsM model when compared against those obtained from the cycle-accurate model. We see that the memory abstraction is fairly accurate and the accuracy does not vary much with different sets of abstraction parameters.

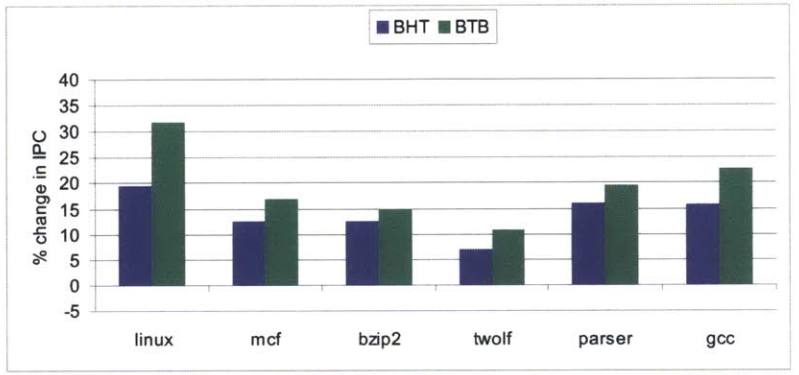
5.3.2 Model with memory and execution abstractions (AbsME)

In case of the AbsME model, on top of the memory abstraction described above, we replaced the back-end of the pipeline with a single execute stage. We also added a bypass stage in front of this execute stage in order to correctly model the latency between the misprediction of a branch and its resolution (and hence, the update of the branch prediction tables). For this abstraction we divide the overall number of cycles required to run an application into chunks of 1000, and we treat y of these cycles, chosen randomly, as stall cycles, where y is the average number of stall cycles due to RAW hazards per one thousand execution cycles obtained from the cycle-accurate model. Every time the abstract model chooses to stall, the instruction remains in the bypass stage, and no progress is made.

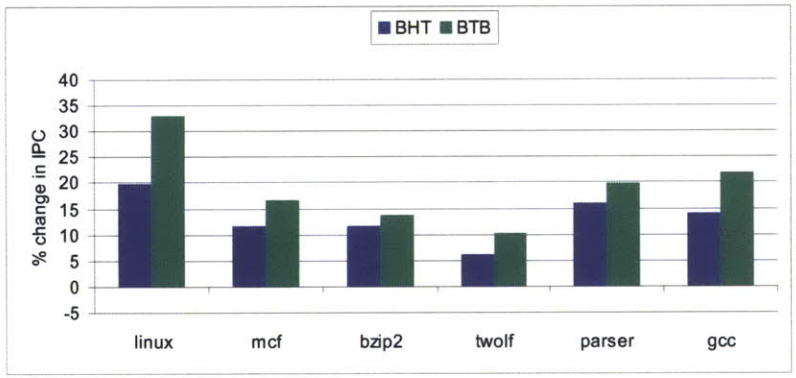
The justification for this abstraction is similar to that for the memory abstraction. If the target machine and the abstract model fetch the same instruction stream (including both correctly and incorrectly predicted instructions), then it again does not



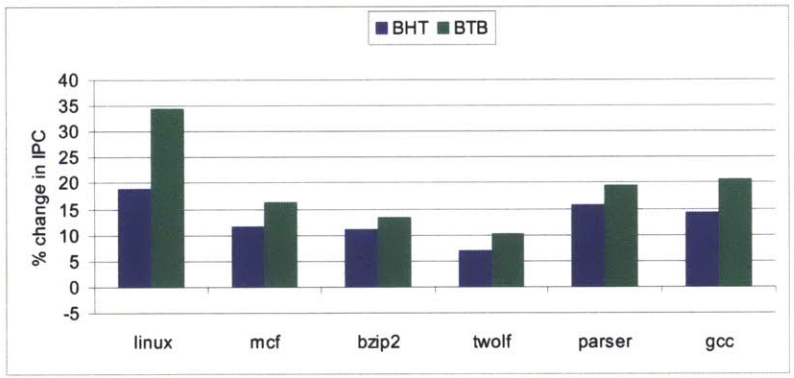
(a) ACC model



(b) AbsM model with ACC-BHT parameters

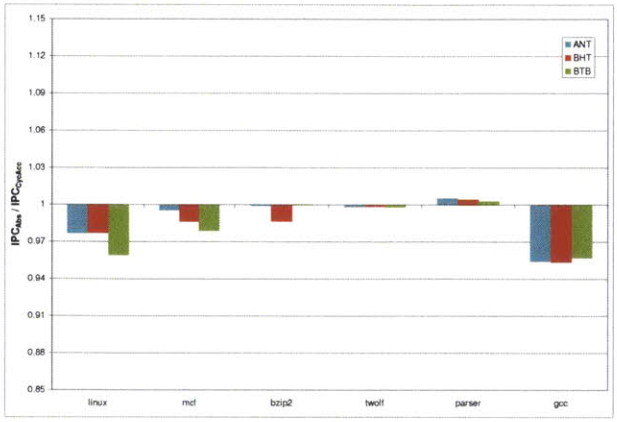


(c) AbsM model with ACC-ANT parameters

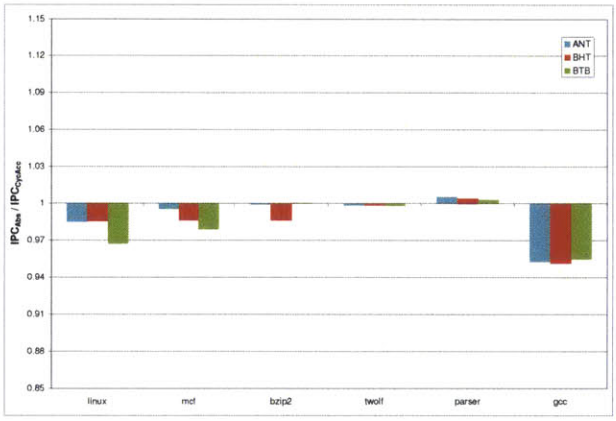


(d) AbsM model with corresponding ACC parameters

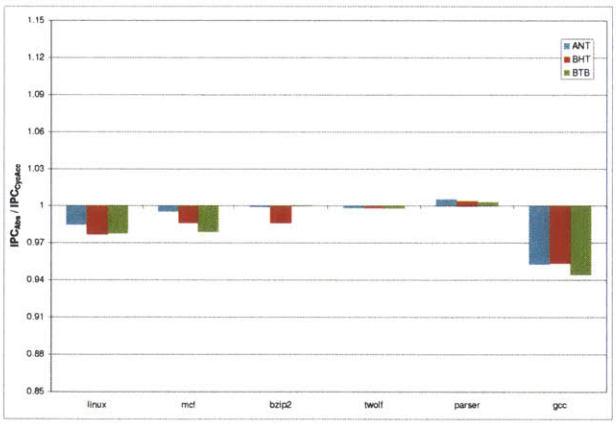
Figure 5-2: Effect of different branch prediction schemes on IPC, obtained from the AbsM models. Graph in (a) is from the ACC model, added for ease of comparison.



(a) ACC-BHT parameters



(b) ACC-ANT parameters



(c) Corresponding ACC parameters

Figure 5-3: Error in IPC obtained from the AbsM models with different abstraction parameters, with respect to the corresponding cycle-accurate models

matter which instructions are penalized for stalling. Again, this assumption breaks down when a mispredicted instruction gets resolved at a different time, resulting in the update of the branch predictor at a different time. We consider this to be a secondary effect as well.

In order to study the effects of different branch prediction schemes through the AbsME model, we required the number of stalls due to RAW hazards in addition to the miss rates of all the caches. We used the same three sets of parameter values that we did in the case of the AbsM model.

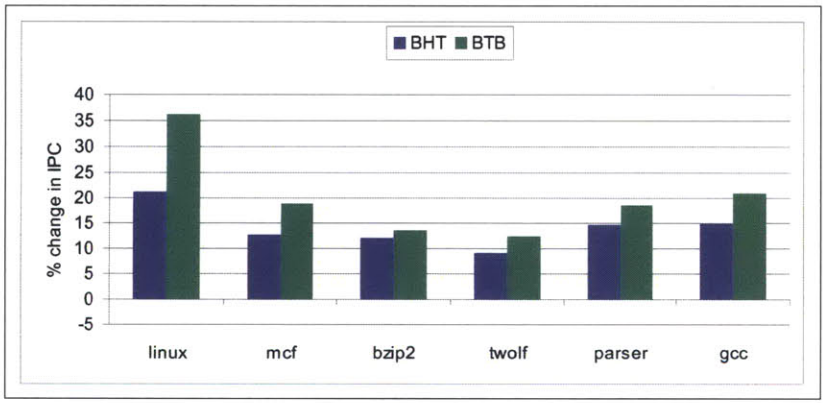
Comparison of results

The three graphs in Figure 5-4 show the change in IPC demonstrated by the AbsME model for the three sets of parameter values. The graph in Figure 5-4(a) shows that when we have both the memory and the execution abstractions in the model, and we use the parameters obtained from the cycle-accurate model with the BHT scheme, adding the BHT has negligible impact on IPC, but adding the BTB increases IPC quite substantially. This result is totally different from the ones obtained from both the cycle-accurate model and the AbsM model.

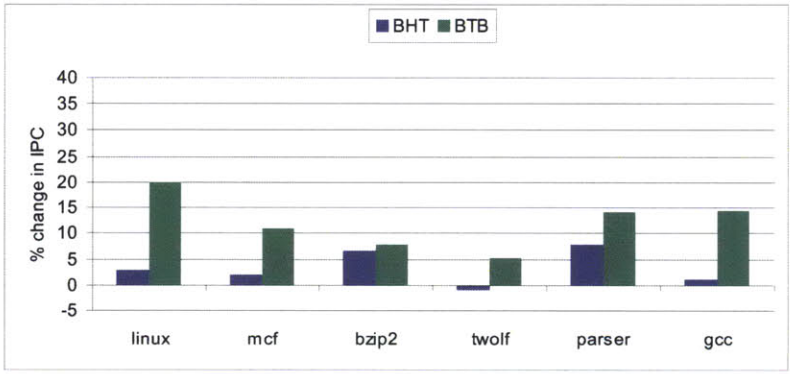
Figure 5-4(b) demonstrates the bizarre nature of abstractions and parameters. When we use the parameters obtained from the cycle-accurate model with the ANT scheme, the variations in IPC obtained from the AbsME model match quite closely with those obtained from the cycle-accurate model.

The use of two different sets of abstraction parameters in the AbsME model had led us to two conflicting observations, albeit one of the observations was quite accurate. To determine whether the inaccuracy was present in the abstraction, the abstraction parameters or both, we decided to use the most accurate set of parameter values. We can see from Figure 5-4(c) that the AbsME model behaves quite differently. However, the difference is not as large as in Figure 5-4(a).

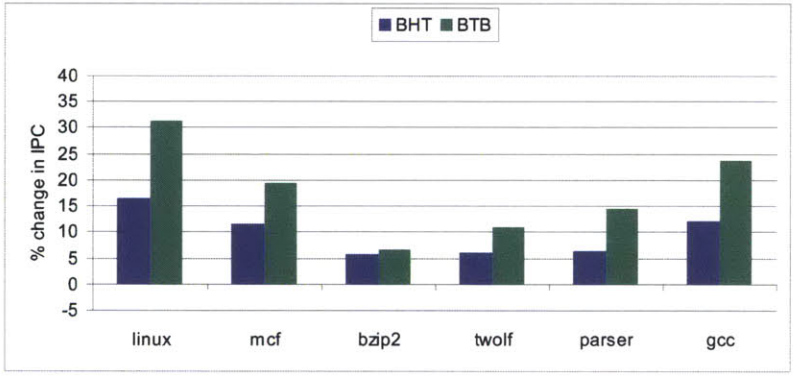
Figure 5-5 provides the quantitative error in IPC values obtained from the AbsME model when compared against those obtained from the cycle-accurate model. We see that the variation in accuracy is quite large when the memory abstraction is coupled



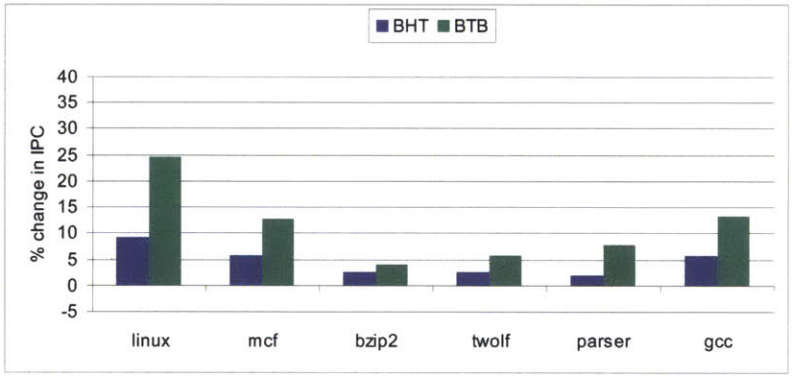
(a) ACC model



(b) AbsME model with ACC-BHT parameters

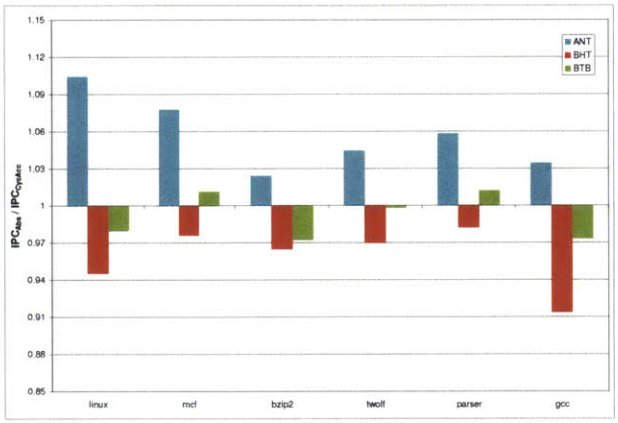


(c) AbsME model with ACC-ANT parameters

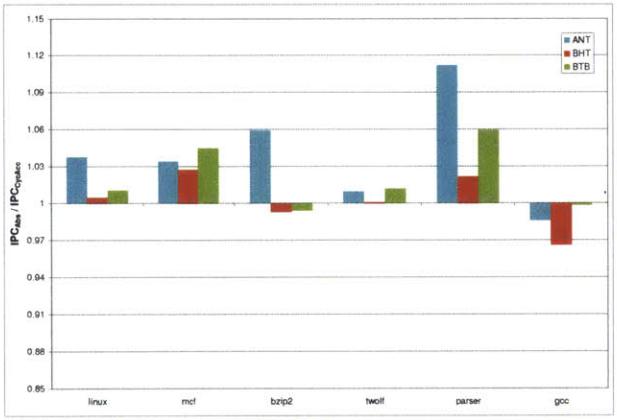


(d) AbsME model with corresponding ACC parameters

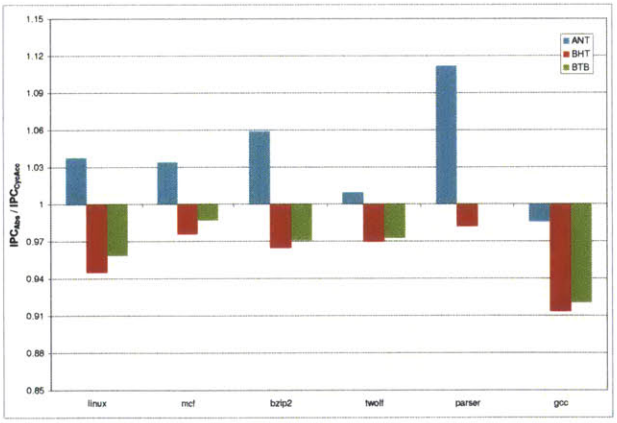
Figure 5-4: Effect of different branch prediction schemes on IPC, obtained from the AbsME models. Graph in (a) is from the ACC model, added for ease of comparison.



(a) ACC-BHT parameters



(b) ACC-ANT parameters



(c) Corresponding ACC parameters

Figure 5-5: Error in IPC obtained from the AbsME models with different abstraction parameters, with respect to the corresponding cycle-accurate models

with the execution abstraction, which results in the varying observations we made in Figure 5-4.

5.3.3 Sampled execution of benchmarks

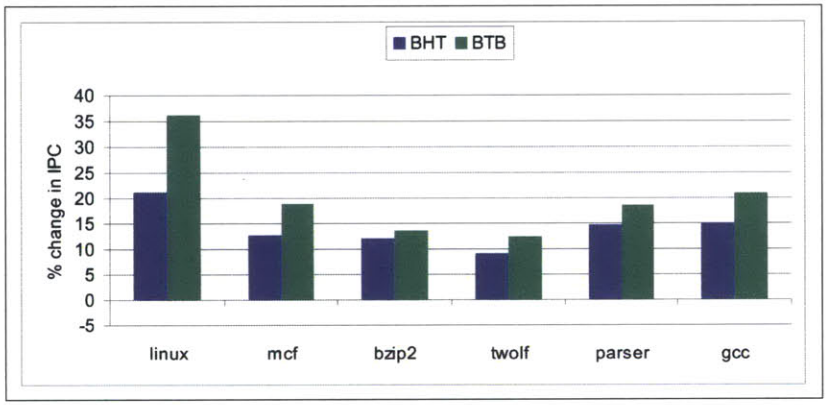
We are able to run the cycle-accurate model of a single-core processor at the rate of 6-10 MIPS on the XUPv5 platform. This enables us to run large workloads with tens of billions of instructions within a few hours. For our final experiment we considered how sampled execution, a technique commonly used in software simulations to quickly obtain performance statistics, would affect the accuracy of the simulator. We obtained performance statistics from the execution of 10 million instructions at different intervals during the execution of a multi-billion instruction program: once at the start, then after 1 billion instructions, and finally after 2 billion instructions.

In Figure 5-6, we can see the variations in IPC obtained from these sampled executions. We see that while the first sampled execution provides quite different observations from the complete execution (Figure 5-1), the accuracy improves with the second and third sampled executions.

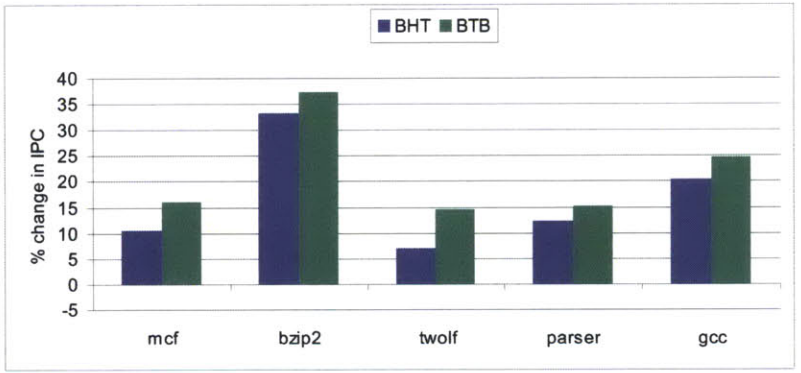
5.4 Summary

We can study various branch prediction schemes in isolation using synthetic stimuli, to determine which provides the lowest misprediction rate. Studying the impact of a branch prediction scheme on processor performance, however, is quite challenging, and requires detailed, full-system modeling. In this chapter we explored if certain parts of the model can be abstracted to ease the model development effort and to increase the simulation speed. From our study, we have reached several conclusions:

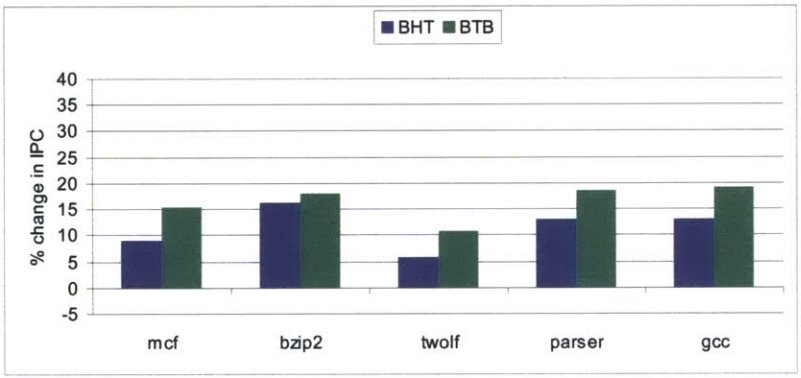
Firstly, even if we can justify the abstraction of individual components, the cumulative effects of having abstractions for several components simultaneously, drastically decreases the overall accuracy of the abstract model. We experienced this with the memory and the execution pipeline abstractions. Just having the memory abstraction did not reduce the accuracy of the abstract model, but having the execution pipeline



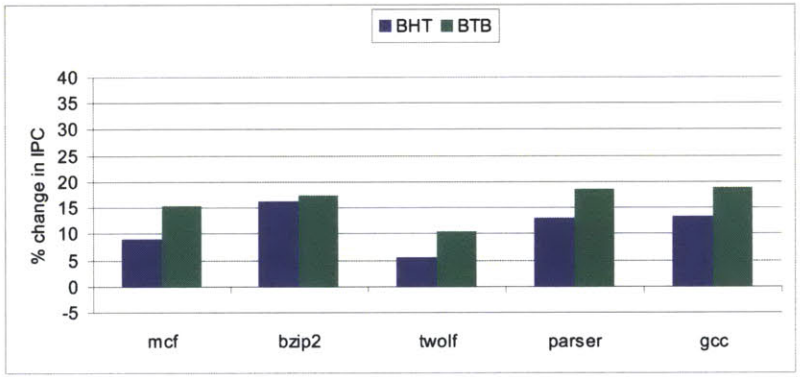
(a) ACC model



(b) First 10 million instructions



(c) After 1 billion instructions



(d) After 2 billion instructions

Figure 5-6: Effect of different branch prediction schemes on IPC obtained from the cycle-accurate models using sampled execution. Graph in (a) is from full execution, added for ease of comparison.

abstraction, in addition to the memory abstraction, drastically reduced the accuracy of the model.

