Speeding Up Architectural Simulation through High-Level Core Abstractions and Sampling

Versnellen van architecturale simulatie
via abstracte prestatiemodellen en bemonstering

Trevor Carlson

Promotor: prof. dr. ir. L. Eeckhout
Proefschrift ingediend tot het behalen van de graad van
Doctor in de Ingenieurswetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Ingenieurswetenschappen en Architectuur
Academiejaar 2013 - 2014

UNIVERSITEIT
GENT

ISBN 978-90-8578-698-6
NUR 958
Wettelijk depot: D/2014/10.500/44

Acknowledgements

Over the years, I have always been very interested in discovering new things. When I was a child, I assumed that I could build electronics that did fancy things just by drawing a schematic on paper. I enjoyed thinking about devices that I could build that would change the way that things worked. Throughout my career, I found that discovering new and interesting ideas and directions was my true passion. I am thankful that both my mother and father have always been supportive of me, allowed me to be creative, and allowed me to make mistakes. During my younger years at school, my Junior High Technology teacher, David Rockett, gave me the freedom to experiment with new ideas, program robots and build trains that used magnetic levitation. Later on, when working for IBM, I realized that I still have this passion and I worked to file four patents as a side-project to my normal work, not even dreaming at the time that it would allow me to start a path towards completing a Ph.D. Solving difficult problems allowed me to think in new ways and better understand my personal limitations. Of course, I wanted to move past these limitations at each chance that I could get, but I also understood that hard-work was required. I will never be done proving myself, and I will always be challenged to accomplish something that others might feel is too difficult to solve.

My Ph.D. would not have been possible if it was not for my original manager at IMEC, Bjorn De Sutter, who saw that I could make the transition from industry to academia. My advisor, Lieven Eeckhout was instrumental in helping me better understand how to become a better researcher. Lieven's advice, support and constant encouragement to do even better have helped me learn how to conduct research in an academic setting while learning from experts in the field. Without his guidance and insight, I would not be where I am today.

Looking back at the time that I spent at the university, I was very happy that two things occurred at just the right time. First, the founding of the ExaScience Lab gave my Ph.D. a new direction. The focus of the lab allowed me to zero in on the needs of High Performance Computing, both from a hardware and software perspective. To accomplish this, we felt that faster simulation methodologies could help us improve the state of the art.

At the same time, Wim Heirman joined our group to help us realize our goal of faster simulation. From one of our very first meetings, I saw that our collaboration had great promise. We sat down to sketch out the ideas for what would become general-purpose time-based multi-threaded sampling, published two years and tens of thousands of simulations later. At the time, I did not realize that it was going to be both an enjoyable and fruitful period, both professionally and personally. Together with Wim's help and guidance, we have been able to accomplish much more than I had envisioned.

My wife, Stephanie, took a chance on me as I moved back into academia and I started my Ph.D.; I had already been working for more than 6 years in industry. While difficult, she saw that it could be a rewarding experience if I could stay focused, complete all tasks early, and above all, enjoy the work that I was doing. She has been there every step of the way, making delicious baked goods, helping to proof-read articles, and supporting me through the many difficult times where it appeared that no hope was in sight. Thankfully, I have enjoyed every minute of my Ph.D., and only hope I will continue to be challenged throughout life. I might not have been the one who accomplished everything early, but I do feel that I have been able to accomplish everything, if not more, that I set out to do.

Trevor E. Carlson
Ghent, June 11th, 2014

Examination Committee

Prof. Jan Van Campenhout, voorzitter
Vakgroup ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Lieven Eeckhout, promotor
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Prof. Koen De Bosschere
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Stijn Eyerman, secretaris
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Ayose Falcon
Intel Barcelona Research Center
Spain

Prof. Jan Fostier
Vakgroep INTEC, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Wim Heirman
Intel ExaScience Lab
Leuven

Prof. Thomas Wenisch
University of Michigan, Advanced Computer Architecture Lab
USA

Reading Committee

Dr. Stijn Eyerman
Vakgroep ELIS, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Ayose Falcon
Intel Barcelona Research Center
Spain

Prof. Jan Fostier
Vakgroep INTEC, Faculteit Ingenieurswetenschappen en Architectuur
Universiteit Gent

Dr. Wim Heirman
Intel ExaScience Lab
Leuven

Prof. Thomas Wenisch
University of Michigan, Advanced Computer Architecture Lab
USA

Summary

Both Moore's Law and Dennard Scaling have allowed us to make large improvements in chip density, energy efficiency and performance over the last 20 to 30 years of microprocessor design. Moore's Law predicts that the number of transistors in a processor will double every two years, and Dennard Scaling states that power density remains constant because of transistor scaling. While Dennard Scaling is seen as now coming to an end, Moore's Law, by predicting the continued growth in the number of available transistors, continues to provide us with opportunities to enhance the speed and efficiency of future microprocessors.

Through Pollack's Rule, we see that a single, large processor will not allow us to improve performance without a large increase in the complexity of current CPU core designs. Pollack's Rule states that the performance due to microarchitectural advancements improves with the square root of the complexity, or area used. This means that using a larger and larger area for a single, monolithic processor results in diminishing returns with respect to improved processor performance. The multi- and many-core era has allowed us to continue to scale the performance of our platforms through the use of many efficient cores instead of a single, less efficient one. On the horizon, we expect this trend to continue with systems utilizing larger numbers of cores. Higher processing requirements go hand-in-hand with larger on-chip caches that are required to sustain processor bandwidth requirements. Processor pin density limitations make it very difficult to increase the bandwidth to the processor, requiring larger caches to reduce bandwidth requirements, maintaining high performance.

These processor trends directly affect the ability to simulate improved architecture designs on today's hardware. As core complexity, core count and on-chip cache size increase, so do the simulation requirements needed to evaluate the performance of these microarchitectures. This is because both greater numbers of cores and more complex cores require additional modeling steps (and therefore additional simulation time) to evaluate. Larger caches tend to translate into longer simulation times as well, because they can require higher instruction counts to exercise the entire cache. Unfortunately, while Pollack's Rule has shown us a path to improve application performance through multi-core microarchitectures, simulation design

technologies have not adapted to the changing environment. Typical microarchitecture simulators are single-threaded and therefore will not continue to improve their performance with newer microarchitectures. This situation presents an increasing simulation gap, where simulators will not be able to continue to scale with microarchitectural improvements while the requirements for simulation will continue to grow.

These growing simulation requirements and the simulation gap require unique solutions to allow architects and designers to continue to advance the state of the art in microarchitectural design. In this work, we propose a number of approaches to directly counter these trends by reducing the simulation time required for design exploration. We do this through two orthogonal approaches. First, we improve the speed of the simulator directly, allowing for the same amount of work to be done in a shorter time. We do this by using high-level abstraction models to provide faster simulation speeds. Second, through application sampling, we reduce the application into a smaller, representative sample, reducing the amount of work that the simulator needs to perform.

Speeding-up Simulation While detailed cycle-accurate simulation methodologies have been used for both discovering and validating microarchitectural improvements, changing the simulator to use higher abstraction levels for microprocessor simulation is one way to alleviate the increasing simulation demands resulting from Moore's Law. In this work, we developed a new microarchitectural simulator, the Sniper Multi-Core Simulator, to show that higher abstraction levels, with respect to core, cache and memory hierarchy and simulation infrastructure models, can provide good accuracy with higher simulation speeds. As we demonstrate, naive, simple core models can produce inaccurate results, both from an absolute and relative perspective. But using more intelligent, mechanistic high-level models can result in high accuracy at very high speeds. More specifically, we demonstrate that high-level abstractions can accurately predict both performance and trends, with average errors starting at 11.1% with aggregate speeds of up to 2.0 MIPS. In addition, we show that while very simple models, such as the one-IPC core simulation models, provide faster simulator performance, they do not necessarily allow one to draw the same conclusions as both the original hardware and more accurate models provide. We find core models that honor both instruction-level parallelism (ILP) and memory-level parallelism (MLP), such as those used in interval simulation, allow for more accurate simulation results.

In addition to interval simulation, we want to allow architects to have access to additional points in the accuracy vs. simulation speed trade-off curve where improved accuracy can help for additional refinement as necessary for late-phase validation. We first enhance the original interval sim-

ulation model with features such as issue contention modeling to enable higher accuracy with minimal slowdown. We show that these features can get us closer to a more accurate simulation compared to hardware. In addition to interval simulation improvements, we present a novel processor core model called the instruction-window centric model, that allows for fine-grained analysis and timing control with a modest slowdown in simulation speed. The instruction-window centric processor core models improve average error from 24.3% to 11.1% for single-core models. Finally, we demonstrate the speed-versus-accuracy trade-offs across a number of benchmarks to provide a detailed overview of how these different models can provide the architect with different options for exploration or detailed analysis.

Application Reduction through Sampling There are two orthogonal approaches to speeding up simulation. The first approach, discussed in the previous paragraph, speeds up the simulator itself, providing a faster result given a fixed amount of work. Sampling is a well-known technique that is orthogonal to speeding up the simulator itself, and is used to reduce the amount of work to be the simulated. This is done by taking advantage of inherent application similarity that occurs naturally in most applications because of looping and other common programming constructs. Single-threaded application sampling has been studied extensively and is considered to be mature. Nevertheless, multi-threaded application sampling poses new challenges. Now, thread interaction and timing needs to be taken into account, either in shared microarchitectural structures, like shared caches, or when looking at thread ordering and other concerns caused by dynamic scheduling, such as through work-stealing.

In this thesis, we present two advances in the field of multi-threaded application sampling. First, we present a generic time-based sampling approach that solves many of the issues inherent to multi-threaded application sampling for synchronizing threads. We show that taking into account per-thread IPCs as well as properly handling inter-thread interactions during fast-forwarding significantly increases sampling accuracy. In addition, we show that application phase behavior needs to be taken into account when selecting appropriate regions. With our methodology, we simulate less than 10% of the application in detail, while maintaining a low average error of 3.5%.

In addition to the general case of multi-threaded sampled simulation, we improve sampled simulation for an important subset of multi-threaded applications, namely, barrier-based applications. We propose new methods for the identification and characterization of inter-barrier regions for the purpose of application similarity analysis and reduction. Using these new similarity methods, we propose a methodology for clustering regions

to allow one to select corresponding representative regions of interest. Finally, we evaluate this methodology on simulated 8- and 32-core machines and find an average reduction of resources by 78×. We demonstrate low average errors, of 0.9%, with a speedup factor of 24.7× on average and a maximum of 866.6×.

Samenvatting

Dankzij de wet van Moore en Dennards schalingswet zijn we erin geslaagd grote verbeteringen in de chipdichtheid, energie-efficiëntie en prestatie te bereiken. Volgens de wet van Moore verdubbelt het aantal transistors op een chip elke twee jaar; Dennards schalingswet stelt dat de vermogendichtheid constant blijft vanwege transistorschaling. Terwijl Dennards schalingswet ten einde loopt, blijft de wet van Moore geldig, wat tot een bijzonder grote uitdaging leidt teneinde de prestatie en efficiëntie van toekomstige processors blijvend te verbeteren.

Op basis van de wet van Pollack, die stelt dat de complexiteit kwadratisch toeneemt als functie van de beoogde prestatieverbetering, zien we dat een grotere processor ons niet zal toelaten de prestatie verder te verbeteren zonder een aanzienlijke toename in complexiteit. De wet van Pollack, in combinatie met de wet van Moore en het einde van de schaling volgens Dennards schalingswet, heeft geleid tot het multi-core tijdperk waarbij meerdere processorkernen samen geïntegreerd worden op één chip, en waarbij software geparallelliseerd dient te worden teneinde prestatieverbetering te bereiken. Er wordt algemeen verwacht dat het aantal processorkernen per chip zal blijven toenemen in de nabije toekomst. Een stijgend aantal processorkernen in combinatie met een beperkt aantal externe pinnen, leidt tot steeds grotere caches teneinde de processorkernen tijdig van data te kunnen voorzien gezien de beperkte geheugenbandbreedte.

Deze trends in processorontwerp hebben een grote invloed op het ontwerpproces en meer specifiek de simulatie van toekomstige processors. Een stijgend aantal processorkernen en steeds grotere caches verhogen de complexiteit van de simulator, en hebben derhalve een negatieve impact op de simulatietijd. Bovendien zijn de meeste hedendaagse simulators enkeldradig waardoor de simulatiesnelheid niet aanzienlijk zal toenemen in de komende jaren. Dit leidt tot een simulatiekloof: simulators schalen niet mee met de architecturen die zij dienen te simuleren.

In dit werk stellen we een aantal oplossingen voor voor het versnellen van de architecturale simulatie van multicore processors. We doen dit op twee manieren. Eerst verbeteren we de simulatiesnelheid door het abstractieniveau van modellering te verhogen. Ten tweede reduceren we de te simuleren werklast m.b.v. bemonstering.