Secondly, the relative performance predictions of different microarchitectures based on the abstract models are completely off from the relative performance in the real machines, quantitatively and sometimes even qualitatively. For example, from Figure 5-4(b), one may conclude that removing the direction predictor (BHT) will have minimal impact on performance. Moreover, the error in the abstraction is in the same range as the performance improvement.

Another interesting aspect in abstract models is the choice of parameters related to the abstraction. Since the correct parameters cannot be known a priori, these parameters must be approximated for abstract models. In our studies, we saw that the accuracy of abstract models for different microarchitectures is highly sensitive to the parameters used.

Finally, we conclude that in order to validate any of the abstractions, we must use a real machine or a cycle-accurate model of the real machine.

Chapter 6

Impact of Simplified Core Models on the Accuracy of Multicore Processor Simulations

6.1 Introduction

While carrying out architectural exploration in the memory hierarchy or the interconnect network of multicore processors, specially those with hundreds or thousands of cores, it is desirable to use coarse-grained or simplified core models, such as the 1IPC core model (which stalls only on cache misses). This helps to lower the simulator development time and improve simulator performance, albeit at the cost of simulation accuracy.

A survey of the proceedings of the three major computer architecture conferences held in the year 2012 reveals that published simulation studies in memory and network of multicore processors use a wide variety of core models. Figure 6-1 presents a classification of the core models used in these publications. We see that most studies use event-driven or execution-driven core models with varying degree of accuracy. However, a few, particularly those that focus on processors with hundreds or thousands of cores, rely on 1IPC core models. Fewer still, particularly those that focus on

Conference	Synthetic traffic	1IPC core model	Event-driven core model[a]	Execution-driven core model[a]
HPCA '12	3	4	5	7
ISCA '12	2	3	3	14
MICRO '12	3	5	5	11

[a]varying degree of accuracy, better than 1IPC

Figure 6-1: Publications with various core models in full-system simulators used to study the memory hierarchy or the interconnect network

on-chip networks with hundreds of nodes, use synthetic traffic generators.

The use of these simplified core models in simulation studies carried out to estimate not only memory and network performance, but also overall system performance, points to the underlying belief that although results obtained from such studies may not be quantitatively accurate, they can be used as qualitative predictions of performance trends. In this chapter we challenge this notion by providing evidence that there is substantial quantitative and qualitative error in such studies, which can lead to wrong conclusions.

We use a full-system simulator with cycle-accurate models of core, memory and network to perform studies in memory and network. We consider this simulator as representative of a real machine, and use memory, network and overall performance results obtained from it as gold standard. We replace the cycle-accurate core model in the simulator with a 1IPC core model, and perform the studies again. Results obtained from these simulations fail to capture any performance trends and have a mean error of 59%. This is clear evidence that 1IPC cores simply cannot be used in simulation studies for estimating performance without a thorough validation effort.

We analyze the error in results obtained using the 1IPC core model, and point out the architectural as well as software phenomena behind it. We then systematically add more details to 1IPC in order to improve simulation accuracy. Through this validation effort, we show that by using a core which does not model pipeline stalls due to data hazards, but accurately models speculative instructions, we can reduce mean error by 6×, and capture every performance trend.

To determine the scaling of error with the number of cores, we perform our simulation studies using 2-core, 4-core and 8-core processor models. Although our processor simulations are modest in size, our results clearly show that error magnitude increases with the number of cores.

6.2 An experiment in the memory subsystem

For the memory experiment, we considered four cache line replacement policies in the L2 cache.

1. random (used as the baseline)
2. LRU (least recently used)
3. MRU (most recently used)
4. LNS (least number of sharers)

We would expect the more sophisticated replacement policies, which are better able to retain cache lines that are likely to be accessed again, to perform better. We evaluated each policy in terms of

(a) cache hit rate,

(b) coherence traffic between L1 and L2 caches, referred to as cache traffic,

(c) coherence traffic between L2 caches and main memory, referred to as memory traffic, and

(d) overall system performance, measured in terms of execution time.

6.2.1 Experimental setup

To conduct our evaluation we used Arete [3], an FPGA-based cycle-accurate full-system simulator. We modeled a tiled multicore processor architecture, in which each of the identical tiles comprised of multiple cores with private split L1 instruction and

data caches, a shared and inclusive L2 cache, a cache coherence engine and a network router. Cache coherence was implemented using a hierarchical directory-based MSI protocol. Tiles communicated with each other using a fully-connected network.

We modeled a multicore processor on the BEE3 board [24], where each FPGA chip was programmed to simulate one tile of the processor. The model was implemented with the configuration provided below.

Tiles	4×
Cores	8×, in-order PowerPC
L1 I-cache	8×, private, 64 KB, 4-way set-associative, 64B blocks, 2 cycle pipelined hit latency
L1 D-cache	8×, private, 64 KB, 4-way set-associative, 64B blocks, 2 cycle pipelined hit latency
L2 cache	4×, shared, inclusive, 1 MB, 8-way set-associative, 64B blocks, 16 cycle pipelined hit latency
Main memory	4×, distributed, shared, 1GB, (256 cycle + network traversal time) latency
Network	16-bit channel width, 6 cycle hop latency

We booted off-the-shelf SMP Linux and ran a mix of PARSEC [16] and SPLASH-2 [17] benchmarks. Each application was run to completion with an average of approximately 100 billion instructions. We achieved an average system throughput of approximately 55 MIPS.

Using the debugging facility described in [6], we were able to freeze the entire system to create precise checkpoints, and accurately capture all the system state.

To isolate the impact of core model behavior on performance results, we eliminated all kinds of variability from our simulation studies. We used highly detailed cycle-accurate models of memory and network with fixed memory and hop latencies, respectively. We ran applications using an automated script which was launched immediately after the Linux boot completed. The only user input to the system was the push of the start button. Thus ensuring that every run of a particular application produced the same results, accurate to the cycle.

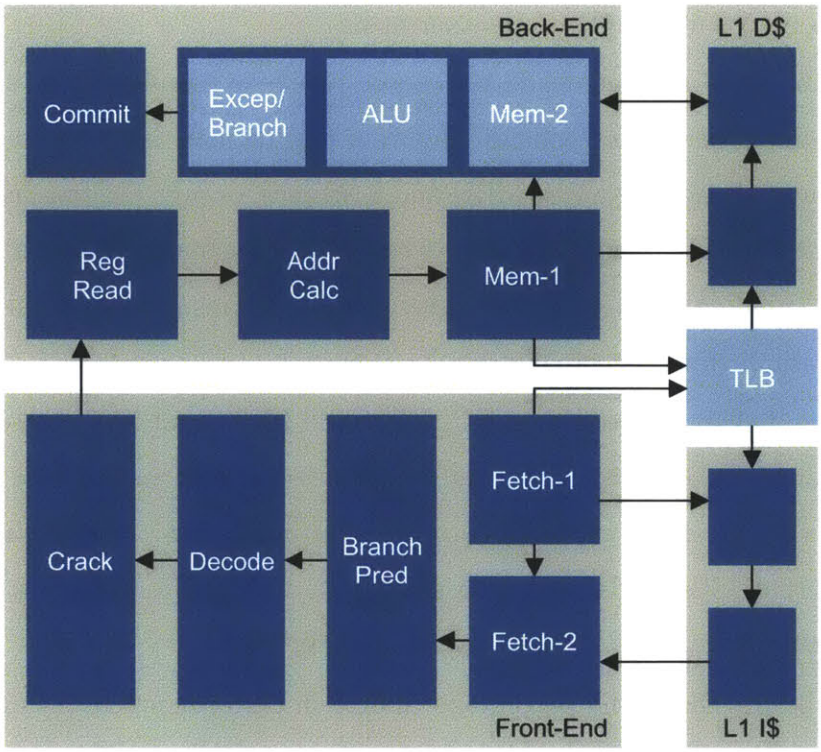


Figure 6-2: Accurate core model

To gain confidence in our results and to ensure that they were not skewed, we also performed a variability study. We ran each application multiple times and for each run varied the memory latency and the scheduling of requests in the L2 cache.

6.3 Memory experiment using the cycle-accurate core model

We began our evaluation with the cycle-accurate core model, referred to as ACC, and shown in Figure 6-2. It comprises of an in-order, 10-stage processor pipeline with split L1 caches and a shared TLB. The front-end of the processor pipeline fetches instructions, predicts branches, decodes instructions, and breaks down complex loads and stores. The back-end reads the register file, calculates memory addresses, executes instructions, handles branch mispredictions and exceptions/interrupts, and updates the register file. In Figure 6-2, dark blue blocks represent pipeline stages, while light

blue blocks represent functional components.

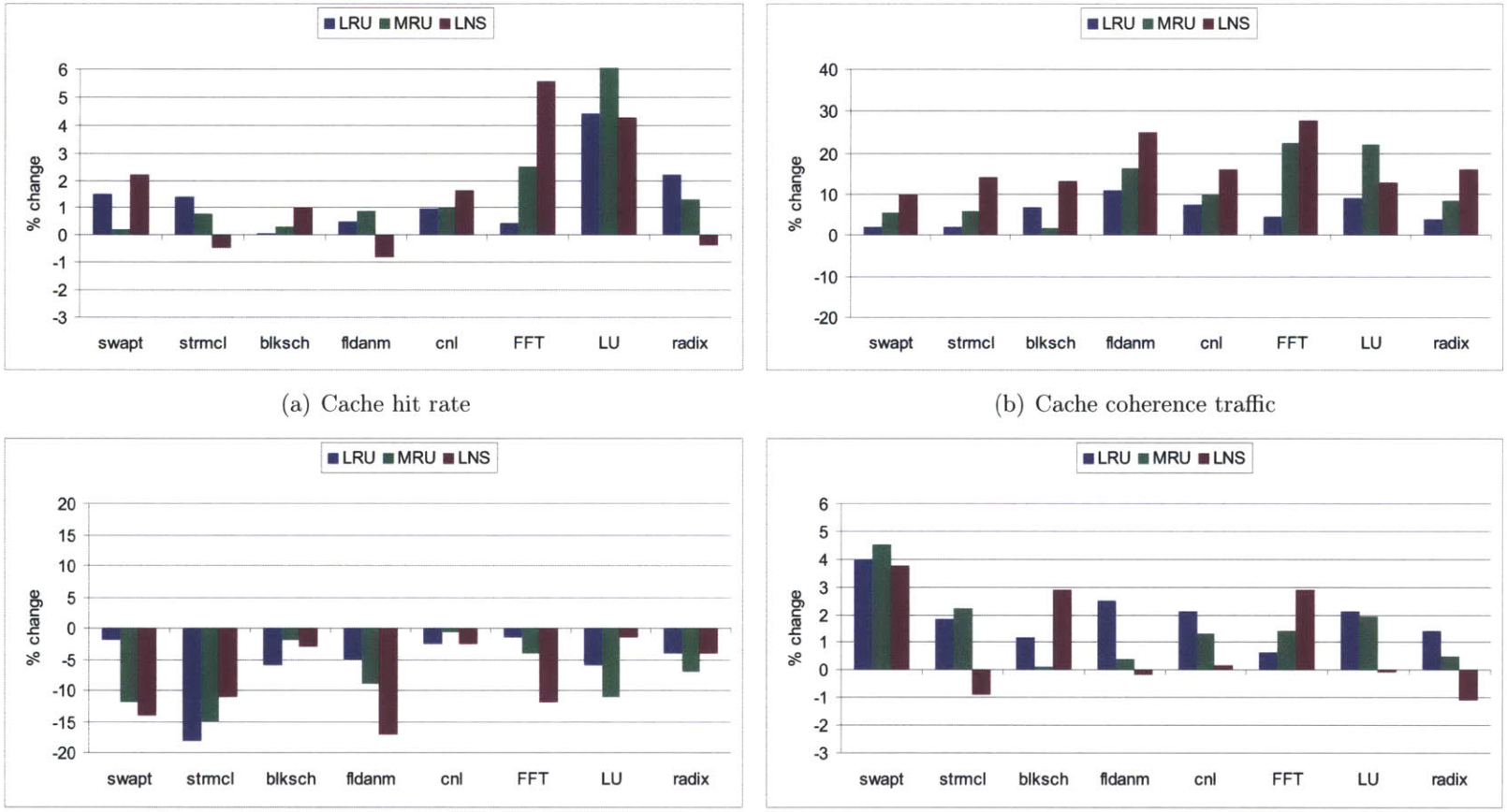
Using the ACC core model in the 8-core processor simulator, we evaluated the four cache line replacement policies. Figure 6-3(a) depicts the percentage change in cache hit rate when using LRU, MRU and LNS policies over the random policy. We observe that both LRU and MRU always result in an increase in cache hit rate, albeit not a very substantial one. On the other hand, LNS results in a small decrease in cache hit rate in three out of eight applications.

In Figure 6-3(b), we see that LRU, MRU and LNS result in an increase in cache traffic, with LNS resulting in the highest increase, except in the case of LU. This increase in cache traffic is expected because of the increase in cache hit rate. Figure 6-3(c) shows that LRU, MRU and LNS result in a decrease in memory traffic. This is again expected because of the increase in cache hit rate.

Figure 6-3(d) shows the percentage change in performance resulting from LRU, MRU and LNS over random. We see that both LRU and MRU provide a small increase in performance, while LNS results in a small decrease in half of the applications. Although the variations in cache and memory traffic are quite significant, we see only a small variation in performance. This can be explained by the under utilization of the available network bandwidth, which remains below 5% for all the applications that we considered for this study.

6.4 Memory experiment using the 1IPC core model

Having obtained accurate results, we replaced the ACC core model with the 1IPC core model in the simulator. Figure 6-4 shows the 1IPC core model. It comprises of a single-stage processor pipeline which only models stalls due to cache misses. Speculative instructions due to branch mispredictions and exceptions/interrupts, and data hazards are not modeled. Instructions which do not suffer a cache miss, are executed in one cycle.



(a) Cache hit rate

(b) Cache coherence traffic

(c) Memory coherence traffic

(d) Performance

Figure 6-3: Impact of LRU, MRU and LNS replacement policies obtained from ACC. Baseline replacement policy is random.

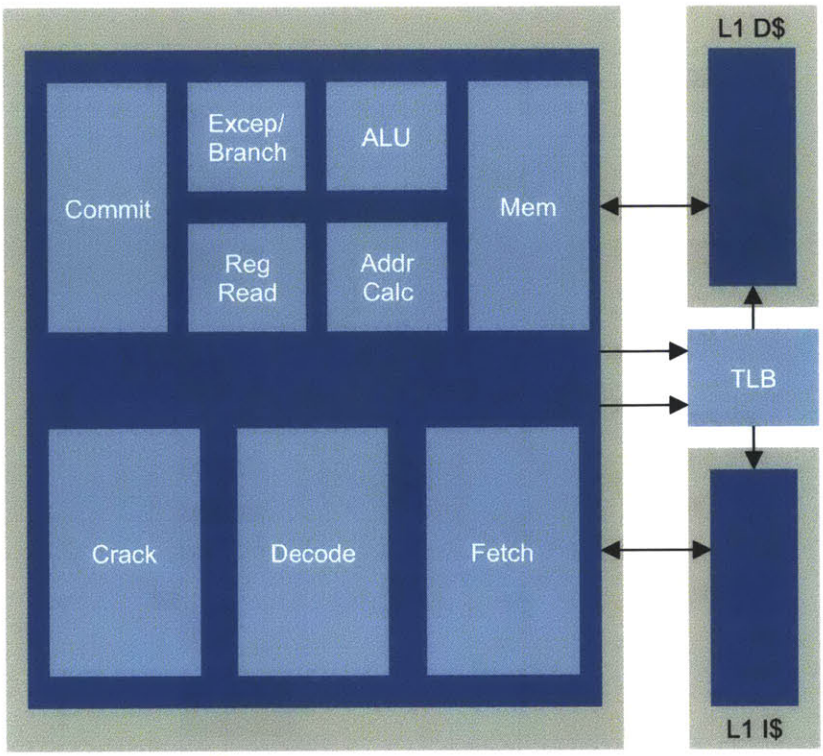
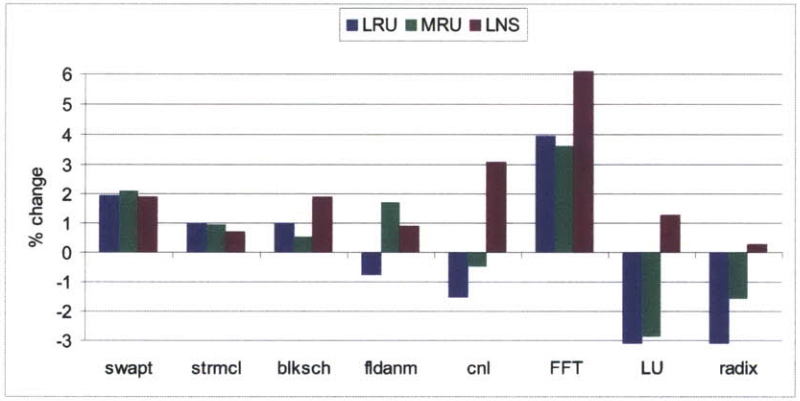


Figure 6-4: 1IPC core model

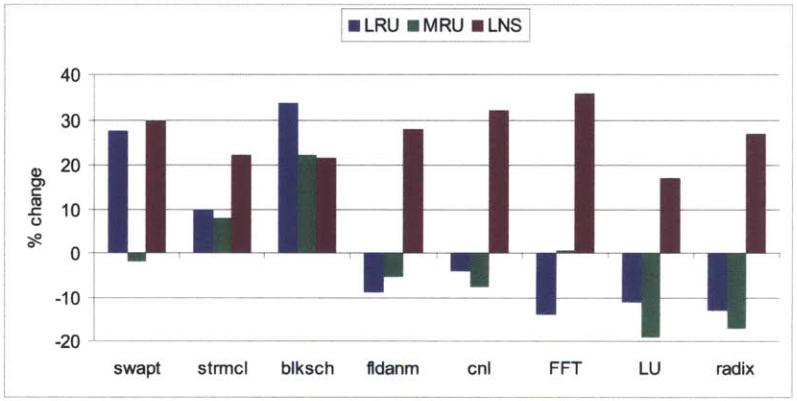
Using the 1IPC core model, we conducted the cache line replacement policy experiments again and obtained the results which are shown in Figure 6-5. Comparing these graphs with the corresponding ones in Figure 6-3, we see that there is a lot of variation in results. Although some results obtained from 1IPC match quite closely with those obtained from ACC, others are very different, and may lead to contradictory conclusions. We calculated the average quantitative error magnitude in these results as approximately 59% when compared with the results obtained from ACC.