Simulatieversnelling. In dit werk beschrijven we Sniper, een parallelle architecturale simulator die de simulatie aanzienlijk versnelt door het modelleren van de processorkernen op een hoger abstractieniveau, zonder aan al te veel nauwkeurigheid in te boeten. We gebruiken hiertoe een analytisch, mechanistisch prestatiemodel van de processorkern, dat leidt tot een hoge nauwkeurigheid en hoge simulatiesnelheden. Onze experimentele resultaten geven aan dat hoog-niveau prestatiemodellen een gemiddelde afwijking kunnen behalen van 11% ten opzichte van echte hardware, en dit aan een simulatiesnelheid tot 2 MIPS (miljoen instructies per seconde)—2 tot 3 grootteordes sneller dan gedetailleerde architecturale simulatie. Bovendien tonen wij aan dat ofschoon heel eenvoudige modellen, zoals het veel gebruikte 'one-IPC' processorkernsimulatiemodel, sneller zijn, zij niet de benodigde nauwkeurigheid behalen. Nauwkeurige simulatie vereist de modellering van instructie-niveauparallellisme (ILP) en geheugen-niveauparallellisme (MLP), zoals aangeleverd in het intervalmodel. Naast verbeteringen tot het intervalmodel teneinde de structurele hazards nauwkeuriger te modelleren, presenteren we eveneens een nieuw processorkernmodel, namelijk het instructievenstersimulatiemodel, dat de prestatie van de processorkern modelleert door enkel te focussen op de werking van het instructievenster, de centrale eenheid van een hedendaagse superscalaire processor. Het eindresultaat van deze studie is een aantal hoog-niveau simulatiemodellen met een waaier aan mogelijkheden qua nauwkeurigheid versus simulatiesnelheid.

Werklast Reductie door Bemonstering. Bemonstering is een gekende techniek om de te simuleren werklast te reduceren, teneinde op die manier de simulatietijd in te korten. Terwijl bemonstering van enkeldradige applicaties een mature en veelvuldig gebruikte techniek is, is de bemonstering van meerdradige applicaties een openstaand onderzoeksvraagstuk. Bemonstering van meerdradige applicaties wordt bemoeilijkt door de onderlinge interacties tussen de draden via synchronisatie en gedeelde hulpbronnen in de hardware.

In dit werk presenteren we twee technieken voor het bemonsteren van meerdradige applicaties. Tijdsgebaseerde bemonstering lost een aantal problemen op die inherent zijn aan bemonstering van meerdradige applicaties. We tonen aan dat rekening houden met de prestatie per draad alsook de inter-draadinteracties tijdens de bemonstering, de nauwkeurigheid sterk ten goede komt. Bovendien tonen we aan er bij de bemonstering rekening gehouden dient te worden met het tijdsvariërend uitvoeringsgedrag. Tijdsgebaseerde bemonstering laat toe minder dan 10% van de applicatie in detail te simuleren met een gemiddelde afwijking van 3,5% t.o.v. gedetailleerde simulatie.

De tweede bemonsteringstechniek spitst zich toe op applicaties die 'bar-

riers' gebruiken als synchronisatieprimitieve. Deze techniek laat toe representatieve inter-barrierregio's te identificeren met behulp van een microarchitectuuronafhankelijke karakterisatie van de code, gevolgd door clustering. De experimentele resultaten tonen aan de techniek leidt tot een gemiddelde afwijking t.o.v. gedetailleerde simulatie van slechts 0,9% en een gemiddelde simulatieversnelling van 24,7$\times$ (maximaal 866,6$\times$); bovendien daalt het vereiste aantal simulatiemachines met een factor van gemiddeld 78$\times$.

Contents

List of Tables

List of Figures

List of Abbreviations

API	Application Programming Interface
APKI	Accesses per One Thousand Instructions
BBV	Basic Block Vector
CMP	Chip Multi-Processor
CPI	Cycles per Instruction
CPU	Central Processing Unit
DRAM	Dynamic Random Access Memory
FPGA	Field-Programmable Gate Array
GCC	GNU Compiler Collection (formerly the GNU C Compiler)
GHz	Gigahertz (Processor Frequency)
GNU	GNU is not Unix
HPC	High-Performance Computing
ILP	Instruction-Level Parallelism
IOCOOM	In-Order Core, Out-of-Order Memory (Graphite Core Model)
IPC	Instructions per Cycle
ISA	Instruction Set Architecture
kB	Kilobytes (1024 bytes)
KIPS	One Thousand Instructions per Second
LDV	LRU stack Distance Vector
L1	Level 1 (Cache)
L2	Level 2 (Cache)
L3	Level 3 (Cache)
LLC	Last Level Cache
MB	Megabyte (1024 kB)
MHS	Memory Hierarchy State
MIC	Many Integrated Cores
MIPS	Million Instructions per Second
MLP	Memory-Level Parallelism
MPKI	Misses per One Thousand Instructions
MRRL	Memory Reference Reuse Latency
MSI	(Modified, Shared, Invalid) Cache Coherency Protocol
MTR	Memory Timestamp Record
NAS	NASA Advanced Supercomputing

NASA	National Aeronautics and Space Administration
NPB	NAS Parallel Benchmarks
NSL	No-State-Loss
OMP	OpenMP, Open Multi-Processing
OS	Operating System
QPI	Quick-Path Interconnect
ROB	Reorder Buffer
SIMD	Single Instruction, Multiple Data (Instruction Type)
SMARTS	Sampling Microarchitecture Simulation
SMP	Symmetric Multi-Processing
SMT	Symmetric Multi-Threading
SSE	Streaming SIMD Extensions
SSE2	Streaming SIMD Extensions 2
SSE3	Streaming SIMD Extensions 3
SSSE3	Supplemental Streaming SIMD Extensions 3
SV	Signature Vector
TLB	Translation Lookaside Buffer
μPC	Micro-ops per Cycle

Chapter 1

Introduction

1.1 Context

Designs of the first digital electronics components were easy to understand and reason about. These designs could be drawn on paper, and evaluated through manual inspection or by building a system prototype. But, as designs became increasingly complicated, building the next generation of digital computers became much more difficult to do without automated verification. Additionally, as the costs to manufacture these complex machines increased, so did the requirement to produce a machine that performed as expected.

In order to meet these increasing demands, the investigation into simulation of designs before manufacturing took hold. Today, all microprocessors are simulated prior to manufacturing to validate many of their characteristics, such as performance, power and energy consumption, as well as reliability.

The primary concern of microarchitects today is the evaluation of potential performance characteristics of a new microarchitectural feature or enhancement. In this work, we provide a number of solutions for microarchitects who evaluate software on simulated hardware involving modern microarchitecture enhancements. These solutions provide faster, yet still accurate, ways to evaluate representative workloads with reduced system simulation resources.

1.2 Challenges

We observe two major trends in contemporary high-performance processors as a result of the continuous progress in chip technology through Moore's Law. First, processor manufacturers integrate multiple processor

cores on a single chip — multi-core processors. Twelve to sixteen cores per chip are commercially available today (in, for example, Intel's E7-8800 v2 Series, IBM's POWER8 and AMD's Opteron 6300 Series). In addition, specialized processors, such as Intel's Xeon Phi tailored for high-performance computing (HPC) applications, have more than 60 cores, while Tilera's TILE-Gx72 processor, targeted to networking applications, has up to 72. Second, we observe increasingly larger on-chip caches. Multi-megabyte caches are becoming commonplace, exemplified by the 37.5 MB L3 cache in Intel's Xeon E7-8890 v2.

These trends pose significant challenges for the tools in the computer architect's toolbox. Current practice employs detailed cycle-accurate simulation throughout the entire design cycle. While this has been (and still is) a successful approach for designing individual processor cores as well as multi-core processors with a limited number of cores, cycle-accurate simulation is not a scalable approach for simulating large-scale multi-cores with tens or hundreds of cores, for two key reasons. First, current cycle-accurate simulation infrastructures are typically single-threaded. Given that clock frequency and single-core performance are plateauing while the number of cores increases, the (simulation) gap between the performance of the target system being simulated versus simulation speed is rapidly increasing. Second, the increasingly larger caches observed in today's processors imply that increasingly larger units of work need to be simulated in order to stress the target system in a meaningful way. Finally, overall processor complexity has continued to increase as well, with the memory-hierarchy becoming a dominant factor in the simulation time of these large many-core processors.

1.3 Future Simulation Requirements

These observations impose at least three requirements for architectural simulation in the multi-core and many-core era. First, the simulation infrastructure needs to be parallel: the simulator itself needs to be a parallel application so that it can take advantage of the increasing core counts observed in current and future processor chips. A key problem in parallel simulation is to accurately model timing at high speed [56]. Advancing all the simulated cores in lock-step yields high accuracy; however, it also limits simulation speed. Relaxing timing synchronization among the simulated cores improves simulation speed at the cost of introducing modeling inaccuracies. Second, we need to raise the level of abstraction in architectural simulation. Detailed cycle-accurate simulation is too slow for multi-core systems with large core counts and large caches. Moreover, many practical design studies and research questions do not need cycle accuracy be-

cause these studies deal with system-level design issues for which cycle accuracy only gets in the way (i.e., cycle accuracy adds too much detail and is too slow, especially during the early stages of the design cycle). These first two requirements result in a direct simulation speedup allowing for higher turn-around times for early design-space exploration. After this design-space exploration has refined our target direction, a more comprehensive and detailed evaluation of the solution is needed. Therefore, the third, parallel requirement is to reduce parallel multi-threaded workloads into smaller representatives. At the late stage of an investigation, the focus changes from a latency-driven discovery phase into a throughput-limited validation where compute resources face a bottleneck. In order to improve the overall throughput in a typically resource-constrained environment (where the number of experiments to run is larger than the number of resources available), intelligent and accurate application reduction becomes critical to delivering timely results.

1.4 Contributions

In this thesis, we will detail a number of solutions to tackle the requirements needed for next-generation microarchitectural simulation.

1.4.1 Architectural Simulation

The Sniper Multi-Core Simulator brings together a number of technologies to allow for faster simulation. One of these technologies is an improved interval core model which improves performance by 10× on average compared to detailed simulation [31]. In addition to faster core models, we integrate these models into a parallel simulator infrastructure which can lead to improved simulation performance when simulating multi-threaded targets on a multi-threaded host machine. We implemented this for the 64-bit x86 architecture and validated the models against the Intel Nehalem microarchitecture. We demonstrate how simple core models can be misleading for early design space exploration while showing how faster, more accurate simulation infrastructure can lead to predictive results for future design-space explorations.

The Sniper Multi-Core simulator was released publicly in November 2011 and has seen more than 700 downloads by researchers across the globe. Since its release, we have also presented a number of tutorials on the simulator. We presented Sniper tutorials at ISPASS 2012, ISCA 2012, HiPEAC 2013, HPCA 2013 and at IISWC 2013.

This work is published in:

T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12, November 2011

In addition to showing that sufficient detail is necessary for accurate simulation, we developed a new core model that provides a new trade-off point for simulation speed and accuracy. This new core model, the instruction-window centric model, provides additional options for the architect to use when evaluating next-generation microarchitectural options, allowing for modeling of simple in-order, and potentially even SMT cores. Using the insights from interval simulation, these models continue to provide relatively fast simulation speeds while improving accuracy compared to hardware. In addition, we discuss an extension to the interval model, functional-unit contention, or issue contention, that provides additional accuracy with a negligible speed penalty.

This work will appear in:

T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014

1.4.2 Application Sampling

Sampling of multi-threaded applications can speed up simulation of general-purpose multi-threaded applications by reducing the amount of an application that needs to be simulated in detail. We detail a general-purpose solution of multi-threaded application sampling that allows for a reduction in the amount of work necessary to evaluate microarchitectural enhancements.

This work is published in:

T. E. Carlson, W. Heirman, and L. Eeckhout. Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, April 2013

In addition to this work, we have also shown that for an important subset of applications, namely barrier-based multi-threaded applications, it is possible to show an even larger reduction in the amount of resources needed to accurately simulate these applications. Application barriers provide a safe point for parallelization and application comparison, where a unit of work is clearly defined. By comparing microarchitecturally independent parameters of an application, we can evaluate application similarity and select representative regions to take the place of running an entire

application.

This work is published in:

T. E. Carlson, W. Heirman, K. V. Craeynest, and L. Eeckhout. BarrierPoint: Sampled simulation of multi-threaded applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, March 2014

1.5 Structure and Overview

In this thesis, we discuss a number of solutions to improve simulation performance while maintaining high accuracy.

In Chapter 2 we discuss the Sniper Multi-Core Simulator as a faster, but still accurate alternative to cycle-level simulation.

Chapter 3 provides a new direction for processor core simulation, increasing the accuracy of high-level core models themselves with respect to hardware while maintaining high performance.

In Chapter 4 and Chapter 5 we move to an orthogonal topic compared to speeding up the simulator itself. By reducing the amount of work that needs to be done in multi-threaded applications, we can reduce the amount of an application that we need to simulate to validate our new microarchitectural ideas while still maintaining accuracy.

More specifically, Chapter 4 provides details on our sampling methodology for general purpose applications.

In Chapter 5, we discuss additional sampling methodology that provides larger application reductions for a specific class of applications, namely, barrier-based applications.

Finally, in Chapter 6, we summarize this work and describe a number of avenues for future research.

Chapter 2

Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations

This chapter presents the Sniper Multi-Core Simulator. Through the use of high-level core and memory models and parallel simulation, Sniper is able to simulate applications at up to an aggregate of 2.0 MIPS. Sniper was validated against a real hardware platform and it maintains good accuracy across a number of benchmarks. By using Sniper and the interval simulation core models, one can more accurately model hardware platforms than compared to using One-IPC core models, while maintaining good simulation performance.