6.4.1 Explaining the differences in results

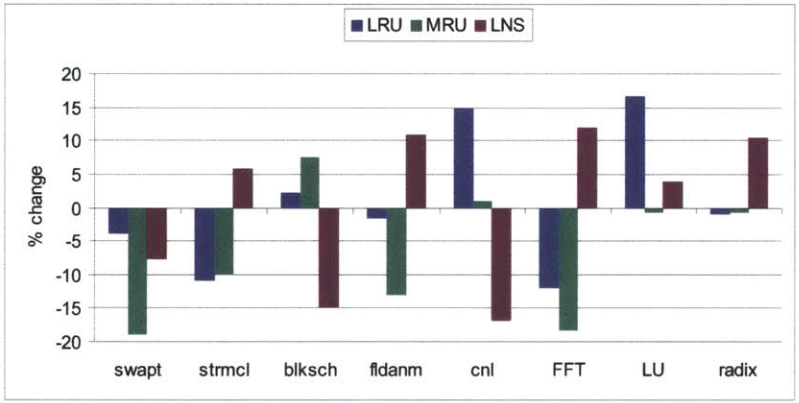
To determine the reasons behind these large differences in the results obtained from the 1IPC and the ACC core models, we looked at the differences in the number of committed instructions and the rate of pipeline bubbles due to cache misses. These differences are shown in Figure 6-6. We see that every multithreaded application assumes different instruction paths on 1IPC and ACC. This results in 26% to 41% fewer



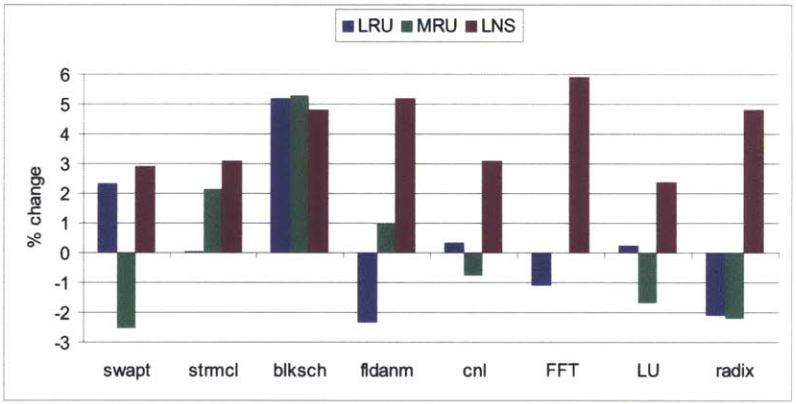
(a) Cache hit rate



(b) Cache coherence traffic

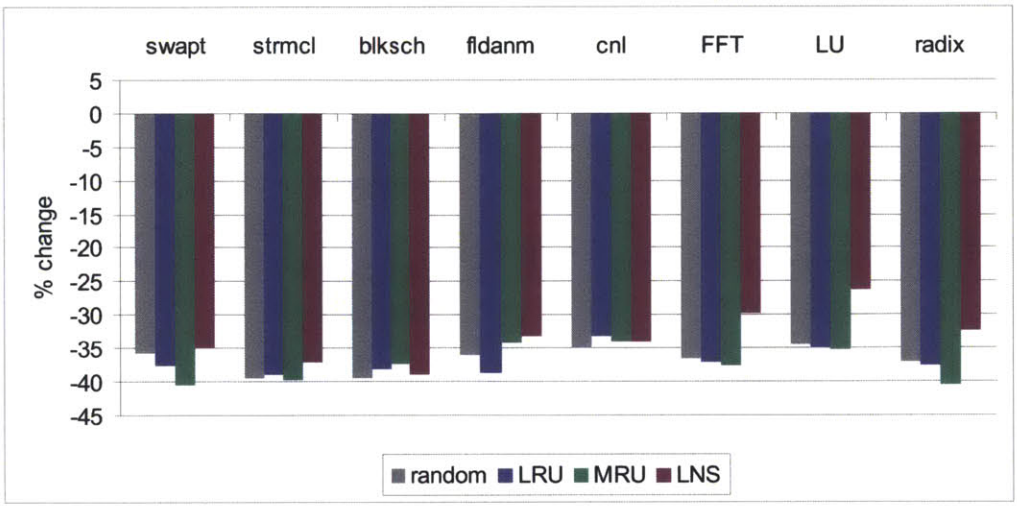


(c) Memory coherence traffic

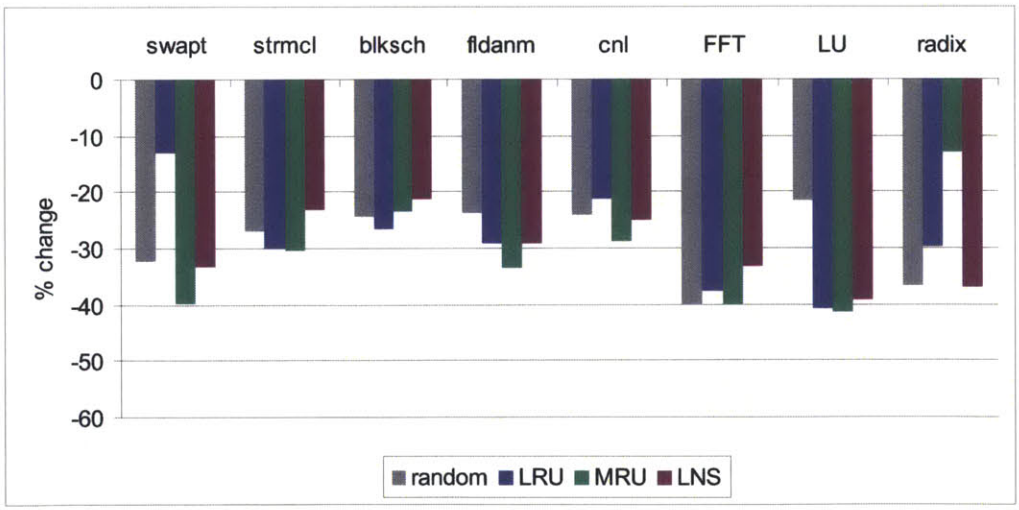


(d) Performance

Figure 6-5: Impact of LRU, MRU and LNS replacement policies obtained from 1IPC. Baseline replacement policy is random.



(a) Number of committed instructions



(b) Rate of pipeline bubbles due to cache misses

Figure 6-6: Comparison of the 1IPC core model with the ACC core model (baseline)

instructions committed by 1IPC than by ACC, when running the same application. We also see that the 1IPC core model results in a substantially lower rate of pipeline bubbles due to cache misses.

In terms of architectural differences, we found that on average 30% of the instructions fetched and executed by the ACC core model were wrong path instructions and approximately 35% of these wrong path instructions were either loads or stores. (Loads and stores also accounted for approximately 35% of the total committed instructions.) This means that when the 1IPC core model was used, both the I-cache and the D-cache had 30% fewer requests from the core, resulting in very different memory and network traffic. We also found that on average 27% of the stalls on ACC were due to data hazards. These stalled cycles were not modeled by 1IPC. As a result of the missing speculative instructions and data hazards, we found that the execution rate of 1IPC was, on average, 31% higher than that of ACC.

These architectural differences, particularly the large difference in the execution rate, resulted in substantial software differences as well. We found that when applications were run on the 1IPC core model, on average, 36% fewer timer interrupts took place compared to when they were run on the ACC core model. Since Arete runs at a clock speed of 11.11 MHz, we expect that the number of timer interrupts during the execution of an application will be 100× to 300× more than on a real processor. When we increased the `clock-frequency` parameter in the device tree source (dts) file for the Linux kernel by 100×, we saw that the difference in the number of timer interrupts between ACC and 1IPC reduced to 14%.

All the applications that we run use POSIX synchronization (*e.g.*, `pthread_mutex_t`) implemented using the kernel's futex API. We found that, on average, 29% fewer instructions were executed from the synchronization subroutines on 1IPC than on ACC.

When an application thread blocks on a `pthread_mutex_t`, the kernel marks the thread as "un-runnable". If there are no runnable threads, a core enters the idle loop (`cpu_idle`) and spins in it waiting for a thread to become runnable. The large difference in the execution rate between 1IPC and ACC also impacted the scheduling of tasks by the operating system which resulted in a substantial difference in time

spent in the idle loop. We found that the 1IPC core model executed, on average, 32% fewer instructions from `cpu_idle` compared to the ACC core model.

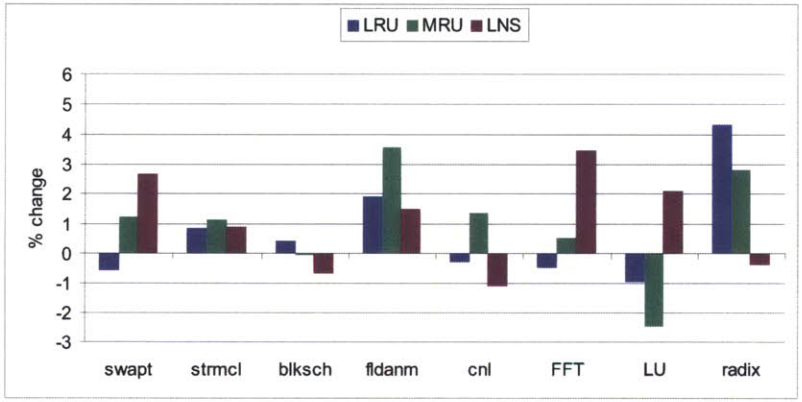
6.5 Improving the accuracy of 1IPC

6.5.1 Lowering the execution rate

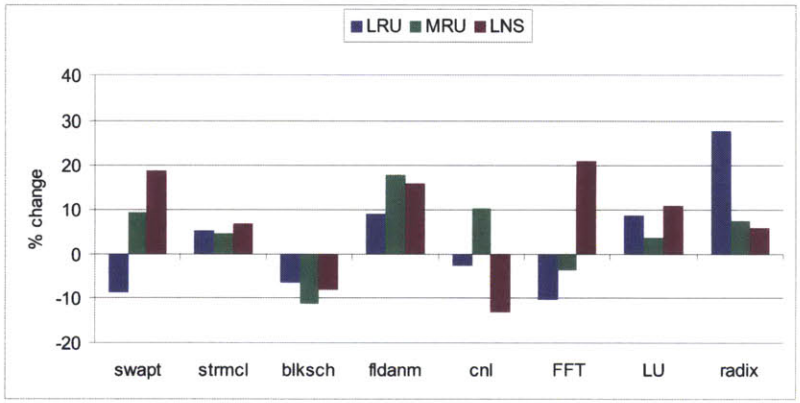
To improve the accuracy of the IPC core model, we decided to lower its execution rate to the same value as that of ACC. For this purpose, we introduced stall logic in 1IPC. We refer to the new core model as 1IPC-R. While running the applications using the ACC core model we had determined that the rate of instructions per cycle per core was approximately 0.67. We configured the stall logic in 1IPC-R to introduce pipeline bubbles every three out of ten model cycles.

Using the 1IPC-R core model in the simulator, we ran the cache line replacement policy experiments again and obtained the results which are shown in Figure 6-7. Comparing these graphs with the corresponding ones in Figure 6-3, we again see that all the results are quite different. Moreover, the results obtained when using 1IPC-R are also very different from those obtained when using 1IPC. We calculated the average quantitative error magnitude in the results obtained from 1IPC-R as approximately 27% when compared with the results obtained from ACC.

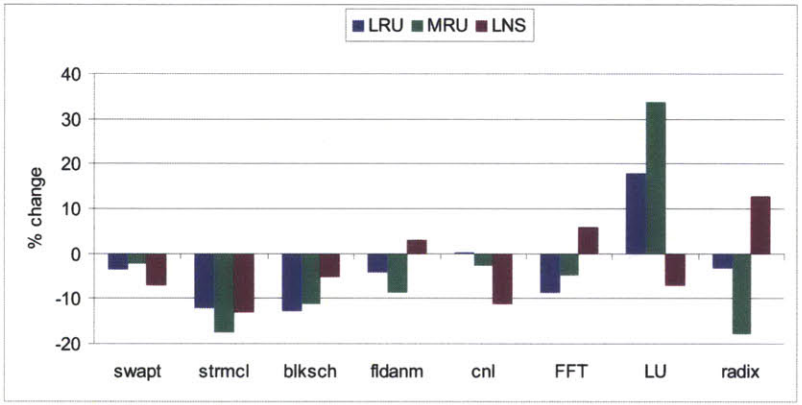
Figure 6-8 provides a comparison between the 1IPC-R and ACC core models. We see that although the difference in the number of committed instructions is substantially reduced compared to Figure 6-6(a), the difference in the rate of pipeline bubbles due to cache misses is higher than in Figure 6-6(b). The latter happens because the stall logic in 1-IPC-R causes only a slight increase in cache misses, but increases the total execution time quite substantially. These results indicate that the role of speculative instructions in the behaviors of memory and network, as well as in the overall execution time, cannot be ignored.



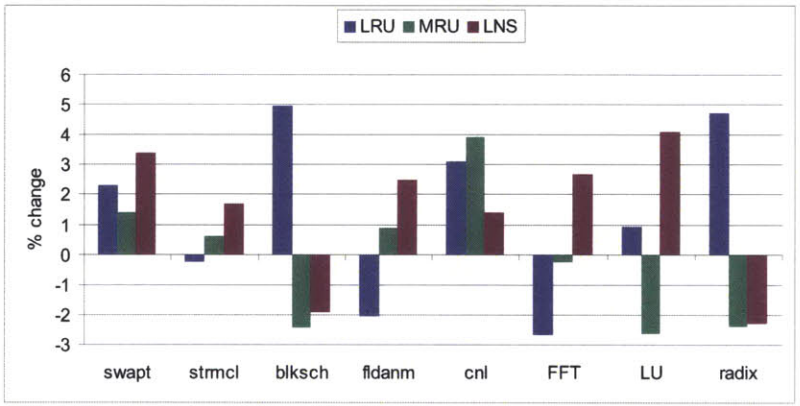
(a) Cache hit rate



(b) Cache coherence traffic

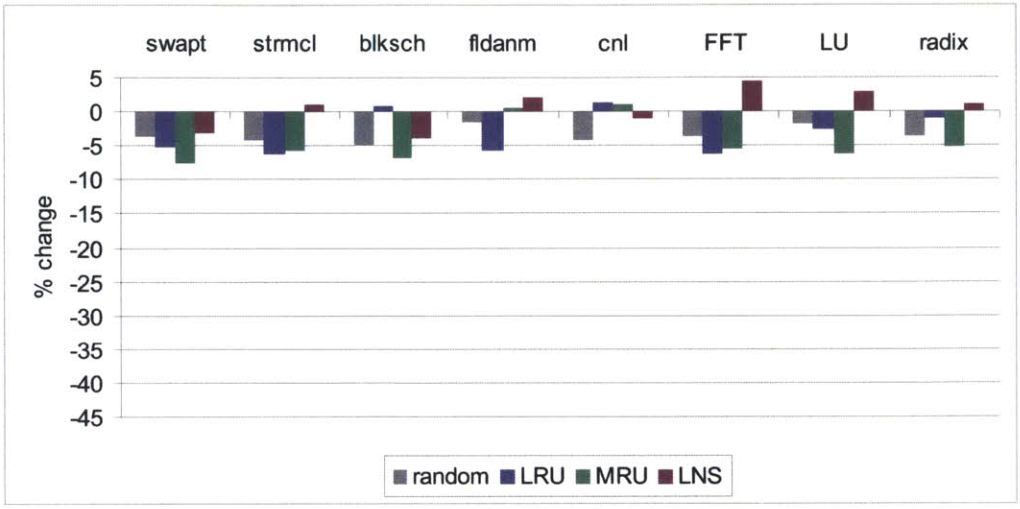


(c) Memory coherence traffic

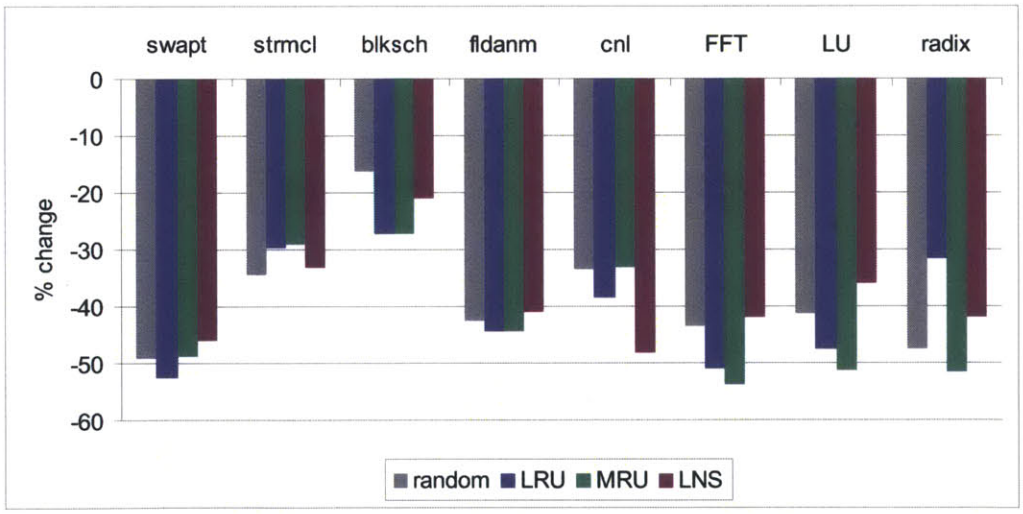


(d) Performance

Figure 6-7: Impact of LRU, MRU and LNS replacement policies obtained from 1IPC-R. Baseline replacement policy is random.



(a) Number of committed instructions



(b) Rate of pipeline bubbles due to cache misses

Figure 6-8: Comparison of the 1IPC-R core model with the ACC core model (baseline)

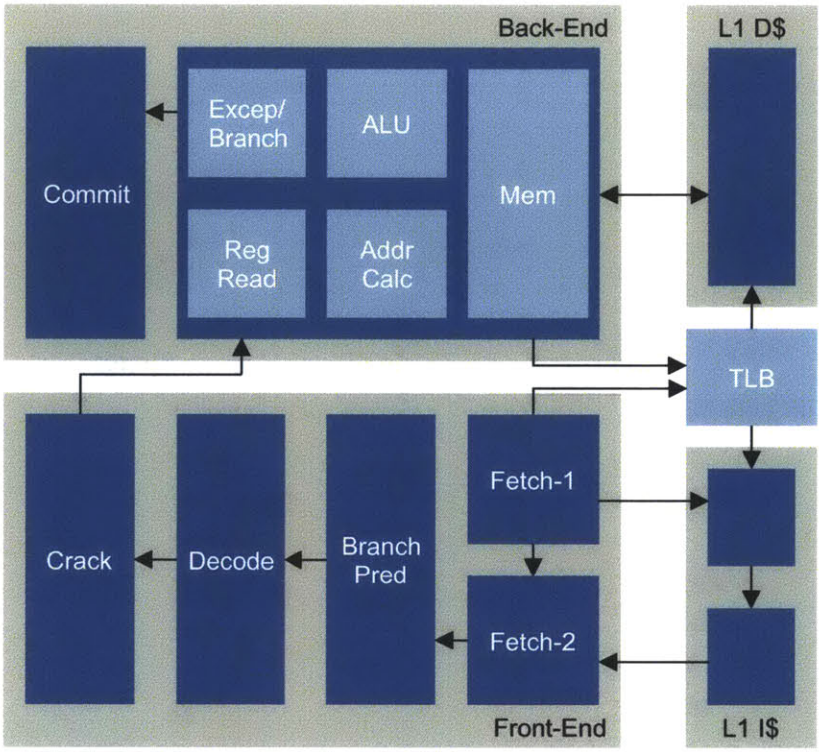


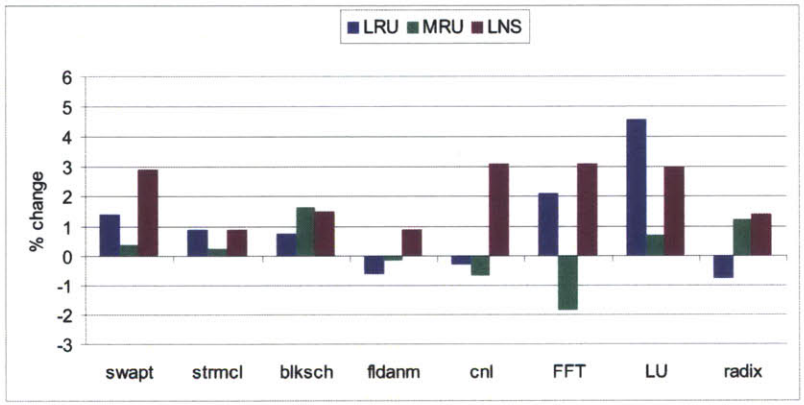
Figure 6-9: 7NDH core model

6.5.2 Adding speculative instructions

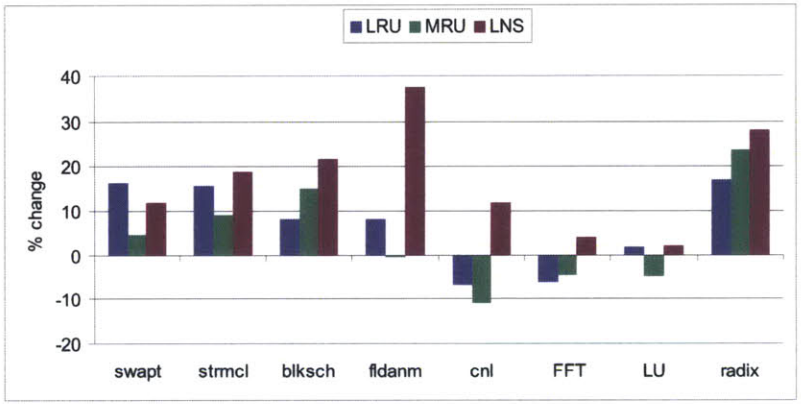
After experimenting with the 1IPC-R core model, we decided to add handling of mispredicted branches and exceptions/interrupts to the processor pipeline. The only functionality missing from this pipeline is the handling of stalls due to data hazards. The first no-data-hazard (NDH) core model is shown in Figure 6-9. The front-end of the processor pipeline is identical to that in ACC. The back-end, however, comprises of only two stages: execute and commit. We refer to this core model as 7NDH.