2.1 Introduction

Because of the trends in computer design, namely larger core counts and cache sizes, simulation runtime has become a primary issue for determining accurate results in a relatively short timeframe. Detailed cycle-accurate simulation is too slow for multi-core systems with large core counts and large caches. Moreover, many practical design studies and research questions do not need cycle accuracy because these studies deal with system-level design issues for which cycle accuracy only gets in the way (i.e., cycle accuracy adds too much detail and is too slow, especially during the early stages of the design cycle).

This chapter deals with exactly this problem. Some of the fundamental questions we want to address are: What is a good level of abstraction for simulating future multi-core systems with large core counts and large

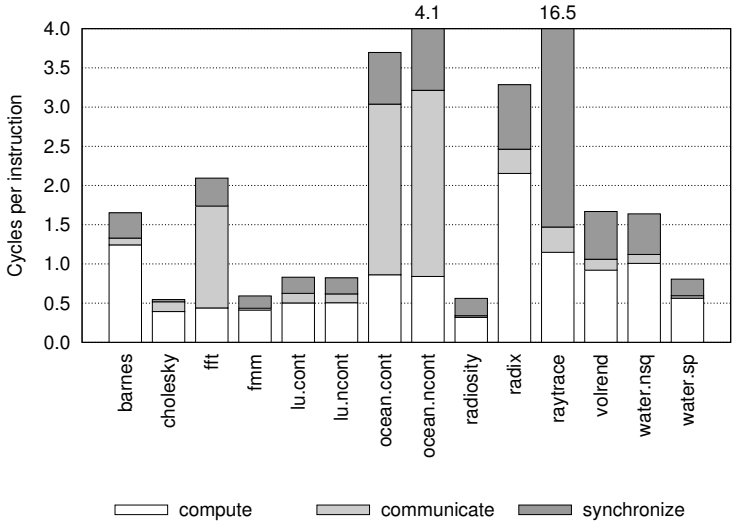


Figure 2.1: Measured per-thread CPI (average clock ticks per instruction) for a range of SPLASH-2 benchmarks, when running on 16 cores. (Given the homogeneity of these workloads, all threads achieve comparable performance.)

caches? Can we determine a level of abstraction that offers both good accuracy and high simulation speed? Clearly, cycle-accurate simulation yields very high accuracy, but unfortunately, it is too slow. At the other end of the spectrum lies the one-IPC model, which assumes that a core's performance equals one Instruction Per Cycle (IPC) apart from memory accesses. While both approaches are popular today, they are inadequate for many research and development projects because they are either too slow or have too little accuracy.

Figure 2.1 clearly illustrates that a one-IPC core model is not accurate enough. This graph shows CPI (Cycles Per Instruction) stacks that illustrate where time is spent for the SPLASH-2 benchmarks. We observe a wide diversity in the performance of these multi-threaded workloads. For example, the compute CPI component of `radix` is above 2 cycles per instruction, while `radiosity` and `cholesky` perform near the 0.5 CPI mark. Not taking these performance differences into account changes the timing behavior of the application and can result in widely varying accuracy. Additionally, as can be seen in Figure 2.2, simulated input sizes need to be large enough to effectively stress the memory hierarchy. Studies performed using short simulation runs (using the *small* input set) will reach different conclusions concerning the scalability of applications, and the effect on scaling of proposed hardware modifications, than studies using the more realistic *large* input sets.

The goal of this chapter is to explore the middle ground between the

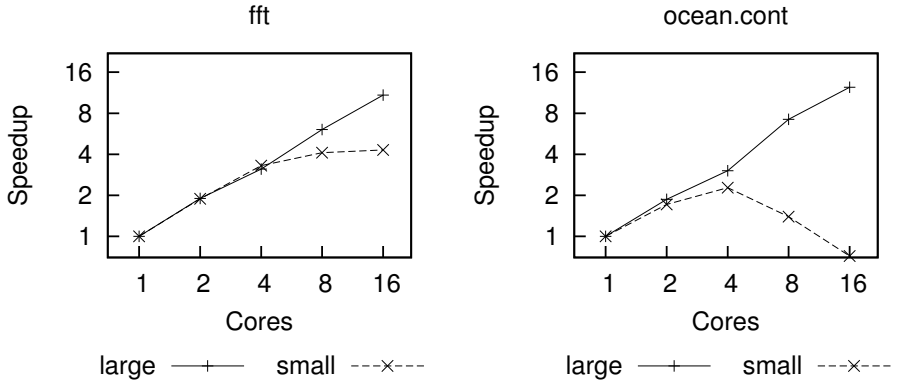


Figure 2.2: Measured performance of SPLASH-2 on the Intel X7460 using large and small input sets.

two extremes of detailed cycle-accurate simulation versus one-IPC simulation, and to determine a good level of abstraction for simulating future multi-core systems. To this end, we consider the Graphite parallel simulation infrastructure [51], and we implement and evaluate various high-abstraction processor performance models, ranging from a variety of one-IPC models to interval simulation [31], which is a recently proposed high-abstraction simulation approach based on mechanistic analytical modeling. In this process, we validate against real hardware using a set of scientific parallel workloads, and have named this fast and accurate simulator Sniper. We conclude that interval simulation is far more accurate than one-IPC simulation when it comes to predicting overall chip performance. For predicting relative performance differences across processor design points, we find that one-IPC simulation may be fairly accurate for specific design studies with specific workloads under specific conditions. In particular, we find that one-IPC simulation may be accurate for understanding scaling behavior for homogeneous multi-cores running homogeneous workloads. The reason is that all the threads execute the same code and make equal progress, hence, one-IPC simulation accurately models the relative progress among the threads, and more accurate performance models may not be needed. However, for some homogeneous workloads, we find that one-IPC simulation is too simplistic and does not yield accurate performance scaling estimates. Further, for simulating heterogeneous multi-core systems and/or heterogeneous workloads, one-IPC simulation falls short because it does not capture relative performance differences among the threads and cores.

More specifically, this chapter makes the following contributions:

1. We evaluate various high-abstraction simulation approaches for

multi-core systems in terms of accuracy and speed. We debunk the prevalent one-IPC core simulation model and we demonstrate that interval simulation is more than twice as accurate as one-IPC modeling, while incurring a limited simulation slowdown. We provide several case studies illustrating the limitations of the one-IPC model.

2. In the process of doing so, we validate this parallel and scalable multi-core simulator, named Sniper, against real hardware. Interval simulation, our most advanced high-abstraction simulation approach, is within 25% accuracy compared to hardware, while running at a simulation speed of 2.0 MIPS when simulating a 16-core system on an 8-core SMP machine.

3. We determine when to use which abstraction model, and we explore their relative speed and accuracy in a number of case studies. We find that the added accuracy of the interval model, more than twice as much, provides a very good trade-off between accuracy and simulation performance. Although we found the one-IPC model to be accurate enough for some performance scalability studies, this is not generally true; hence, caution is needed when using one-IPC modeling as it may lead to misleading or incorrect design decisions.

This chapter is organized as follows. We first review high-abstraction processor core performance models and parallel simulation methodologies, presenting their advantages and limitations. Next, we detail the simulator improvements that were critical to increasing the accuracy of multi-core simulation. Our experimental setup is specified next, followed by a description of the results we were able to obtain, an overview of related work and finally the conclusions.

2.2 Processor Core Modeling

As indicated in the introduction, raising the level of abstraction is crucial for architectural simulation to be scalable enough to be able to model multi-core architectures with a large number of processor cores. The key question that arises though is: What is the right level of abstraction for simulating large multi-core systems? And when are these high-abstraction models appropriate to use?

This section discusses higher abstraction processor core models, namely, the one-IPC model (and a number of variants on the one-IPC model) as well as interval simulation, that are more appropriate for simulating multi-core systems with large core counts.

2.2.1 One-IPC model

A widely used and simple-to-implement level of abstraction is the so-called 'one-IPC' model. Many research studies assume a one-IPC model when studying for example memory hierarchy optimizations, the interconnection network and cache coherency protocols in large-scale multi-processor and multi-core systems [33; 52; 40]. We make the following assumptions and define a one-IPC model, which we believe is the most sensible definition within the confines of its simplicity. Note that due to the limited description of the one-IPC models in the cited research papers, it is not always clear what exact definition was used, and whether it contains the same optimizations we included in our definition.

The one-IPC model, as it is defined in this chapter, assumes in-order single-issue at a rate of one instruction per cycle, hence the name one-IPC or 'one instruction per cycle'. The one-IPC model does not simulate the branch predictor, i.e., branch prediction is assumed to be perfect. However, it simulates the cache hierarchy, including multiple levels of caches. We assume that the processor being modeled can hide L1 data cache hit latencies, i.e., an L1 data cache hit due to a load or a store does not incur any penalty and is modeled to have an execution latency of one cycle. All other cache misses do incur a penalty. In particular, an L1 instruction cache miss incurs a penalty equal to the L2 cache data access latency; an L2 cache miss incurs a penalty equal to the L3 cache data access latency, or main memory access time in the absence of an L3 cache.

2.2.2 One-IPC models in Graphite

Graphite [51], which forms the basis of the simulator used in this work and which we describe in more detail later, offers three CPU performance models that could be classified as one-IPC models. We will evaluate these one-IPC model variants in the evaluation section of this chapter.

The 'magic' model assumes that all instructions take one cycle to execute (i.e., unit cycle execution latency). Further, it is assumed that L1 data cache accesses cannot be hidden by superscalar out-of-order execution, so they incur the L1 data access cost (which is 3 cycles in this study). L1 misses incur a penalty equal to the L2 cache access time, i.e., L2 data cache misses are assumed not to be hidden. This CPU timing model simulates the branch predictor and assumes a fixed 15-cycle penalty on each mispredicted branch.

The 'simple' model is the same as 'magic' except that it assumes a non-unit instruction execution latency, i.e., some instructions such as multiply, divide, and floating-point operations incur a longer (non-unit) execution latency. Similar to 'magic', it assumes all cache access latencies and a fixed

branch misprediction penalty.

Finally, the 'iocoom' model stands for 'in-order core, out-of-order memory', and extends upon the 'simple' model by assuming that the timing model does not stall on loads or stores. More specifically, the timing model does not stall on stores, but it waits for loads to complete (stall-on-use). Additionally, register dependencies are tracked and instruction issue is assumed to take place when all of the instruction's dependencies have been satisfied.

2.2.3 Sniper: Interval simulation

Interval simulation is a recently proposed simulation approach for simulating multi-core and multiprocessor systems at a higher level of abstraction compared to current practice of detailed cycle-accurate simulation [31]. Interval simulation leverages a mechanistic analytical model to abstract core performance by driving the timing simulation of an individual core without the detailed tracking of individual instructions through the core's pipeline stages. The foundation of the model is that miss events (branch mispredictions, cache and TLB misses) divide the smooth streaming of instructions through the pipeline into so called intervals [25]. Branch predictor, memory hierarchy, cache coherence and interconnection network simulators determine the miss events; the analytical model derives the timing for each interval. The cooperation between the mechanistic analytical model and the miss event simulators enables the modeling of the tight performance entanglement between co-executing threads on multi-core processors.

The multi-core interval simulator models the timing for the individual cores. The simulator maintains a 'window' of instructions for each simulated core. This window of instructions corresponds to the reorder buffer of a superscalar out-of-order processor, and is used to determine miss events that are overlapped by long-latency load misses. The functional simulator feeds instructions into this window at the window tail. Core-level progress (i.e., timing simulation) is derived by considering the instruction at the window head. In case of an I-cache miss, the core simulated time is increased by the miss latency. In case of a branch misprediction, the branch resolution time plus the front-end pipeline depth is added to the core simulated time, i.e., this is to model the penalty for executing the chain of dependent instructions leading to the mispredicted branch plus the number of cycles needed to refill the front-end pipeline. In case of a long-latency load (i.e., a last-level cache miss or cache coherence miss), we add the miss latency to the core simulated time, and we scan the window for independent miss events (cache misses and branch mispredictions) that are overlapped by the long-latency load — second-order effects. For a serializing instruction, we

add the window drain time to the simulated core time. If none of the above cases applies, we dispatch instructions at the effective dispatch rate, which takes into account inter-instruction dependencies as well as their execution latencies. We refer to [31] for a more elaborate description of the interval simulation paradigm.

We added interval simulation into Graphite and named our version, with the interval model implementation, Sniper[1], a fast and accurate multicore simulator.

2.2.4 Interval simulation versus one-IPC

There are a number of key differences between interval simulation and one-IPC modeling.

- Interval simulation models superscalar out-of-order execution, whereas one-IPC modeling assumes in-order issue, scalar instruction execution. More specifically, this implies that interval simulation models how non-unit instruction execution latencies due to long-latency instructions such as multiplies, divides and floating-point operations as well as L1 data cache misses, are (partially) hidden by out-of-order execution.

- Interval simulation includes the notion of instruction-level parallelism (ILP) in a program, i.e., it models inter-instruction dependencies and how chains of dependent instructions affect performance. This is reflected in the effective dispatch rate in the absence of miss events, and the branch resolution time, or the number of cycles it takes to execute a chain of dependent instructions leading to the mispredicted branch.

- Interval simulation models overlap effects due to memory accesses, which a one-IPC model does not. In particular, interval simulation models overlapping long-latency load misses, i.e., it models memory-level parallelism (MLP), or independent long-latency load misses going off to memory simultaneously, thereby hiding memory access time.