Using the 7NDH core model in the simulator, we conducted the cache line replacement policy experiments again. The results obtained from these experiments are shown in Figure 6-10. Comparing these graphs with the corresponding ones in Figure 6-3, we see that all the results are still quite different. The average quantitative error magnitude in these results was approximately 30%.

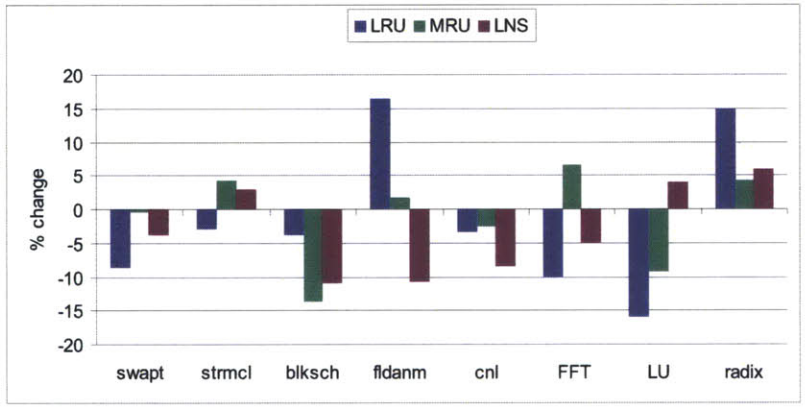
To gain insight into the differences in results obtained from 7NDH and ACC, we compared the two core models in terms of committed instructions and rate of pipeline



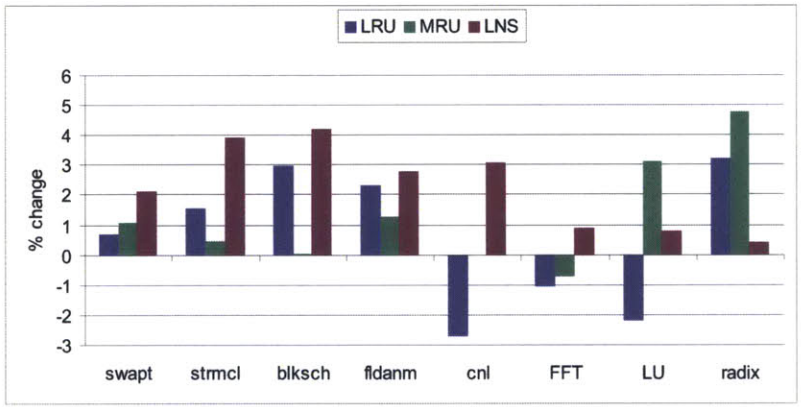
(a) Cache hit rate



(b) Cache coherence traffic

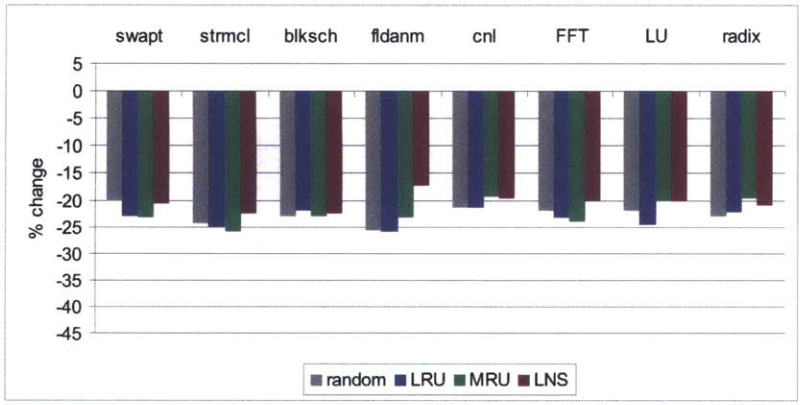


(c) Memory coherence traffic

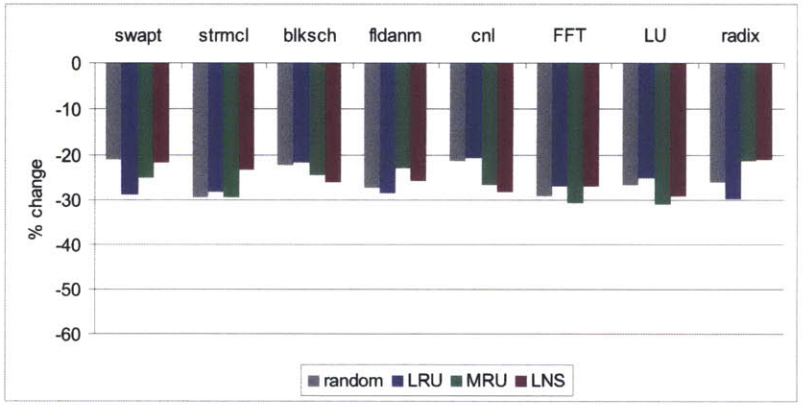


(d) Performance

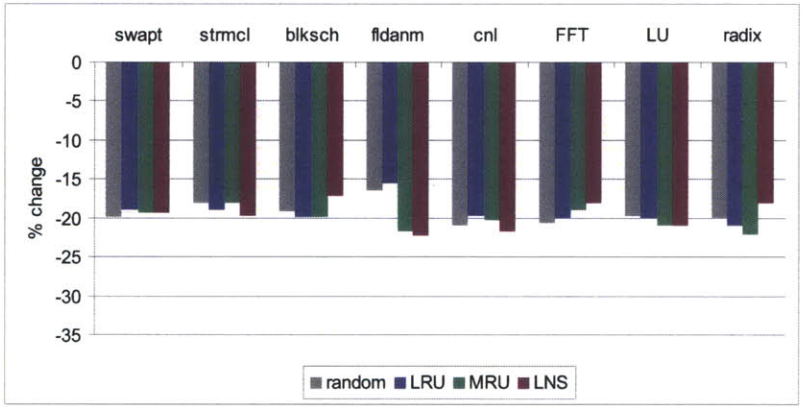
Figure 6-10: Impact of LRU, MRU and LNS replacement policies obtained from 7NDH. Baseline replacement policy is random.



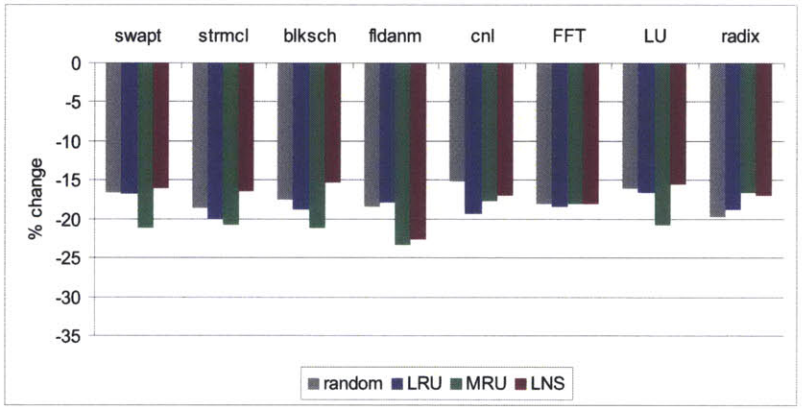
(a) Number of committed instructions



(b) Rate of pipeline bubbles due to cache misses



(c) Rate of pipeline bubbles due to branch mispredictions



(d) Rate of pipeline bubbles due to exceptions/interrupts

Figure 6-11: Comparison of the 7NDH core model with the ACC core model (baseline)

bubbles due to cache misses, branch mispredictions and exceptions/interrupts. These comparisons are shown in Figure 6-11. We see that in all the comparisons, 7NDH trails ACC by approximately 15% to 30%.

The execution rate of 7NDH was found to be approximately 15% higher than that of ACC. To further improve the accuracy of 7NDH we decided to lower its execution rate by adding stall logic to the back-end of the pipeline. To achieve the same rate of instructions per cycle per core as in ACC, we configured the stall logic to introduce a pipeline bubble every fourth model cycle. We refer to the new core model as 7NDH-R.

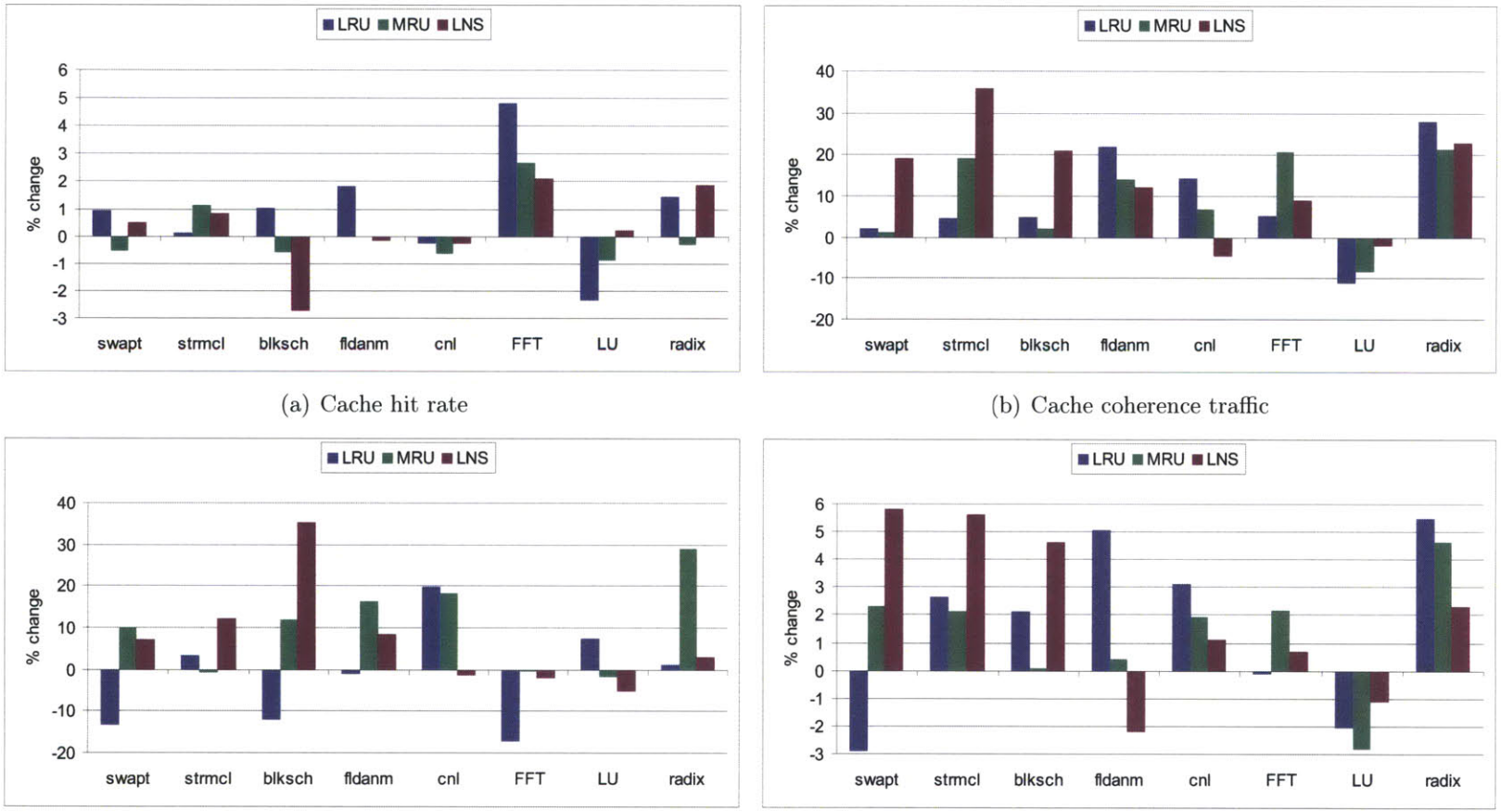
After running the cache line replacement policy experiments using 7NDH-R, we obtained the results that are shown in Figure 6-12. These results also turned out to be quite different from those obtained using ACC. The average quantitative error magnitude in these results was approximately 15%.

Figure 6-13 provides the comparison between 7NDH-R and ACC. As in the case of 1IPC and 1IPC-R, we see that when 7NDH-R is used, the difference in the number of committed instructions is substantially reduced compared to Figure 6-11(a), but the differences in the rate of pipeline bubbles due to cache misses, branch mispredictions and exceptions/interrupts are higher than in Figures 6-11(b), 6-11(c) and 6-11(d), respectively.

6.5.3 Adding the full speculative path

Besides the absence of pipeline stalls due to data hazards, a significant difference between 7NDH and ACC is the mismatch between the number of speculative instructions that are fetched and executed. In the case of 7NDH, the speculative path is 6 stages long, while in the case of ACC, it is 9 stages long. In order to determine if this mismatch can account for most of the differences between 7NDH and ACC, we introduced three bypass stages in the back-end of the processor pipeline, as shown in Figure 6-14. We refer to the core model thus obtained as 10NDH.

We performed the cache line replacement policy experiments using 10NDH, and obtained the results which are shown in Figure 6-15. Comparing these graphs with the corresponding ones in Figure 6-3, we see that the two sets of results are quite



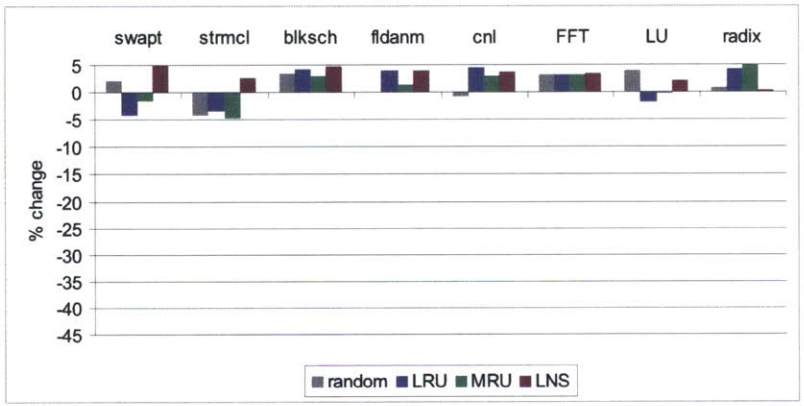
(a) Cache hit rate

(b) Cache coherence traffic

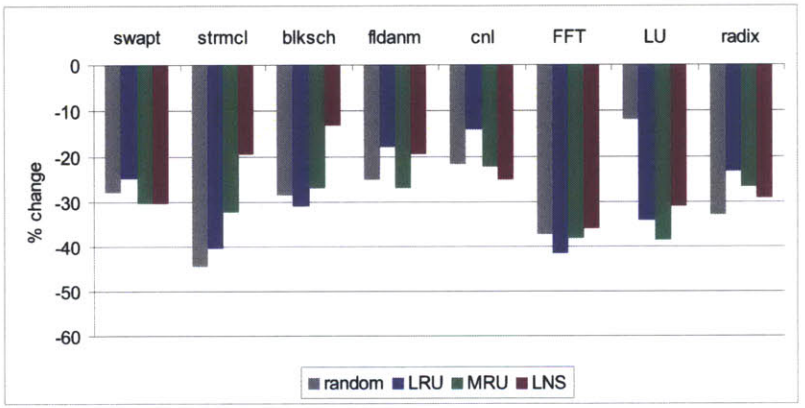
(c) Memory coherence traffic

(d) Performance

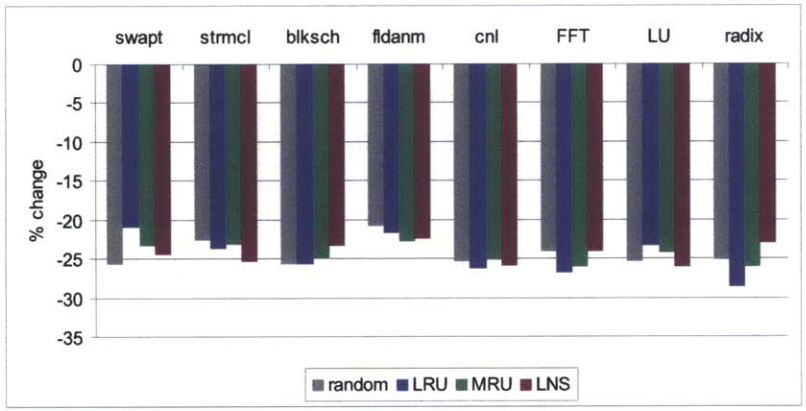
Figure 6-12: Impact of LRU, MRU and LNS replacement policies obtained from 7NDH-R. Baseline replacement policy is random.



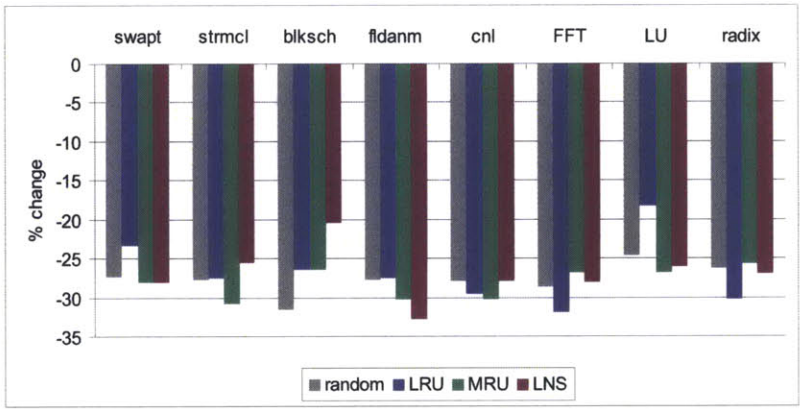
(a) Number of committed instructions



(b) Rate of pipeline bubbles due to cache misses



(c) Rate of pipeline bubbles due to branch mispredictions



(d) Rate of pipeline bubbles due to exceptions/interrupts

Figure 6-13: Comparison of the 7NDH-R core model with the ACC core model (baseline)

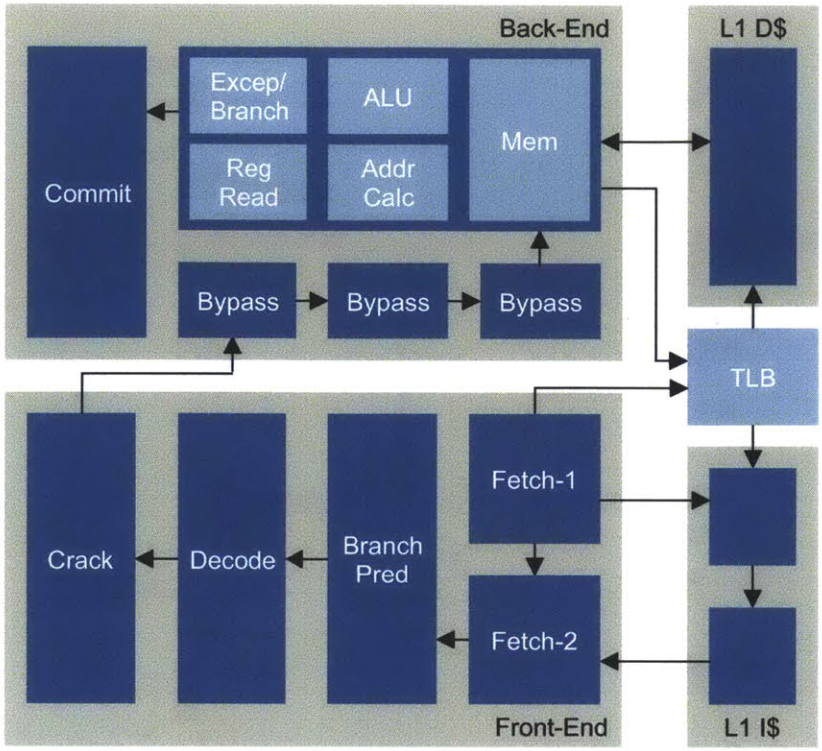
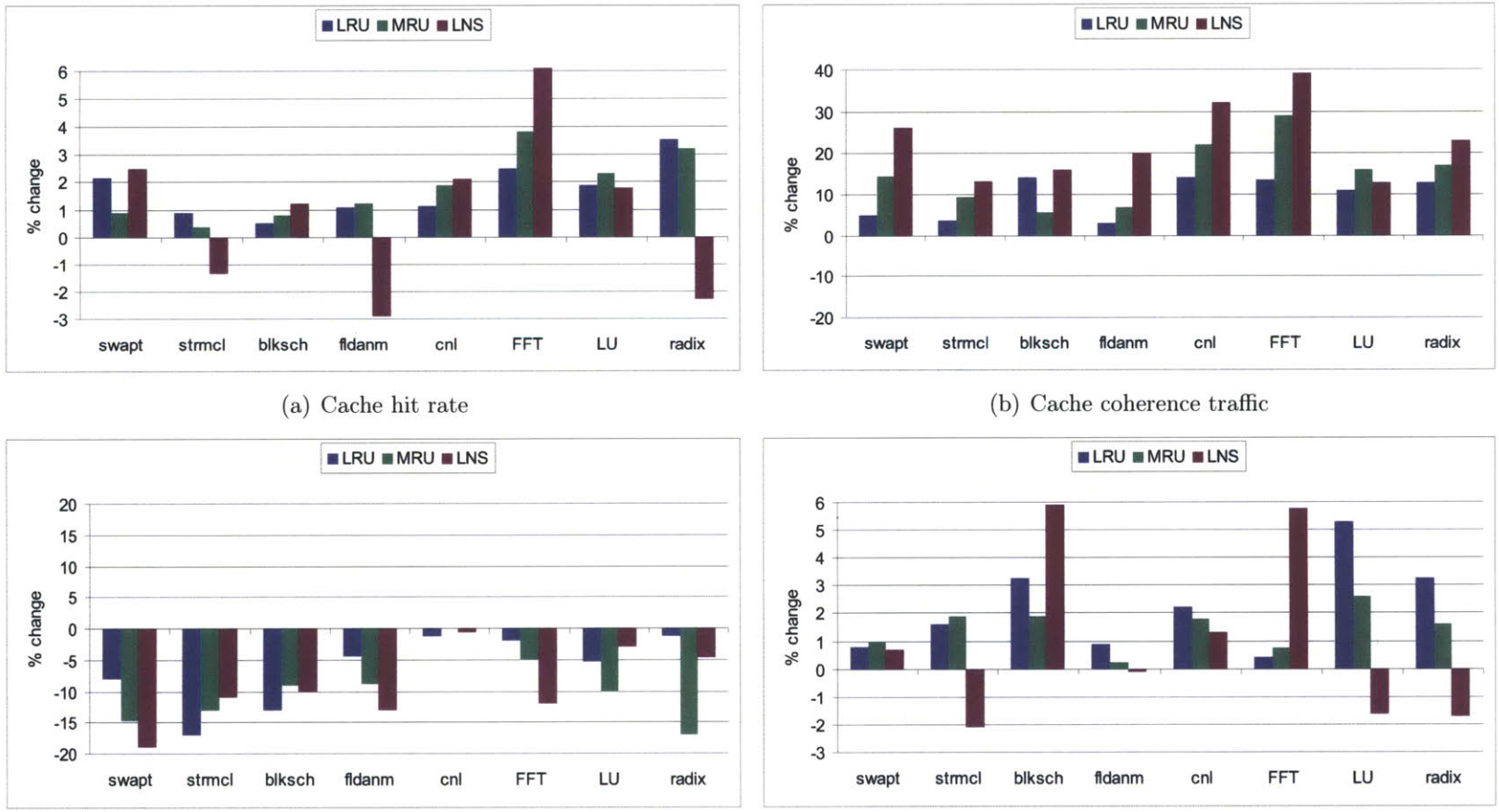


Figure 6-14: 10NDH core model

similar, and 10NDH is able to capture all the trends obtained using ACC. The average quantitative error magnitude in these results was approximately 11%.

Figure 6-16 provides the comparison between 10NDH and ACC. We see that all the differences are quite small, and that variation in differences across cache line replacement policies is also quite small. Moreover, the difference in the rate of instructions per cycle per core between 10NDH and ACC is minimal.

The results obtained using 1IPC, 1IPC-R, 7NDH, 7NDH-R and 10NDH concretely show that a) accurate modeling of speculative instructions is necessary, and b) pipeline stalls due to data hazards can be ignored when exploring the impact of replacement policies in the L2 cache.



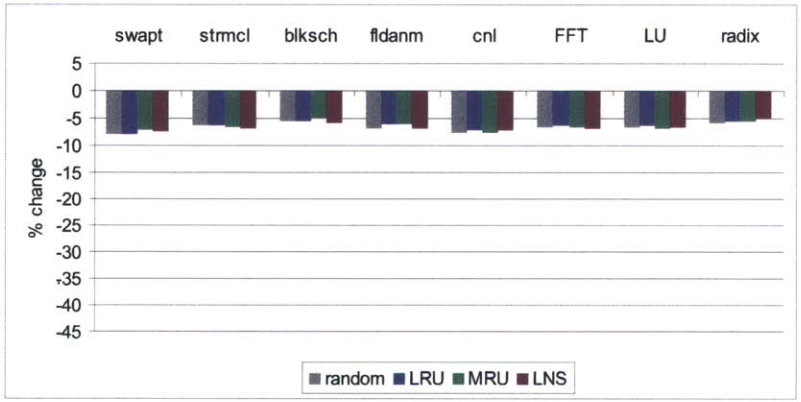
(a) Cache hit rate

(b) Cache coherence traffic

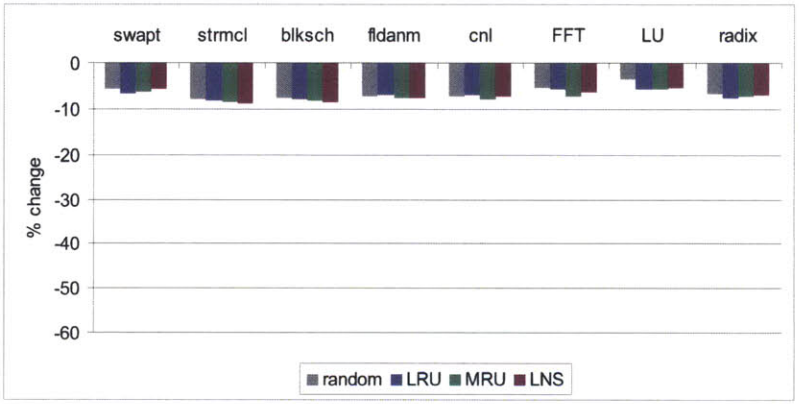
(c) Memory coherence traffic

(d) Performance

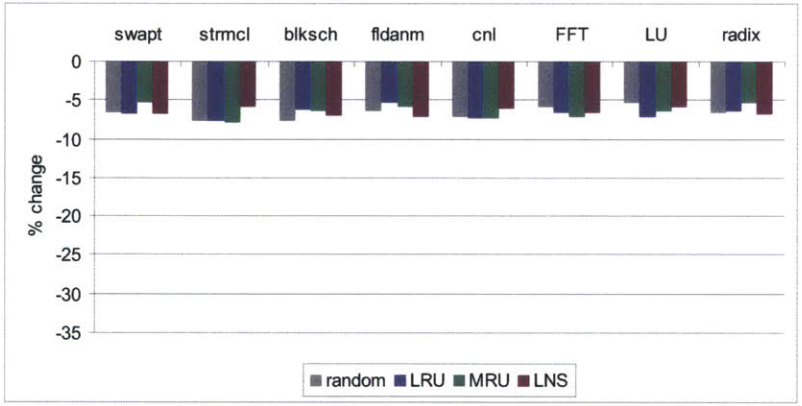
Figure 6-15: Impact of LRU, MRU and LNS replacement policies obtained from 10NDH. Baseline replacement policy is random.



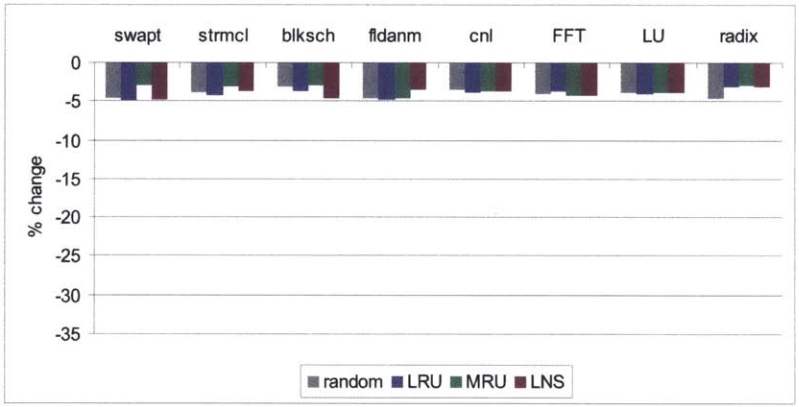
(a) Number of committed instructions



(b) Rate of pipeline bubbles due to cache misses



(c) Rate of pipeline bubbles due to branch mispredictions



(d) Rate of pipeline bubbles due to exceptions/interrupts

Figure 6-16: Comparison of the 10NDH core model with the ACC core model (baseline)

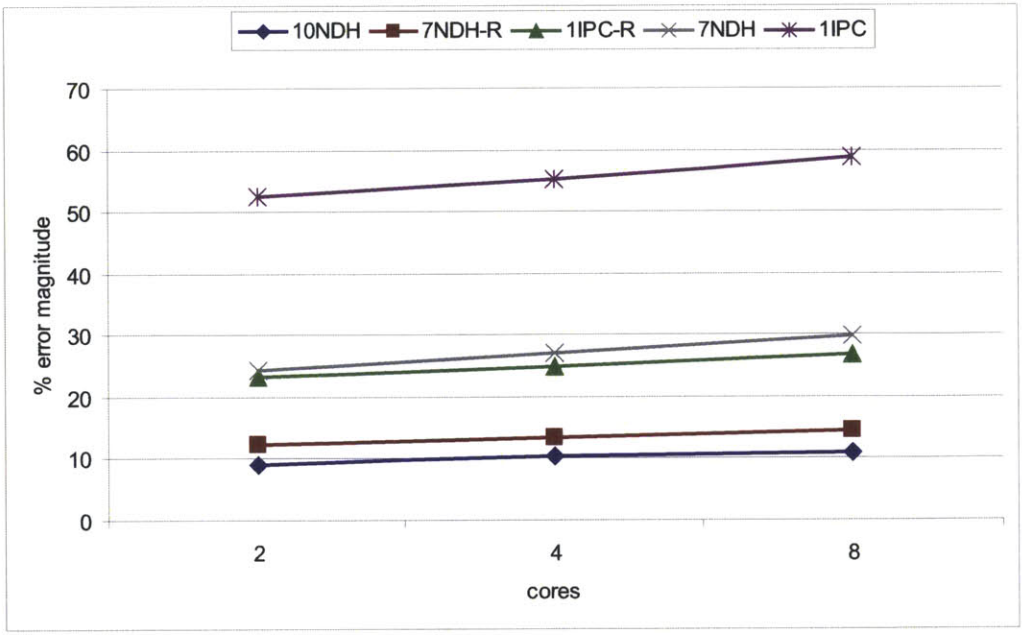


Figure 6-17: Increase in mean error magnitude as the number of cores increases in processor simulations with various coarse-grained core models

6.6 Additional comments

6.6.1 Error scaling

To determined how error magnitude scales with the number of cores, we computed the mean error magnitude for the 2-core, 4-core and 8-core simulator configurations. The mean is calculated across the four experiments, *i.e.,* cache hit rate, cache traffic, memory traffic and performance, for each of the five coarse-grained core models. In Figure 6-17, we see that the mean error magnitude increases with the number of cores for all the coarse-grained core models, albeit at different rates. When simulating processor configurations with hundreds or thousands of cores using coarse-grained core models, the error magnitude is expected to be quite substantial.

6.6.2 Variability study

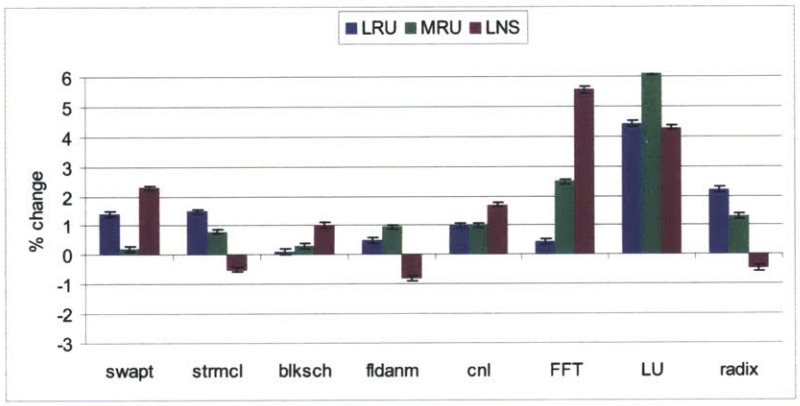
To gain confidence in our results and to ensure that they are not skewed, we also performed a variability study. We ran each application sixteen times on the 8-core

simulator configuration with the ACC and 1IPC core models. For each application run, we varied the memory latency using a uniformly distributed pseudo random integer between 0 and 32. We also varied the scheduling of requests in the L2 cache, again using a uniformly distributed pseudo random integer. The results are shown in Figures 6-18 and 6-19. The colored bars show the average results across sixteen application runs, while the marks show the minimum and the maximum values. These results exhibit little variability and match quite well with those from Figures 6-3 and 6-5. We believe that this is due to the length of application runs which is 100 billion instructions on average [66].

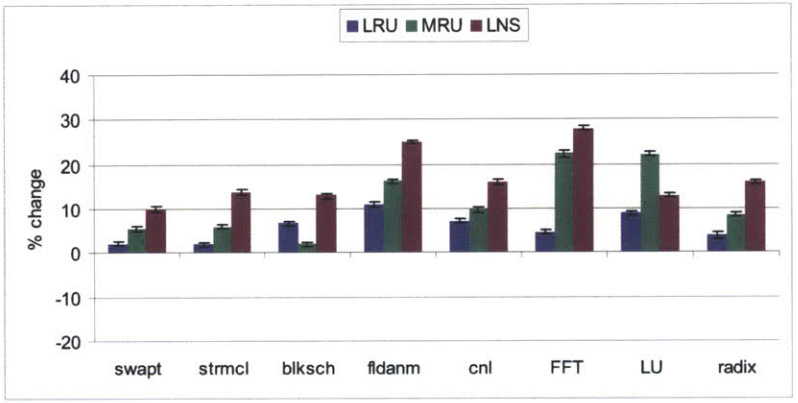
6.6.3 Comparison against real machines

Figure 6-20 presents a comparison of statistics obtained from running canneal with the simlarge input on Arete, ARM Cortex-A9 and Core i7-965. In case of Arete and Core i7, the instruction count includes both user and privileged code, while in case of ARM, the application is run in stand-alone mode (without booting an operating system). We see that the instruction count on ARM is slightly less than that on Arete. This difference can be attributed to the missing operating system effects on ARM, the 70× more timer interrupts on Arete due to the lower clock frequency, and the counting of load and store multiword instructions as multiple instructions on Arete. On the other hand, the instruction count on Core i7 is half of that on Arete, but it matches quite well with the instruction count on an SMP x86 processor reported in [67]. Comparing the micro ops on Core i7 with the instruction count on Arete, we see that the difference is reduced to 23%.

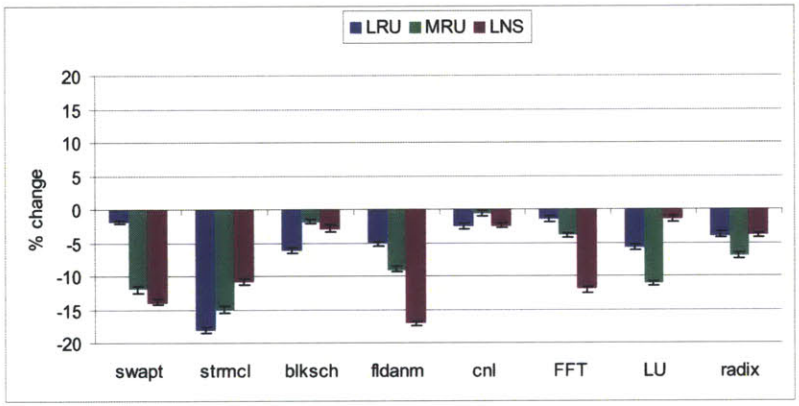
The cycle count on Arete and Core i7 match almost exactly, but the cycle count on ARM is only a quarter of that on Arete. The ARM Cortex-A9 processor includes a 4-wide out-of-order super-scalar core, and it is able to achieve a very high instruction throughput. The low instruction throughput on Core i7 can be attributed to the low cache hit rate. A survey of reported last level cache hit rates for canneal confirms large fluctuations with the architecture of the cache hierarchy. In [68], cache hit rate it is reported as 12% on a Phenom4 processor; in [67], it is reported as 30% for a 1



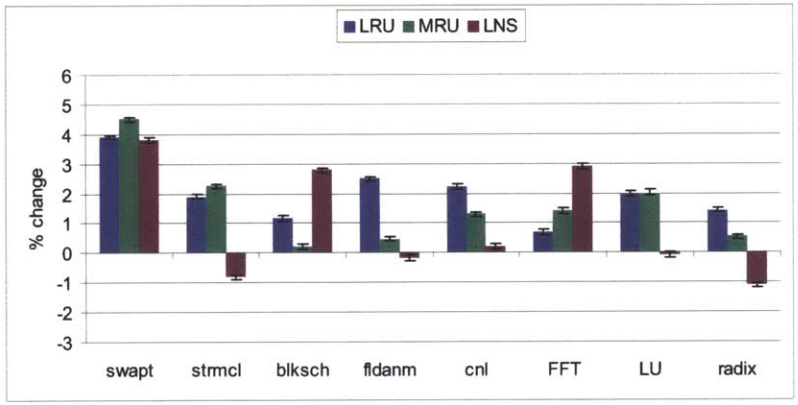
(a) Cache hit rate



(b) Cache coherence traffic

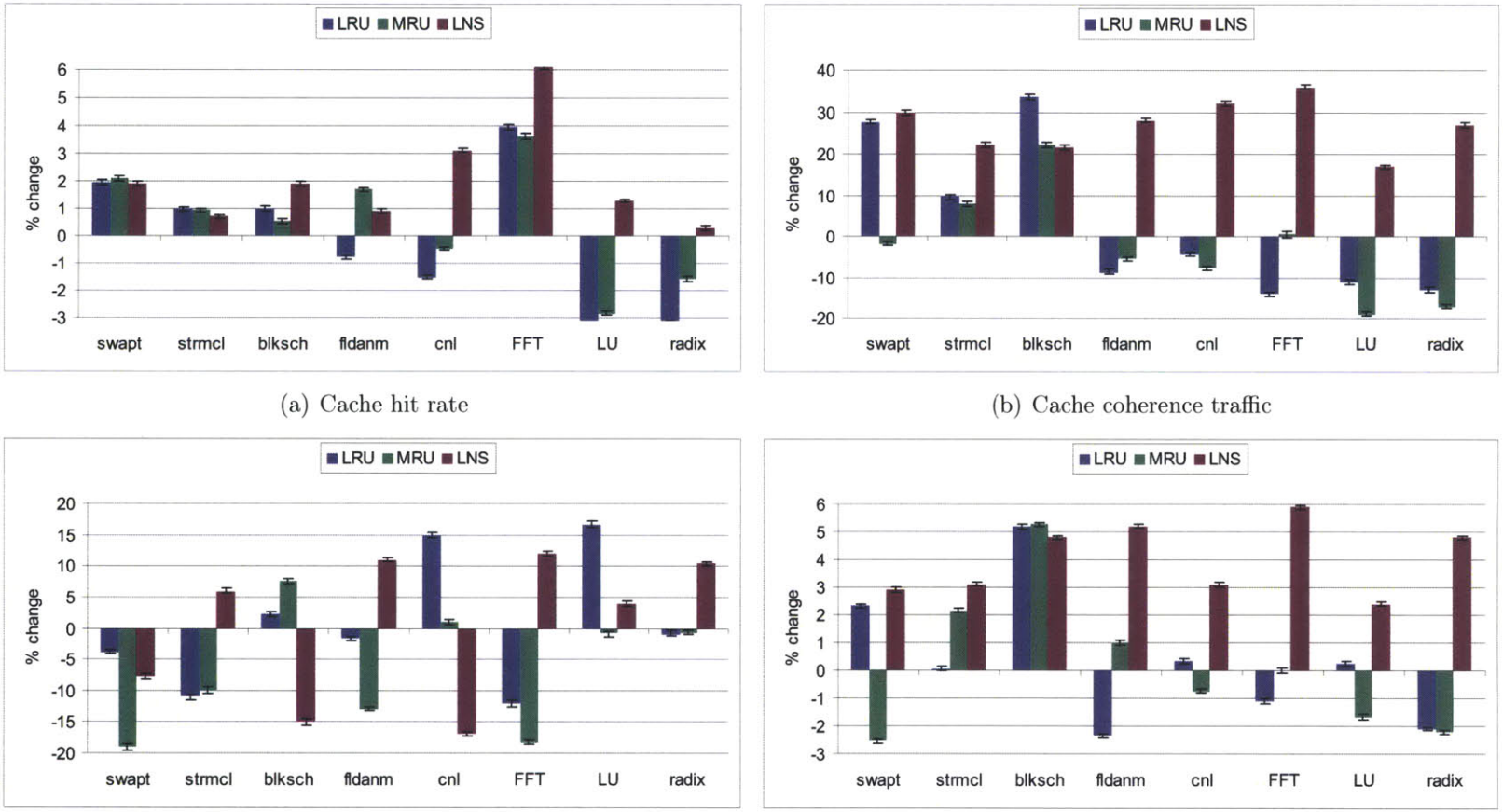


(c) Memory coherence traffic



(d) Performance

Figure 6-18: Impact of LRU, MRU and LNS replacement policies obtained from ACC. Baseline replacement policy is random. Bars show the average values from sixteen application runs, while marks show the minimum and the maximum values.