- Interval simulation also models other second-order effects, or miss events hidden under other miss events. For example, a branch misprediction that is independent of a prior long-latency load miss is completely hidden. A one-IPC model serializes miss events and therefore overestimates their performance impact.

[1] The simulator is named after a type of bird called a snipe. This bird moves quickly and hunts accurately.

Because interval simulation adds a number of complexities compared to one-IPC modeling, it is slightly more complex to implement, hence, development time takes longer. However, we found the added complexity to be limited: the interval model contains only about 1000 lines of code.

2.3 Parallel Simulation

Next to increasing the level of abstraction, another key challenge for architectural simulation in the multi/many-core era is to parallelize the simulation infrastructure in order to take advantage of increasing core counts. One of the key issues in parallel simulation though is the balance of accuracy versus speed. Cycle-by-cycle simulation advances one cycle at a time, and thus the simulator threads simulating the target threads need to synchronize every cycle. Whereas this is a very accurate approach, its performance may be reduced because it requires barrier synchronization between all simulation threads at every simulated cycle. If the number of simulator instructions per simulated cycle is low, parallel cycle-by-cycle simulation is not going to yield substantial simulation speed benefits and scalability will be poor.

There exist a number of approaches to relax the synchronization imposed by cycle-by-cycle simulation [30]. A popular and effective approach is based on barrier synchronization. The entire simulation is divided into quanta, and each quantum comprises multiple simulated cycles. Quanta are separated through barrier synchronization. Simulation threads can advance independently from each other between barriers, and simulated events become visible to all threads at each barrier. The size of a quantum is determined such that it is smaller than the critical latency, or the time it takes to propagate data values between cores. Barrier-based synchronization is a well-researched approach, see for example [56].

More recently, researchers have been trying to relax even further, beyond the critical latency. When taken to the extreme, no synchronization is performed at all, and all simulated cores progress at a rate determined by their relative simulation speed. This will introduce skew, or a cycle count difference between two target cores in the simulation. This in turn can cause causality errors when a core sees the effects of something that — according to its own simulated time — did not yet happen. These causality errors can either be corrected through techniques such as checkpoint/restart, but usually they are just allowed to occur and are accepted as a source of simulator inaccuracy. Chen et al. [16] study both unbounded slack and bounded slack schemes; Miller et al. [51] study similar approaches. Unbounded slack implies that the skew can be as large as the entire simulated execution time. Bounded slack limits the slack to a preset number of cycles,

without incurring barrier synchronization.

In the Graphite simulator, a number of different synchronization strategies are available by default. The 'barrier' method provides the most basic synchronization, requiring cores to synchronize after a specific time interval, as in quantum-based synchronization. The most loose synchronization method in Graphite is not to incur synchronization at all, hence it is called 'none' and corresponds to unbounded slack. The 'random-pairs' synchronization method is somewhat in the middle between these two extremes and randomly picks two simulated target cores that it synchronizes, i.e., if the two target cores are out of sync, the simulator stalls the core that runs ahead waiting for the slowest core to catch up. We evaluate these synchronization schemes in terms of accuracy and simulation speed in the evaluation section of this chapter. Unless noted otherwise, the multithreaded synchronization method used in this chapter is barrier synchronization with a quantum of 100 cycles.

2.4 Simulator Improvements

As mentioned before, Graphite [51] is the simulation infrastructure used for building Sniper. During the course of this work, we extended Sniper substantially over the original Graphite simulator. Not only did we integrate the interval simulation approach, we also made a number of extensions that improved the overall functionality of the simulator, which we describe in the next few sections. But before doing so, we first detail our choice for Graphite.

2.4.1 Simulator choice

There are three main reasons for choosing Graphite as our simulation infrastructure for building Sniper. First, it runs x86 binaries, hence we can run existing workloads without having to deal with porting issues across instruction-set architectures (ISAs). Graphite does so by building upon Pin [48], which is a dynamic binary instrumentation tool. Pin dynamically adds instrumentation code to a running x86 binary to extract instruction pointers, memory addresses, register content, etc. This information is then forwarded to a Pin-tool, Sniper in our case, which estimates timing for the simulated target architecture. Second, a key benefit of Graphite is that it is a parallel simulator by construction. A multi-threaded program running in Graphite leads to a parallel simulator. Graphite thus has the potential to be scalable as more and more cores are being integrated in future multicore processor chips. Third, Graphite is a user-level simulator, and therefore only simulates user-space code. This is appropriate for our purpose

of simulating (primarily) scientific codes which spend most of their time in user-space code; very limited time is spent in system-space code [50].

2.4.2 Timing model improvements

We started with the Graphite simulator as obtained from GitHub.[2] Graphite-Lite, an optimized mode for single-host execution, was back-ported into this version. From this base we added a number of components that improve the accuracy and functionality of the simulator, which eventually led to our current version of the simulator called Sniper.

The interval core model was added to allow for the simulation of the Intel Xeon X7460 processor core; in fact, we validated Sniper against real hardware, as we will explain in the evaluation section. Instruction latencies were determined through experimentation and other sources [27].

In addition to an improved core model, there have also been numerous enhancements made to the uncore components of Graphite. The most important improvement was the addition of a shared multi-level cache hierarchy supporting write-back first-level caches and an MSI snooping cache coherency protocol. In addition to the cache hierarchy improvements, we modeled the branch predictor for the modeled Intel X7460 (Dunnington) as the Pentium-M branch predictor [64]. This model was the most recent branch predictor model publicly available but differs only slightly from the branch predictor in the Dunnington (Penryn) core.

2.4.3 OS modeling

As mentioned before, Graphite only simulates an application's user-space code. In many cases, this is sufficient, and basic system-call latencies can be modeled as simple costs. In some cases, however, the operating system plays a vital role in determining application performance. One example is how the application and kernel together handle pthread locking. In the uncontended case, pthread locking uses futexes, or fast userspace mutexes [29]. The observation here is that for uncontended locks, entering the kernel would be unnecessary as the application can check for lock availability and acquire the lock using atomic instructions. In practice, futexes provide an efficient way to acquire and release relatively uncontended locks in multithreaded code.