(a) Cache hit rate

(b) Cache coherence traffic

(c) Memory coherence traffic

(d) Performance

Figure 6-19: Impact of LRU, MRU and LNS replacement policies obtained from 1IPC. Baseline replacement policy is random. Bars show the average values from sixteen application runs, while marks show the minimum and the maximum values.

	Arete	ARM Cortex-A9	Core i7-965
Instructions	14B	12.1B	6.9B
Micro ops	-	-	10.7B
Cycles	23B	5.8B	22.6
Branch miss rate	11.8%	-	9.6%
L1 I-cache hit rate	99.9%	-	99.6%
L1 D-cache hit rate	97.7%	-	71.4%
Last level cache hit rate	97.5%	-	29.9%

Figure 6-20: Comparison of statistics obtained from running the same application on Arete, ARM Cortex-A9 and Core i7-965

MB last level cache in an SMP x86 processor; while in [69] it is reported as 99% for a 1 MB shared cache in an 8-way CMP.

6.7 Related work

There is a large body of work which proposed various techniques to improve the speed of processor simulations without compromising the accuracy of the architectural studies.

Black *et al.* [64] provided an instructive overview of the various simulation techniques. They identified the three basic kinds of simulation errors: 1) modeling errors, which are present in the simulator code, 2) specification errors, where the specification of the target processor architecture is erroneous, and 3) abstraction errors, which creep into the simulator because modeling is performed with insufficient detail. They described the design process used by architects. It starts with a crude simulation model which is systematically refined by adding more architectural details and fixing bugs, resulting in an accurate simulator of the target architecture. They also claimed that model refinement does not improve the accuracy of the model monotonically. Their work showed the need for extensive, iterative validation before the results obtained from a simulator can be trusted.

Skadron *et al.* [65] gave an overview of the state of affairs in simulation techniques,

and provided recommendations on how they can be improved. They argued that designing benchmarks, both micro and macro, is paramount for accurately predicting processor performance. They also claimed that analytical modeling techniques were not very well studied, and that such techniques would gain significance given the trend in computer architecture to move towards multicore architectures, which would further slow down the already quite slow simulators. They also argued that the impact of modeling abstractions on the accuracy of performance projections was not well understood and extensive studies were needed to classify a specific abstraction as good or bad.

In [70], Yi *et al.* described, classified and compared a wide range of simulation methodologies. These included techniques for validation of simulators, selection of parameters, benchmarks and input sets, shortening of simulation time through reduced input sets, truncated execution and sampling, and reduction of the variability in performance analysis through statistical approaches.

Bose *et al.* [62] argued that detailed simulation was expensive and not plausible. They stated that one should build abstract simulators, instead, and that these abstract simulators must be calibrated against existing real machines by running microbenchmarks that target specific features such as cache behavior, loop execution, etc.

Desikan *et al.* [60] compared sim-alpha, which is an out-of-order processor simulator implemented according to the specification of the Alpha 21264 microarchitecture, against a real Compaq DS-10L workstation. Their results showed that unvalidated simulators report higher performance than the target architectures that they model. They recommended using microbenchmarks to calibrate simulators against real machines. To avoid errors due to incorrect parameters, they recommended obtaining parameter values from either published documents or real machines.

In [61], Cain *et al.* argued the need for detailed models and actual workloads to accurately predict the performance of real designs. They showed that OS and I/O effects significantly impact the accuracy of performance results. They also asserted that simulating instructions on the speculative paths did not impact the accuracy

of the performance results. In our results, we show that this assertion is completely false, and that accurate modeling of speculative instructions is perhaps the most significant factor in the accuracy of simulation studies in memory and network of multicore processors.

Collectively, [71, 72], showed that performance results obtained from unvalidated uniprocessor simulators vary substantially from the performance of the target architecture, and cannot be deemed reliable. However, abstractions used in the simulator can be iteratively refined through calibration against real machines to obtain fairly accurate results. Similarly, parameter values can be obtained from either real machines or rigorous statistical methods [73] to improve simulation accuracy.

Alameldeen and Wood [66] investigated the potential impact that variability can have on simulation results. Their experiments showed that even a small amount of variability, such as addition of a random amount of time to each L2 cache miss, can lead to wrong conclusions in architectural studies. To minimize this problem, they recommended adding pseudo random perturbation to the simulations, simulating each test case multiple times, and using confidence intervals and hypothesis testing.

In [74], Alameldeen and Wood argued that during the various instruction paths that result from timing variations, a program can execute a substantially larger or smaller number of instructions to perform the same amount of useful work. The number of instructions executed on a particular path is determined by how much time the program spends executing idle-loop instructions, spin lock wait instructions or system-level privileged code instructions. Although such instructions have little impact on the amount of useful work a user program actually performs, they significantly change system behavior. They concluded that using the rate of instructions per cycle as a measure of system performance was not reliable because instructions per program vary substantially across various runs of the program.

In [40], Pellauer *et al.* studied the impact of core detail on on-chip network simulations. They performed the same simulation study using 1IPC cores with only one outstanding instruction miss, and using detailed 9-stage core models. The results showed a significant error in performance estimates obtained from the 1IPC cores.

6.8 Summary

In this chapter we presented a comprehensive evaluation of the use of coarse-grained core models in multicore processor simulators for studying the memory system and the on-chip network, and presented four significant results.

1. Use of the 1IPC core model leads to grossly inaccurate conclusions.

2. Pipeline stalls due to data hazards need not be modeled, but accurate modeling of speculative instructions is necessary for reliable performance results.

3. Difference in the number of executed instructions between simplified and accurate core models can be minimized through stall logic which equalizes the rate of instructions per cycle per core, but performance results remain largely inaccurate.

4. Error magnitude increases with the number of cores.

Based on these results, we argue that although simulation speed is a major concern in the modeling of many-core processors, obtaining performance results at a faster rate would prove inconsequential if the results led to wrong architectural decisions. We propose that when architects use coarse-grained models in full-system simulations, they carefully validate the simulator through a wide range of techniques, such as those demonstrated in this chapter, and established in previous work.

Chapter 7

Data Movement Control: An Architectural Experiment[1]

7.1 Introduction

There is an inherent cost for applications to access off-chip DRAM. A potential solution is a single large on-chip cache that all cores access with uniform latency. This architecture makes it easy for application developers to implement efficient inter-core sharing and use the entire on-chip cache. A large shared on-chip cache, however, is still prohibitively slow. Architects ensure each core has fast access to some portion of on-chip memory by distributing on-chip memory in pieces so that every core is near some cache. In theory this provides a large amount of aggregate cache capacity and fast memory for each core. Unfortunately, it is more difficult for software to use a distributed cache effectively than a shared cache effectively. The goal of this project is to explore whether extensions to hardware can help software make better use of distributed caches on multicore processors.

Consider some of challenges faced by software trying to use a distributed cache on a multicore processor. In some architectures, an application can cache data only in

[1]The work presented in this chapter was jointly carried out with Silas Boyd-Wickizer.

the caches of the cores that it is currently executing on. This provides applications with access to only a small amount of on-chip cache capacity. Even if the application is executing on all cores, it is expensive to access data in a remote core's cache, and it is likely that each core's individual caches would end up caching the same commonly accessed data. Duplicating data reduces the number of distinct data items cached on-chip, which essentially reduces the effective cache capacity.

Promising software solutions (*e.g.,* [75, 76]) use thread migration to help manage cache contents. The basic idea behind these solutions is to assign data items to on-chip caches and migrate threads amongst the caches as they access the data items. Moving a thread closer to the data that it accesses reduces access latencies and helps ensure that the same data is not duplicated many times. The implementations of these solutions, however, can have significant overheads. Migrating a thread can require as many as 20,000 cycles [77]. Distributing data items to caches requires software to track the data items it assigns to a certain cache, adding overhead and essentially duplicating directories maintained by hardware. Even if software is able to efficiently manage mappings of data items to caches, it can only guess if a data item is actually cached or has been evicted by hardware.

This project explores the opportunity to extend hardware to make it easier for applications to use on-chip caches efficiently, thereby improve performance. We introduce a set of hardware extensions, which we refer to as Data Movement Control (or DMC), that are in the form of three new instructions: `cpush`, `clookup`, and `cmsg`. The instructions give software more information about and control over on-chip cache contents. `cpush` allows a thread running on one core to move cache lines into another core's cache; `clookup` returns the location of any cache line; and `cmsg` provides an efficient mechanism for software to send an active message [78] to a remote core, which allows a thread to efficiently manipulate data in a remote core's cache. Collectively, these instructions address some of the shortcomings of previous software-only cache management solutions.

To evaluate potential performance improvements we implemented the DMC instructions in Arete. The resulting DMC PowerPC microarchitecture is backwards

compatible with the Book-E PowerPC microarchitecture originally implemented in Arete. The original implementation, as well as the DMC implementation, provide cycle-accurate processor timing when synthesized for the BEE3 board. Results from running synthetic benchmarks on the DMC-BEE3 core indicate that using DMC instructions can improve the performance of operations that manipulate as few as two shared cache lines.

A main challenge in implementing DMC is doing so without causing deadlocks or invalid states in the cache coherence controller. The implementation of `cpush` is particularly tricky because the cache coherence controller must handle race conditions where a core requests a particular cache line in a particular mode, *e.g.*, M, while another core simultaneously pushes the same cache line in a different mode, *e.g.*, S. Another challenge is implementing the DMC extensions so that they are compatible with existing PowerPC applications, yet still provide high performance. For example, when a remote core begins executing an active message it must not violate the PowerPC ABI, which mandates that software restore all the execution state (*e.g.*, register values) when the active message completes. If software saved and restored all execution state, however, active messages would be prohibitively expensive.

The main contributions of this project are (1) the introduction of the DMC hardware primitives that simplify software cache management; (2) a new type of active message that is addressed by memory address instead of destination core; and (3) an implementation and evaluation of DMC hardware using synthetic benchmarks.

Chapter organization: Section 7.2 briefly discusses some related work. Section 7.3 describes the interface and semantics of the DMC instructions, while Section 7.4 describes their design and the solutions to the design challenges that we faced. Section 7.5 discusses the implementation of the DMC instructions and our testing procedure. Section 7.6 describes the results obtained from applying the DMC instructions to microbenchmarks. Section 7.7 summarizes the work and discusses some of its limitations.

7.2 Related work

There is a significant amount of work related to multicore cache management and computation migration. This section presents a few examples from each category.

7.2.1 Multicore cache management

Several techniques have been proposed to improve cache management on multicore processors. O^2 [75] is a software runtime that manages cache contents using thread migration. The O^2 runtime attempts to track cache contents, assigns data to a cache when there is spare capacity, and migrates threads amongst cores as they access data items. Software data spreading [76] aims to allow single-threaded applications to use the capacity of caches in all the cores by using techniques similar to those of O^2. Several research operating systems, such as Corey [79], Barrelfish [80], and fos [81], try to improve cache usage through the operating system kernel by dedicating cores to operate on particular sets of kernel data.

7.2.2 Computation migration

The J-Machine was a 1024-node parallel computer built from message-driven processors [82], which provided low-overhead messaging and context switching, similar to `cmsg`. Application developers wrote fine-grained concurrent programs for the J-Machine using J-Machine-specific programming languages and tools that distributed data objects amongst the nodes and took advantage of the cheap messaging to access objects efficiently. MCRL [83] and Olden [84] are software systems that migrate computation to the chip that stores the data in its local memory in order to avoid the latency of off-chip memory accesses. In MCRL the decision to migrate is made dynamically by a runtime, while in Olden the decision is made statically by the compiler.

This project differs from previous work by augmenting an existing microarchitecture with the DMC instructions. The goal is to improve the performance of existing applications and runtimes with only a few minor modifications.

7.3 DMC hardware interface

We present the hardware interface for each DMC instruction, describe the semantics guaranteed by hardware, and give examples of how software might use each instruction. We assume a cache architecture with per-core L1 data caches and an inclusive L2 cache shared by all the cores. We think, however, that DMC instructions could be implemented for other architectures as well.

7.3.1 `cpush`

The `cpush` instruction takes two arguments: an address and a core ID. When a software thread executes `cpush address core-id` it is requesting that the hardware copy the contents of the cache line at `address` to the core identified by `core-id`. If, for some reason, hardware ignores the request, software correctness is not affected (similar to ignoring a prefetch instruction).

The outcome of executing `cpush address core-id` depends on the cache line state (modified, shared, or invalid) of `address` in the local L1 cache. The following list describes each outcome.

- If `address` is marked as shared in the local L1 then the cache controller copies the cache line to the destination cache and marks it as shared.
- If `address` is marked as modified in the local L1 then the cache controller invalidates the local cache line, copies the cache line to the destination cache, and marks the cache line as modified in the remote cache.
- If `address` is invalid in the local L1 then the cache controller ignores the request.

The processor pipeline does not wait for the cache controller to copy data between caches.

Software can use `cpush` to optimize inter-core communication of shared memory applications. If a thread running on one core knows it will need to to share recently accessed data with another core it can use `cpush` to move the data to the other core's

cache. The hope is that the data will arrive in the core's cache before the core tries to access it.

One example usage of `cpush` it to optimize thread migration in multicore runtimes, like MIT Cilk [85] or the Go programming language [86]. Multicore run-times migrate a thread by de-scheduling the thread off the source core, saving the values of the CPU registers in a thread context buffer, and adding the thread context buffer to the run-queue on the destination core, which will execute the thread.

The cost of migration is composed of cache miss penalties to transfer the thread context from one cache to another, and the cache miss penalties once a thread starts executing and accessing its working set. A multicore run-time could reduce both of these components using `cpush` to push the thread context and parts of the thread working set (*e.g.,* the top stack frames) from the source to the destination core before the source core adds the thread context to the destination core's run-queue. The cache controller will be transferring the thread context and the working set to the destination core while the source core is adding the thread context to the run-queue. The destination core can read the thread context buffer without incurring cache misses and once the thread begins executing it will be able to access parts of its working set without incurring cache misses.

7.3.2 `clookup`

The `clookup` instruction takes an address as an argument and returns the closest core that caches that address. The return value of executing `clookup address` depends on the cache line state in the local L1 cache and the L2 directory. The following list describes the return value based on the cache states.

- If `address` is marked as shared or modified in the source core's L1 then hardware returns the source core's core ID.
- If `address` is invalid in the local L1 and the L2 directory indicates the cache line is shared or modified in another core's L1 then the hardware returns the remote core's core ID.

- If the cache line is invalid in the sending core's L1 and invalid in the directory then hardware returns -1 to indicate that no core caches `address`.

`clookup` was originally designed to help test the implementation of `cmsg`. We think, however, that `clookup` might be useful in its own right. One challenge to building software run-times that manage cache contents is tracking which cores cache what data. Tracking data location in software is error prone, costly, and essentially duplicates the cache line state maintained by hardware. These systems could potentially replace their software data tracking schemes with `clookup`, which would be accurate and have lower overhead.

7.3.3 `cmsg`

The `cmsg` instruction is an implementation of active messages. Active messages [78] are an asynchronous communication mechanism. An active message contains a destination core ID and a function pointer which the destination core executes upon arrival of the message, passing the message body as arguments to the function. Instead of requiring software to provide a core ID, `cmsg` allows software to specify a memory address, which the cache controller resolves to a core ID. Specifically, `cmsg address, pc, body` causes the nearest core that caches address to start executing the function at `pc`, loads the contents of body into Special Purpose Registers (SPRs), and loads the source core's ID into an SPR. We refer to active messages sent in this manner as content addressable active messages. The DMC implementation also allows an application to specify a destination core directly using the core's ID, which is often useful for replying to a content addressed active message.

Hardware handles `cmsg address, pc, argument` in several ways depending on the cache line state in the source core's L1 cache and in the L2 directory.

- If the cache line is marked as shared or modified in the source core's L1 then hardware clears a "delivery" bit in the local condition register (CR) to indicate the message was not delivered.

- If the cache line is invalid in the local L1 and the L2 directory indicates the cache line is marked as shared or modified in another core's L1 then the hardware interrupts the other core as described below and sets the delivery bit in the source core's CR to indicate that the message was delivered.

- If the cache line is invalid in the sending core's L1 and invalid in the L2 directory then hardware clears the delivery bit in the source core's CR.

Hardware always delivers an active message when the application passes the destination core ID to `cmsg`.

If the L2 directory holds a suitable destination core, the source core sends a message containing `pc` and `body` to the destination core. When the message arrives at the destination core, the destination core loads `body` into SPRs and generates an interrupt, setting the program counter to `pc`. Similar to standard PowerPC interrupts, the destination core saves a small amount of execution state in Save/Restore Registers so that software can resume the execution before the interrupt.

`cmsg` provides a low-overhead mechanism for executing code on a remote core. If a thread running on one core needs to manipulate several cache lines in another core's cache, it can use `cmsg` to do so, instead of copying the cache lines into its local cache. A type of application where this might useful is one that creates many threads which operate on shared data structures. For example, the Linux kernel uses linked lists and other shared data structures to implement the physical page allocator, LRU page replacement, reverse page tables, and many other facilities. Adding to and removing from these linked lists often incurs several cache misses as the kernel updates linked list pointers and modifies subsystem specific shared meta-data.

Linux could reduce the number of cache misses by using `cmsg` to execute the list manipulation code on the core likely to cache the list. The address supplied to `cmsg` could be the address of the spin lock (or some other synchronization primitive) that the kernel uses to serialize updates to the list. Since the spin lock would always be acquired before updating the list, it is likely that if a core caches the address for the spin lock it will also cache the list meta-data. Using `cmsg`, updates to the list and

meta-data might avoid incurring cache misses.

7.4 DMC hardware design

We now discuss the microarchitecture design for implementing `cpush`, `clookup`, and `cmsg`, and highlight some important decisions for ensuring correctness and high performance.

We were able to augment the original PowerPC pipeline with the DMC instructions without making substantial revisions to the original design. The main reason for this is that executing `cpush`, `clookup`, and `cmsg` requires performing many of the same operations (*e.g.*, calculating the effective address) and state updates (*e.g.*, queuing a request to the L1 cache) required to execute a load or a store instruction.

The bulk of our redesign was centered on the L1 and L2 modules. The original PowerPC design implements a directory-based MSI protocol and uses request and response messages to communicate cache line state between the cores, the L1's, and the L2. To avoid deadlocks the original PowerPC cache coherence design enforces the invariant that requests do not block responses and that the L1 handles L2-to-L1 requests before handling pending core-to-L1 requests. In the original PowerPC cache coherence design each L2-to-L1 response had a matching L1-to-L2 request.

Figure 7-1 provides the state transitions for cache state, while Figure 7-2 provides the state transitions for directory state added to the original cache coherence protocol to support DMC.

7.4.1 Correctness of `cpush`

To work well with the original MSI implementation, the DMC PowerPC cache controller sends the cache line associated with a `cpush` in a response message. This design, however, breaks the invariant assumed in the original PowerPC implementation that L2-to-L1 response has an associated L1-to-L2 request. A potential bug might be that a core's L1 cache receives a response due to a push request from another core, but never processes the response. In this example the L1's cache state would

Current state	Request trigger	Dequeue trigger	Response trigger	Request from parent	Response to parent	Request to parent	Response from parent	Next state
M	Push, id	yes			Push, id, data			I
S	Push, id	yes			Push, id			S
I	Push, id	yes						I
I							M, data	M
I							S, data	S
(I,M)		no					S, data	(S,M)
M	MSnd, msg	yes						M
S	MSnd, msg	yes						S
I	MSnd, msg	yes			MSnd, msg		dlvrd/!dlvrd	I

Figure 7-1: Cache state transitions for DMC

Child's curr. state	Other children's curr. state	Trgr.	Deq. trgr.	Req. from child	Deq. req. from child	Resp. to child	Req. to child	Resp. from child	Req. to other children	Resp. to other children	Child's next state	Other children's next state
M	I							Push, id, data		M, data	I	M
M	I			M/S	yes						M	I
S	S							Push, id			S	S
S	I							Push, id		S, data	S	S
S	X = S/I			S	yes						S	X
I	I					!dlvrd		MSnd, msg			I	I
I	S					dlvrd		MSnd, msg		MSnd, msg	I	S
I	M					dlvrd		MSnd, msg		MSnd, msg	I	M

Figure 7-2: Directory state transitions for DMC

differ from the directory maintained by the L2.

Another tricky problem that arises with `cpush` is handling the case where a core requests a cache line that another core is simultaneously pushing. For example, if a core is waiting for a response to a request for a modified cache line and a response arrives due to a push request from another core that contains a shared copy of the cache line. If the L1 accepts the shared copy, but marks it as modified, the L1 state and L2 directory state would differ.

7.4.2 Performance of `cmsg`

One potential performance problem with interrupting execution to handle an active message is the cost of saving and restoring General Purpose Registers (GPRs) and setting up a PowerPC ABI compliant environment for executing C code. This process requires about 70 instructions.

To avoid saving and restoring execution state, we added a second register file. When a core receives a `cmsg`, it switches to the secondary register file, and switches back when the `cmsg` interrupt handler returns. It should be possible to ensure that the secondary register file is always in an ABI compatible state when the core switches to it. This solution precludes supporting nested `cmsg` interrupts, and requires an additional register file, which is quite expensive in terms of area. On the FPGA, however, the extra register file fits into a BRAM partially used by the original register file.

Another shortcoming is that the source core sends `cmsg` after receiving a reply from the L2 directory. In a more efficient implementation the L2 would send `cmsg` directly after performing the lookup, instead of replying to the source core.

7.5 Implementation

We now describe the hardware implementation of the DMC extension to Arete, the software implementation of the DMC run-time, and the tests we wrote for verification and benchmarking.

7.5.1 Hardware

The DMC PowerPC implementation adds about 250 lines and modifies about 750 lines of code in the original 7701 line PowerPC BSV code. Most of the modifications were to the L2 cache and the L1 data cache modules.

Working with a simulator written in BSV, and that runs on an FPGA has two advantages over software simulators. One is that adding DMC instructions actually requires modifying hardware, in contrast to software simulators. This allows us to gauge the complexity of adding the instructions to a real processor implementation and forces us to respect hardware constraints, such as limited on-chip storage and short critical paths. This is in contrast to software simulators which developers often extend using C or high-level languages like Python. Without the constraints of hardware, developers can implement overly simplistic or unrealistic designs.

A second advantage of running a design on an FPGA is that simulation is fast. The PowerPC model runs at 100 MHz and uses approximately 9 FPGA cycles to model 1 PowerPC cycle. Therefore, the simulated PowerPC runs at about 11.11 MHz. This is two orders of magnitude slower than a real PowerPC chip, but also two orders of magnitude faster than a cycle-accurate full system simulator written in C [87].

7.5.2 Software

The DMC run-time, which includes threads, a thread stealing scheduler, locks, a linked list implementation, and a memory allocator, is about 2000 lines of C code. DMC instructions are easy to use. Modifying code to use DMC instructions usually requires changing only a few lines of C code.

7.5.3 Testing

We tested the DMC hardware using a series of software stress tests. Much of our testing focused on `cpush`. One reason for this is that `cpush` is tricky to implement correctly because it modifies cache state, therefore we wanted to test it thoroughly. A

second reason is that we implemented `cpush` first and the implementation of `clookup` and `cmsg` reused much of the well tested `cpush` code.

Our tests for `cpush` try to trigger the corner cases described earlier. For example, to trigger the case where a core executes a `cpush` on a modified cache line while another core simultaneously requests a shared copy of the cache line, the test would create threads on different cores, one thread would spin in a loop incrementing a shared variable then calling `cpush`, while the other thread would spin and constantly read the value of the variable.

One challenge in testing `cpush` was to verify that executing `cpush` would cause a cache line to be copied into another core's cache. To verify that `cpush` was behaving as expected, we instrumented the L2 cache and the L1 data cache modules to print push requests and responses and inspected the output.

7.6 Evaluation

We evaluated the DMC PowerPC implementation by running Arete on the BEE3 board with the following configuration.

Tiles	1×
Cores	2×, in-order, 10-stage PowerPC
L1 I-cache	2×, private, 64 KB, 4-way set-associative, 64B blocks, 1 cycle pipelined hit latency
L1 D-cache	2×, private, 64 KB, 4-way set-associative, 64B blocks, 1 cycle pipelined hit latency
L2 cache	1×, shared, inclusive, 512 KB, 4-way set-associative, 64B blocks, 32 cycle pipelined hit latency
Main memory	1×, 1 GB, 256 cycle latency

We used three microbenchmarks, programmed to run with and without making use of `cpush` and `cmsg`, to measure performance. We chose one microbenchmark to measure how expensive it was to execute `cmsg`, and two others on the basis that they

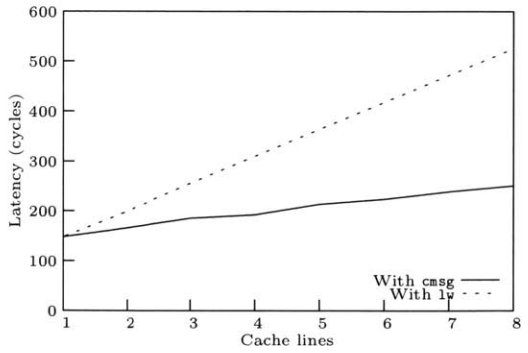


Figure 7-3: Results for the memory scan benchmark. The x-axis shows the number of cache lines in a segment and the y-axis shows the average latency to the read segment from another core's L1 cache.

represented operations implemented in complex applications. The performance measurements should be considered encouraging preliminary results. The DMC PowerPC lacks features found in advanced processors, such as out-of-order execution, hardware prefetching, and symmetric multi-threading, which would change performance.

7.6.1 Cost of `cmsg`

To understand the cost of executing a `cmsg` instruction we wrote a microbenchmark that compares the cost of reading cache lines from another core's cache to the cost of executing a `cmsg` to read the cache lines. The benchmark creates two threads. One thread fills its cache with shared cache lines by modifying every cache line in a 64 KB array. The second thread then reads the entire array in N cache line segments. The benchmark measures the average time to read one N cache line segment using `lw` and using `cmsg`. After executing `cmsg`, the thread spins in a loop until a flag variable is set to zero. The destination core sends replies using an active message, which clears the flag variable. The point at which using `cmsg` becomes cheaper than `lw` helps show when it might improve performance.

Figure 7-3 shows the cost of using `lw` and `cmsg`. The x-axis shows the number of cache lines in each segment and the y-axis shows the average latency to read each segment. The `cmsg` case always generates a pair of lookup request and response messages, one inter process active message to read the cache lines, and one inter

process active message to signal that the read is complete. The `lw` case generates a pair of request and response cache coherence messages for each cache line. Therefore, we expect the latency of the `lw` case to increase much faster than the `cmsg` case.

With 1 cache line, the use of `lw` or `cmsg` generates the same number of cache coherence requests and responses, and perform about the same. For two cache lines, the `cmsg` reduces the latency to read the cache lines by about 17%. As the benchmark manipulates more cache lines, `cmsg` provides more benefit. Using `cmsg` to access 8 cache lines is 52% faster than using `lw`. The cost of `cmsg` increases slightly as the set size increases because the benchmark must execute more instructions to read all the cache lines.

7.6.2 Thread migration

To evaluate the potential software performance improvement from using `cpush`, we wrote a microbenchmark that ping-pongs a thread between two cores and measures the average round-trip time. The benchmark uses two cores, one executes the migrating thread while the other spins in its scheduling idle loop, continuously checking for threads on its run-queue. To migrate the thread, the source core deschedules the thread, switches to another thread (the idle thread in this benchmark), which saves the core's registers in a context buffer, and adds the thread context buffer to the run-queue of the remote core. The idle core notices the new thread context on its run-queue, dequeues the context, and starts executing the thread by reloading the thread register values from the context buffer.

When the remote core loads the values of a thread's registers it usually incurs several cache misses, which increases the round-trip time. We use `cpush` to reduce the round-trip time by pushing the contents of the context buffer to the destination core before writing to the shared variable. This means that transferring the context buffer from one core to another will be overlapped with the operation of adding the context to the remote run-queue.

Figure 7-4 presents the results of the thread ping-pong microbenchmark. The x-axis shows the number of cache lines in the thread context that the benchmark

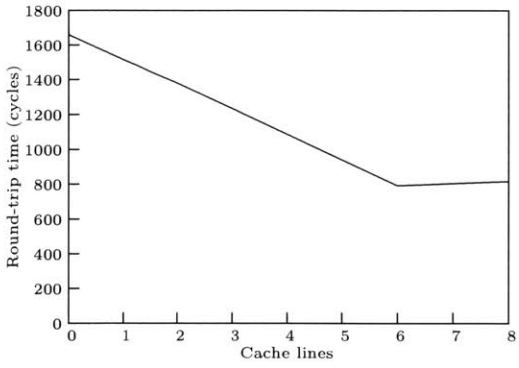


Figure 7-4: Results for the thread migration microbenchmark. The x-axis shows the number of cache lines the source core pushes to the destination core using `cpush`.

uses `cpush` to move from the source to the destination core. The y-axis measures the round-trip time in cycles to migrate a thread from the source to the destination and back.

Without using `cpush`, the round-trip time is about 2202 cycles. As the benchmark uses `cpush` to move more cache lines, the round-trip time reduces steadily until 6 cache lines, where the round-trip time is 831 cycles. Pushing more than 6 cache lines does not decrease the round-trip time further because the FIFOs connecting the pushing core's L1, the shared L2, and the destination core's L1 become full. In the current implementation the destination core stalls in this case. It would be correct, however, to simply drop the push request in the source L1 if the FIFO to the destination L1 is full.

7.6.3 Linked lists

Linked lists are commonly used to build more complex data structures. For example, the Linux kernel uses linked lists to implement the physical page allocator, LRU page replacement, reverse page tables, and many other facilities. The kernel usually maintains invariants, implemented with shared memory, and associated with complex data structure. When the kernel updates the underlying linked list, it also updates the other invariants. For example, when adding a virtual page to a reverse page table, the kernel acquires a lock, inserts the page into a list, and increments a per-page

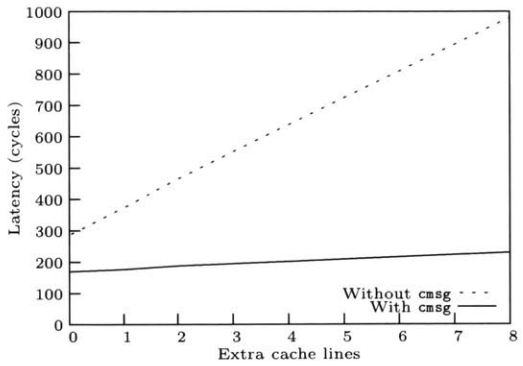


Figure 7-5: Results for the list microbenchmark

reference count.

We wrote a list microbenchmark to measure potential performance improvements from using `cmsg`. The list microbenchmark initializes a list by inserting 20 elements into the list, then creates two threads that insert into or remove from the list. To operate on the list, a thread acquires a spin lock protecting the list, performs insertion or removal with equal probability, and releases the lock. Performing an operation on the list can incur as many as 5 cache misses: acquiring the lock, setting the list entries next and previous pointers, setting the previous elements next pointer, setting the next elements previous pointer, and releasing the lock. To model the situations where software might update additional shared memory (*e.g.,* the reverse page table described above), the benchmark modifies a variable number of extra cache lines while holding the spin lock.

The list microbenchmark uses `cmsg` to perform the list operation. The microbenchmark uses the address of the spin lock to address the message, uses the address of the function performing the list operation as the PC, and uses the list element to insert or delete as the argument. After executing `cmsg` the thread spins until a flag variable is set to zero. The destination core replies using an active message, which clears the flag variable.

Figure 7-5 presents the results for the linked list microbenchmark. The x-axis shows the number of extra cache lines the benchmark modifies while holding the spin lock. The y-axis shows the average latency for executing a list operation.

The results indicate that, even when modifying one extra cache line, using `cmsg` decreases latency by about 22%. As the number of extra cache lines increases, the cost of performing a list operation increases with and without `cmsg`. Both increase because of the additional instructions the CPU must execute to modify the extra cache lines. However, the cost without `cmsg` increases much faster than the cost with `cmsg`, because every additional cache line modification incurs a cache miss. When using `cmsg`, on the other hand, the extra cache lines are most likely present in the destination core's L1 cache.

7.7 Summary

In this chapter we introduced DMC instructions for managing on-chip caches, and described their design and implementation on Arete. Results from mircobenchmarks indicate that using DMC instructions improves the performance of certain operations by reducing the number of cache misses. These results suggest that DMC instructions may be useful for a large class of workloads.

The use of a cycle-accurate, full-system simulator to conduct these experiments provided insight into the impact of the architectural changes on the system as a whole. Not only were we able to explore and understand the reasons behind the change in performance, we also gained an appreciation for the hardware constraints, namely, limited resources and short critical paths.

7.7.1 Limitations

One factor limiting the range of workloads that we can evaluate is the low core and cache count of our simulator. A dual-core processor model cannot evaluate how well DMC performs for workloads that take advantage of a large aggregate on-chip cache capacity by actively managing cache contents. Adding support for more cores would allow us to explore the use of DMC instructions in such workloads.

The implementation and evaluation of `cmsg` and active messaging has a number of loose ends. Currently, the use of `cmsg` assumes that the destination core is always

running in the same virtual address space as the thread executing `cmsg`. This assumption works when executing kernel functions in a kernel with a global virtual address space, like Linux, but it does not allow user-level threads to execute `cmsg`, because the destination core might be running in a different virtual address space. One potential solution is to include the value of the Process ID (PID) register, which serves as the core's TLB tag in the active message. Upon receiving the active message, the destination core can load the PID value.

Our evaluation of `cmsg` does not address how the operating system kernel can guarantee fairness. A core could spend all its time executing active messages from other cores, starving the threads on its own run-queue. It might be possible to detect this situation and either migrate all the threads to other cores, or mask active message interrupts for a short while.

We also need to evaluate the patterns of memory accesses applications make for which `cmsg` might hurt performance. For example, if an application reads some set of data objects very often, it might not be beneficial to use `cmsg` to accesses them. Instead, the cache coherence protocol should copy the objects into all the L1 caches.

Chapter 8

Conclusion

To conclude the thesis, we first discuss our modeling technique for developing FPGA-based cycle-accurate simulators. We then discuss the empirical evidence which shows that simplified models lead to wrong conclusions. Finally, we discuss the future directions that are based on our modeling technique and processor simulator.

8.1 Processor modeling on FPGAs

Architectural experimentation requires fast, flexible and accurate simulators. In the past few years, the technology for developing FPGA-based simulators has matured to the point where a flexible simulator, which can provide up to 1000× speedup over detailed software simulators, can be readily built. FPGA-based simulators, however, remain harder to develop than software simulators.

In this thesis we presented a new robust technique for developing cycle-accurate simulators on FPGAs. We showed how a cycle-level specification of the processor microarchitecture can be automatically transformed into an efficient FPGA-based model. The model, while preserving the timing behavior of the specification, can achieve both high performance and low resource utilization. A concrete evidence of the efficacy of our modeling technique is Arete. It runs at 55 MIPS when simulating 8 cores, and it has been modified for various architectural studies.

8.2 The need for cycle-accurate modeling

It is understandable why architects may want to use simplified models, such as the 1-IPC core model, in simulation studies. These simplified models not only improve simulation speed, if they turn out to be reliable, they can simplify the analysis of experimental results.

The empirical evidence provided in this thesis, however, showed that simplified models can lead to wrong conclusions. For example, if the replacement policy experiment was performed using the 1-IPC core model, one might have concluded that LRU was not much better than random. Similarly, if the branch prediction study was performed using the abstract model of stalls due to data hazards, one might have concluded that having a branch history table was not much better than having no branch prediction at all. The only way to ascertain if a simplified model is reliable is to compare it against a cycle-accurate model. Once validated, it can be used for many faster simulation runs.

Our goal is to point out the flaws in 1K-core simulations which use 1-IPC core models. Although, at present there may be no other feasible way to perform such large-scale simulations, our results show that performance estimates obtained from such simulations cannot be trusted.

8.3 Future work

In this section we present some of the future avenues we are planning to explore using our modeling methodology and cycle-accurate processor simulator.

8.3.1 Power modeling

Reasonably accurate power estimates of a hardware design obtained prior to tape out are as important, if not more, as its performance estimates. Tools and techniques for power modeling encounter the same set of challenges as those for performance modeling. To provide accurate estimates, commercial power modeling tools require

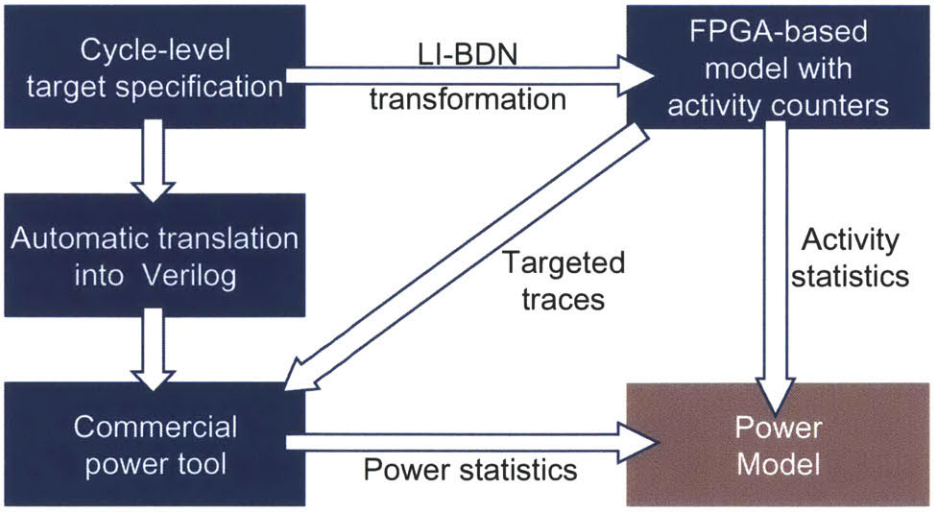


Figure 8-1: Power modeling approach

a placed and routed netlist of the design, and a value change dump (VCD) of a representative activation sequence, such as software running on a processor design. A placed and routed design is only available in the final stages of ASIC development, and generating a VCD for a placed and routed design is excruciatingly slow.

Another approach is to obtain activity statistics, such as register file reads and writes, TLB lookups and misses, *etc.*, from a cycle-accurate performance model of the target design. These statistics are then combined with power measurements of the respective events to estimate the power consumed by the entire design. The modeling methodology we developed and used to build Arete is specially suited to this approach. Figure 8-1 shows the proposed power modeling approach based on our modeling methodology. The cycle-level target specification is transformed into a refined LI-BDN implementation on FPGAs, from which accurate activity statistics are obtained at a very high speed. The specification is also compiled into RTL which is used to obtain power measurements for low-level events. The two are then fed into a power model which provides reasonably accurate estimates of overall power consumption.

Another interesting opportunity afforded by Arete is in low-level power gating of processor designs. Figure 8-2 shows the activity rate and the toggle rate for the 3 read and 2 write ports of the register file in the multicore PowerPC processor. Booting

Application	Read port 0		Read port 1		Read port 2		Write port 0		Write port 1	
	Activity	Toggle	Activity	Toggle	Activity	Toggle	Activity	Toggle	Activity	Toggle
linux	19%	38%	5%	23%	14%	10%	11%	4%	10%	8%
blackscholes	27%	51%	5%	33%	16%	13%	14%	4%	14%	11%
swaptions	26%	51%	6%	33%	15%	13%	14%	4%	14%	11%
streamcluster	26%	50%	6%	33%	15%	12%	14%	4%	13%	11%
canneal	27%	52%	6%	34%	16%	13%	13%	4%	14%	10%

Figure 8-2: Statistics for register file ports

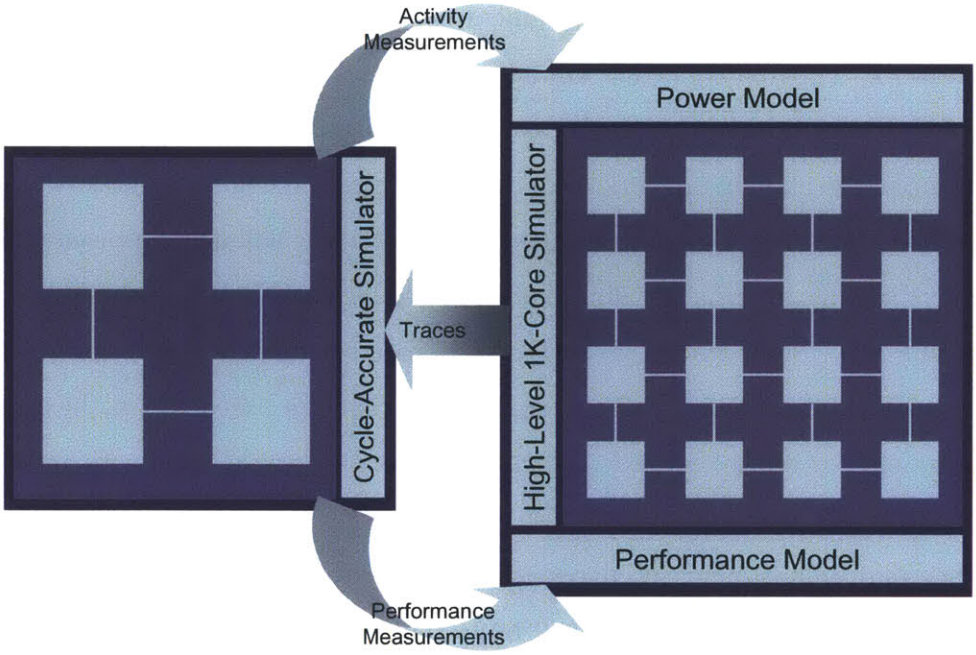


Figure 8-3: Combining Arete with large-scale simulators like Graphite

of the Linux kernel and execution of four applications from the PARSEC benchmark suite were used to obtain these statistics. We see that all the ports have low to moderate activity as well toggling, which makes them suitable candidates for power gating. Such insight can be obtained for all the low-level events in a processor.

8.3.2 Combining moderate-scale cycle-accurate simulations with large-scale functional simulations

We propose to combine the large-scale simulation and high-level power-performance modeling capabilities of Graphite [34] with the cycle-accurate activity and performance measurement capability of Arete [3] to provide a simulation infrastructure that delivers the best of both worlds. Figure 8-3 provides a high-level view of the proposed modeling approach and the information flow between the two simulation platforms. Accomplishing such a merger gives rise to two research challenges: restriction and projection.

Arete will play the role of a fast and accurate partial-system simulator. For

instance, it will simulate a 64-core segment of the 1K-core processor, or only the on-chip network. The first challenge will be to generate a restricted cycle-level trace from Graphite that will drive a particular partial-system simulation. In order to simulate, for example, the on-chip network, the trace will need to include all the network traffic generated by the various nodes. Furthermore, this traffic will need to be coupled with cycle-level timing information obtained from the high-level simulation. Similarly, for simulating a 64-core segment of the 1K-core processor, the trace will need to include the code and data segments of the application that are executed on the particular core segment. It will also need to include timed memory and network interactions between the core segment and the rest of the system.

Once performance and activity statistics are gathered from many partial-system simulations, the second challenge will be to develop models that are capable of projecting full-system power and performance from these statistics. In case of a homogeneous multicore processor running homogeneous parallel applications, the projection problem can be reduced to a straight-forward extrapolation. However, when either the processor or the application exhibits heterogeneity, a careful partitioning of the system will be required, along with many partial-system simulations. Statistics obtained from these simulations will then be combined into full-system estimates. A more complex processor microarchitecture or application behavior will translate into both a higher frequency of partial-system simulations and a more involved combinatorics problem.

8.3.3 Hardware/software codesign

To satisfy power and performance constraints, programs running on system-on-chip (SoC) platforms exploit an increasingly wide range of special-purpose accelerators. Generally, these are implemented as fixed-function ASIC blocks and are connected to the application processor by an interconnection network. Choosing which accelerators to implement in hardware is a risky undertaking, since it requires chip designers to identify the important applications a priori. Moreover, implementing applications on such platforms requires the programmer to use the fixed functionality effectively.

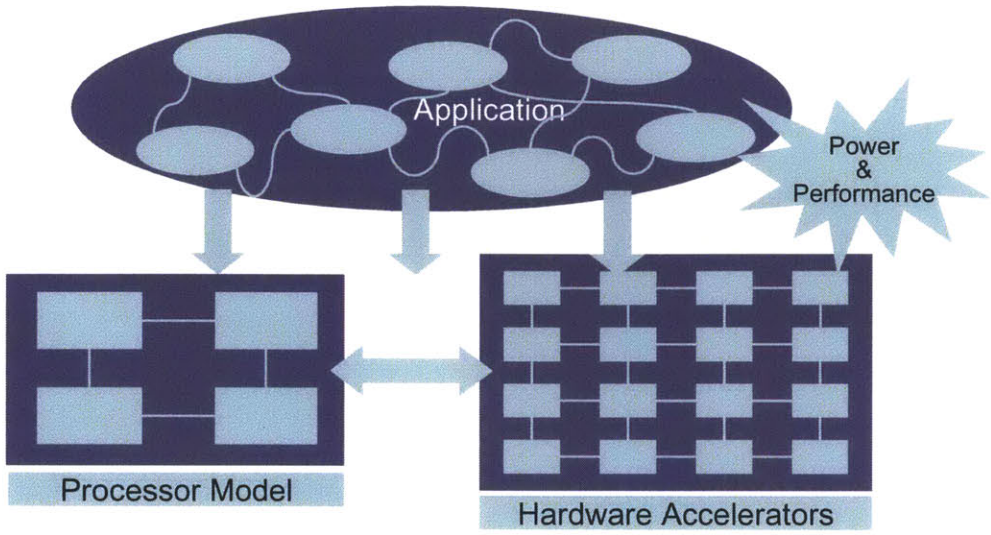


Figure 8-4: Hardware/software codesign on Arete

Our modeling methodology facilitates the addition of hardware accelerates to Arete, providing a cycle-accurate power-performance modeling platform for SoCs, as shown in Figure 8-4. We can accurately model the relative clock speeds of the processor, the accelerators and the communication channel. For example, in order to model a clock speed ratio between the processor and an accelerator of $1:10$, we will set the FPGA to model cycle ratio (FMR) for the accelerator to 90, since the FMR for the processor is 9. The latency and bandwidth of the communication channel can be modeled in a similar fashion.

Bibliography

[1] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe, "Smarts: accelerating microarchitecture simulation via rigorous statistical sampling," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, 2003, pp. 84–95.

[2] R. M. Fujimoto, "Parallel discrete event simulation," *Commun. ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.

[3] A. Khan, M. Vijayaraghavan, S. Boyd-Wickizer, and Arvind, "Fast and Cycle-Accurate Modeling of a Multicore Processor," in *ISPASS '12: Proceedings of the International Symposium on Performance Analysis of Systems and software*, April 2012.

[4] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-Port Networks: Preserving the Timed Behavior of Synchronous Systems for Modeling on FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 3, pp. 1–26, 2009.

[5] M. Vijayaraghavan and Arvind, "Bounded Dataflow Networks and Latency-Insensitive circuits," in *MEMOCODE'09: Proceedings of the 7th IEEE/ACM International Conference on Formal Methods and Models for Codesign*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 171–180.

[6] A. Khan, M. Vijayaraghavan, and Arvind, "A general technique for deterministic model-cycle-level debugging," in *Formal Methods and Models for Codesign (MEMOCODE), 2012 10th IEEE/ACM International Conference on*, july 2012, pp. 109 –118.

[7] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 2, no. 2, pp. 1–32, 2009.

[8] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "Quick Performance Models Quickly: Closely-Coupled Partitioned Simulation on FPGAs," in *ISPASS '08: Proceedings of the International Symposium on Performance Analysis of Systems and software*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 1–10.

[9] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 249–261.

[10] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanovic, "RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors," in *DAC '10: Proceedings of the 47th Annual Design Automation Conference*, 2010, pp. 463–468.

[11] L. Carloni, K. McMillan, and A. Sangiovanni-Vincentelli, "Theory of latency-insensitive design," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 20, no. 9, pp. 1059–1076, Sep 2001.

[12] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *MEMOCODE'09: Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 171–180.

[13] S. Krstic, J. Cortadella, M. Kishinevsky, and J. O'Leary, "Synchronous elastic networks," in *Formal Methods in Computer Aided Design, 2006. FMCAD '06*, Nov. 2006, pp. 19–30.

[14] T. Harris, Z. Ruan, and D. Penry, "Techniques for LI-BDN synthesis for hybrid microarchitectural simulation," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, oct. 2011, pp. 253 –260.

[15] K. E. Fleming, M. Adler, M. Pellauer, A. Parashar, A. Mithal, and J. Emer, "Leveraging latency-insensitivity to ease multiple fpga design," in *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, ser. FPGA '12. New York, NY, USA: ACM, 2012, pp. 175–184.

[16] C. Bienia and K. Li, "PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[17] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," *SIGARCH Comput. Archit. News*, vol. 23, pp. 24–36, May 1995.

[18] *Power ISA Version 2.05*, IBM, October 2007.

[19] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, June 2004, pp. 69 – 70.

[20] Buildroot: making Embedded Linux easy. [Online]. Available: `http://buildroot.uclibc.org/`

[21] N. Wells, "BusyBox: A Swiss Army knife for Linux," *Linux Journal*, november 2000.

[22] H. K.-H. So and R. Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *Transactions on Embedded Computing Systems*, vol. 7, no. 2, pp. 1–28, 2008.

[23] A. Parashar, M. Adler, K. E. Fleming, M. Pellauer, and J. Emer, "LEAP: A Virtual Platform Architecture for FPGAs," in *The First Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*, ser. CARL '10, Atlanta, GA, USA, 2010.

[24] J. Davis, C. Thacker, and C. Chang, "BEE3: Revitalizing computer architecture research," *Technical Report MSR-TR-2009-45*, April 2009.

[25] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve, "Rsim: Simulating Shared-Memory Multiprocessors with ILP Processors," *IEEE Computer*, pp. 40–49, 2002.

[26] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *IEEE Computer*, pp. 50–58, 2002.

[27] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacets General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, September 2005.

[28] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, pp. 52–60, July 2006.

[29] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, April 2009.

[30] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega, "COTSon: infrastructure for full system simulation," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 52–61, January 2009.

[31] R. Bedichek, "SimNow: Fast Platform Simulation Purely In Software," in *HotChips 16*, August 2004.

[32] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," *Journal of VLSI Signal Processing*, pp. 169–182, 2005.

[33] G. Zheng, G. Kakulapati, and L. Kale, "BigSim: a parallel simulator for performance prediction of extremely large parallel machines," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, April 2004.

[34] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A Distributed Parallel Simulator for Multicores," in *HPCA-16: Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, January 2010.

[35] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proceedings of the 40th annual International Symposium in Computer Architecture (ISCA-40)*, June 2013.

[36] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. Hoe, D. Chiou, and K. Asanovic, "Ramp: Research accelerator for multiple processors," *Micro, IEEE*, vol. 27, no. 2, pp. 46–57, 2007.

[37] G. Gibeling, A. Schultz, and K. Asanovic, "RAMP Architecture and Description Language," in *2nd Workshop on Architecture Research using FPGA Platforms*, February 2006.

[38] D. Chiou, H. Angepat, N. A. Patil, and D. Sunwoo, "Accurate Functional-First Multicore Simulators," *Computer Architecture Letters*, July 2009.

[39] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX 2005 Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[40] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in *Proceedings of the 17th International Symposium on High-Performance Computer Architecture*, February 2011, pp. 406 –417.

[41] D. Penry, D. Fay, D. Hodgdon, R. Wells, G. Schelle, D. August, and D. Connors, "Exploiting parallelism and structure to accelerate the simulation of chip multiprocessors," in *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, 2006, pp. 29–40.

[42] A. Krasnov, A. Schultz, J.Wawrzynek, G. Gibeling, and P. Droz, "RAMP Blue: A Message-Passing Manycore System In FPGAs," in *Proceedings of International Conference on Field Programmable Logic and Applications*, 2007, pp. 27–29.

[43] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun, "A practical fpga-based framework for novel cmp research," in *Proceedings of the 2007 ACM/SIGDA 15th international symposium on Field programmable gate arrays*, ser. FPGA '07. New York, NY, USA: ACM, 2007, pp. 116–125.

[44] C. Thacker, "Beehive: A many-core computer for fpgas (v5)," *MSR Silicon Valley*, 2010.

[45] M. A. Kinsy, M. Pellauer, and S. Devadas, "Heracles: Fully Synthesizable Parameterized MIPS-Based Multicore System," *International Conference on Field Programmable Logic and Applications*, pp. 356–362, 2011.

[46] N. Sonmez, O. Arcas, G. Sayilar, O. S. Unsal, A. Cristal, I. Hur, S. Singh, and M. Valero, "From Plasma to BeeFarm: Design Experience of an FPGA-based Multicore Prototype," in *Proceedings of the 7th International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, 2011, pp. 350–362.

[47] P. G. Del Valle, D. Atienza, I. Magan, J. G. Flores, E. A. Perez, J. M. Mendias, L. Benini, and G. D. Micheli, "A Complete Multi-Processor System-on-Chip FPGA-Based Emulation Framework," in *Proceedings of the IFIP Conference*, 2006, pp. 140–145.

[48] M. D. Nava, P. Blouet, P. Teninge, M. Coppola, and T. Ben-Ismail, "An Open Platform for Developing Multiprocessor SoCs," *IEEE Computer*, pp. 60–67, July 2005.

[49] D. F. Bacon and S. C. Goldstein, "Hardware-Assisted Replay of Multiprocessor Programs," *SIGPLAN Not.*, vol. 26, no. 12, pp. 194–206, Dec. 1991.

[50] J.-D. Choi and H. Srinivasan, "Deterministic Replay of Java Multithreaded Applications," in *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools*, ser. SPDT '98. New York, NY, USA: ACM, 1998, pp. 48–59.

[51] M. Xu, R. Bodik, and M. Hill, "A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, june 2003, pp. 122 – 133.

[52] G. Altekar and I. Stoica, "ODR: Output-Deterministic Replay for Multicore Debugging," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 193–206.

[53] *ChipScope Pro - Software and Cores User Guide Version 13.4*, Xilinx, January 2012.

[54] *Identify - Simulator-like Visibility into Hardware Debug*, Synopsys, 2011.

[55] P. Graham, B. Nelson, and B. Hutchings, "Instrumenting Bitstreams for Debugging FPGA Circuits," in *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, April 2001, pp. 41 –50.

[56] K. Camera, H. K.-H. So, and R. W. Brodersen, "An integrated debugging environment for reprogrammble hardware systems," in *Proceedings of the sixth international symposium on Automated analysis-driven debugging*, ser. AADEBUG'05. New York, NY, USA: ACM, 2005, pp. 111–116.

[57] http://www.ultrasoc.com/.

[58] http://www.arm.com/products/system-ip/coresight/index.php.

[59] *emVM User Manual*, Bluespec, January 2012.

[60] R. Desikan, D. Burger, and S. W. Keckler, "Measuring Experimental Error in Microprocessor Simulation," in *ISCA '01: Proceedings of the 28th Annual International Symposium on Computer Architecture*, 2001.

[61] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti, "Precise and Accurate Processor Simulation," in *Fifth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2002.

[62] P. Bose and T. M. Conte, "Performance Analysis and Its Impact on Design," *Computer*, vol. 31, no. 5, pp. 41–49, 1998.

[63] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins, "Characterizing and Comparing Prevailing Simulation Techniques," in *11th International Symposium on High Performance Computer Architecture*, January 2005.

[64] B. Black and J. P. Shen, "Calibration of Microprocessor Performance Models," *Computer*, vol. 31, no. 5, pp. 59–65, 1998.

[65] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai, "Challenges in Computer Architecture Evaluation," *Computer*, pp. 30–36, 2003.

[66] A. R. Alameldeen and D. A. Wood, "Variability in Architectural Simulations of Multi-Threaded Workloads," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, ser. HPCA '03. Washington, DC, USA: IEEE Computer Society, 2003.

[67] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81.

[68] M. Bhadauria, V. M. Weaver, and S. A. McKee, "Understanding parsec performance on contemporary cmps," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 98–107.

[69] C. Bienia, S. Kumar, and K. Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, 2008, pp. 47–56.

[70] J. Yi and D. Lilja, "Simulation of computer architectures: simulators, benchmarks, methodologies, and recommendations," *Computers, IEEE Transactions on*, vol. 55, no. 3, pp. 268–280, March.

[71] J. Gibson, R. Kunz, D. Ofelt, M. Horowitz, J. Hennessy, and M. Heinrich, "FLASH vs. (Simulated) FLASH: closing the simulation loop," *SIGARCH Comput. Archit. News*, vol. 28, no. 5, pp. 49–58, Nov. 2000.

[72] M. Moudgill, P. Bose, and J. Moreno, "Validation of Turandot, a fast processor model for microarchitecture exploration," in *Performance, Computing and Communications Conference, 1999 IEEE International*, Feb, pp. 451–457.

[73] J. J. Yi, D. J. Lilja, and D. M. Hawkins, "A Statistically Rigorous Approach for Improving Simulation Methodology," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, ser. HPCA '03. Washington, DC, USA: IEEE Computer Society, 2003.

[74] A. Alameldeen and D. Wood, "IPC Considered Harmful for Multiprocessor Workloads," *Micro, IEEE*, pp. 8 –17, july-aug. 2006.

[75] S. Boyd-Wickizer, R. Morris, and M. F. Kaashoek, "Reinventing scheduling for multicore systems," in *Proceedings of the 12th conference on Hot topics in operating systems*, ser. HotOS'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 21–21.

[76] M. Kamruzzaman, S. Swanson, and D. M. Tullsen, "Software data spreading: leveraging distributed caches to improve single thread performance," *SIGPLAN Not.*, vol. 45, no. 6, pp. 460–470, Jun. 2010.

[77] R. Strong, J. Mudigonda, J. C. Mogul, N. Binkert, and D. Tullsen, "Fast switching of threads between cores," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 35–45, Apr. 2009.

[78] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 1992, pp. 256–266.

[79] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, "Corey: an operating system for many cores," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 43–57.

[80] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new os architecture for scalable multicore systems," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 29–44.

[81] D. Wentzlaff and A. Agarwal, "Factored operating systems (fos): the case for a scalable operating system for multicores," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 76–85, Apr. 2009.

[82] W. J. Dally, L. Chao, A. Chien, S. Hassoun, W. Horwat, J. Kaplan, P. Song, B. Totty, and S. Wills, "Architecture of a message-driven processor," in *25 years of the international symposia on Computer architecture (selected papers)*, ser. ISCA '98. New York, NY, USA: ACM, 1998, pp. 337–344.

[83] W. C. Hsieh, M. F. Kaashoek, and W. E. Weihl, "Dynamic computation migration in dsm systems," in *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, ser. Supercomputing '96. Washington, DC, USA: IEEE Computer Society, 1996.

[84] M. C. Carlisle and A. Rogers, "Software caching and computation migration in olden," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPOPP '95. New York, NY, USA: ACM, 1995, pp. 29–38.

[85] The Cilk Project. [Online]. Available: http://supertech.csail.mit.edu/cilk/

[86] The Go Programming Language. [Online]. Available: http://golang.org/

[87] P. Bohrer, M. Elnozahy, A. Gheith, C. Lefurgy, T. Nakra, J. Peterson, R. Rajamony, R. Rockhold, H. Shafi, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang, "Mambo a full system simulator for the powerpc architecture," *ACM SIGMETRICS Performance Evaluation Review*, 2004.