Performance problems can arise, unfortunately, when locks are heavily contended. When a lock cannot be acquired immediately, the `pthread_*` synchronization calls invoke the `futex_wait` and `futex_wake` system

[2] Version dated August 11, 2010 with git commit id
`7c43a9f9a9aa9f16347bb1d5350c93d00e0a1fd6`

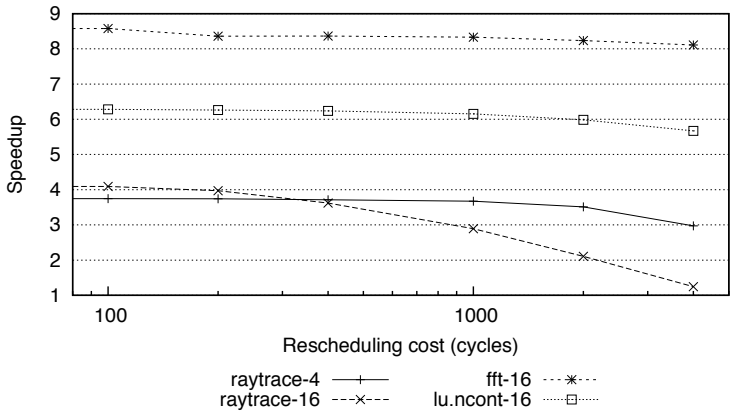


Figure 2.3: Resulting application runtime from an increasing rescheduling cost. For `fft` (very few synchronization calls), `lu.ncont` (moderate synchronization) and `raytrace` (heavy synchronization), with 4 or 16 threads.

calls to put waiting threads to sleep. These system calls again compete for spinlocks inside the kernel. When the pthread synchronization primitives are heavily contended, these kernel spinlocks also become contended which can result in the application spending a significant amount of time inside the kernel. In these (rare) cases, our model, which assumes that kernel calls have a low fixed cost, breaks down.

Modeling kernel spinlock behavior is in itself a research topic of interest [22]. In our current implementation, we employ a fairly simple kernel lock contention model. To this end we introduce the concept of a *rescheduling cost*. This cost advances simulated time each time a thread enters the kernel to go into a wait state, and later needs to be rescheduled once it is woken up. Figure 2.3 explores the resulting execution time when varying this parameter. For applications with little (`fft`), or even a moderate amount of synchronization (`lu.ncont`), increasing the rescheduling cost does not significantly affect the application's runtime. Yet for `raytrace`, which contains a very high amount of synchronization calls, the rescheduling costs quickly compound. This is because, when one thread incurs this rescheduling cost, it is still holding the lock. This delay therefore multiplies as many other threads are also kept waiting.

Figure 2.4 shows the run-times measured on `raytrace`. The *hardware* set shows that on real hardware, `raytrace` suffers from severe contention when running on more than four cores. Our initial simulations (center, *baseline*) however predicted near-perfect scaling. After taking a rescheduling cost into account (right, *reschedule-cost*), the run-times are predicted much more accurately. Note that the rescheduling cost is a very rough approximation, and its value is dependent on the number of simulated cores. This

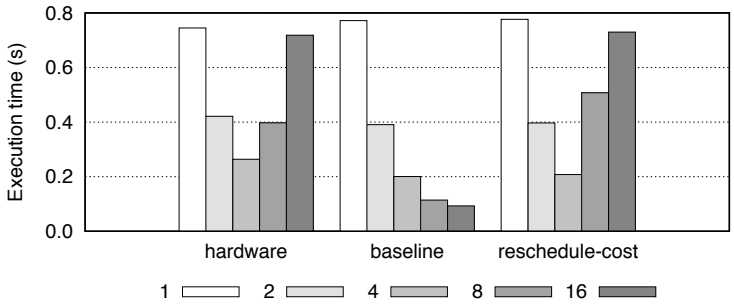


Figure 2.4: Application runtime for `raytrace` on hardware (left), and simulated before (center) and after (right) adding basic kernel spinlock contention modeling.

is because it models operating system penalties for a specific core count. We used a value of 1000 cycles for simulations with up to four cores, 3000 cycles for 8 cores, and 4000 cycles for 16 cores. Only `raytrace` is affected by this; all other application run-times did not change significantly from the baseline runs with a zero-cycle rescheduling cost.

2.5 Experimental Setup

The hardware that we validate against is a 4-socket Intel Xeon X7460 Dunnington shared-memory machine, see Table 2.1 for details. Each X7460 processor chip integrates six cores, hence, we effectively have a 24-core SMP machine to validate against. Each core is a 45 nm Penryn microarchitecture, and has private L1 instruction and data caches. Two cores share the L2 cache, hence, there are three L2 caches per chip. The L3 cache is shared among the six cores on the chip. As Graphite did not contain any models of a cache prefetcher, all runs were done with the hardware prefetchers disabled. Although we recognize that most modern processors contain data prefetchers, we currently do not model their effects in our simulator. Nevertheless, there is no fundamental reason why data prefetching cannot be added to the simulator. Intel Speedstep technology was disabled, and we set each processor to the high-performance mode, running all processors at their full speed of 2.66 GHz. Benchmarks were run on the Linux kernel version 2.6.32. Each thread is pinned to its own core. The simulator was configured to use barrier synchronization with a time quantum of 100ns.

The benchmarks that we use for validation and evaluation are the SPLASH-2 benchmarks [70]. SPLASH-2 is a well-known benchmark suite that represents high-performance, scientific codes. See Table 2.2 for more details on these benchmarks and the inputs that we have used. The benchmarks were compiled in 64-bit mode with –O3 optimization and with the

Parameter	value
Sockets per system	4
Cores per socket	6
Dispatch width	4 micro-operations
Reorder buffer	96 entries
Branch predictor	Pentium M [64]
Cache line size	64 B
L1-I cache size	32 KB
L1-I associativity	8 way set associative
L1-I latency	3 cycle data, 1 cycle tag access
L1-D cache size	32 KB
L1-D associativity	8 way set associative
L1-D latency	3 cycle data, 1 cycle tag access
L2 cache size	3 MB per 2 cores
L2 associativity	12 way set associative
L2 latency	14 cycle data, 3 cycle tag access
L3 cache size	16 MB per 6 cores
L3 associativity	16 way set associative
L3 latency	96 cycle data, 10 cycle tag access
Coherence protocol	MSI
Main memory	200 ns access time
Memory Bandwidth	4 GB/s

Table 2.1: Simulated system characteristics for the Intel Xeon X7460.

SSE, SSE2, SSE3 and SSSE3 instruction set extensions enabled. We measure the length of time that each benchmark took to run its parallel section through the use of the Read Time-Stamp Counter (`rdtsc`) instruction. A total of 30 runs on hardware were completed, and the average was used for comparisons against the simulator. All results with error-bars report the confidence interval using a confidence level of 95% over results from 30 hardware runs and 5 simulated runs.

2.6 Results

We now evaluate interval simulation as well as the one-IPC model in terms of accuracy and speed. We first compare the absolute accuracy of the simulation models and then compare scaling of the benchmarks as predicted by the models and hardware. Additionally, we show a collection of CPI-stacks as provided by interval simulation. Finally, we compare the performance of the two core models with respect to accuracy, and we provide a performance and accuracy trade-off when we assume a number of components provide perfect predictions. Because the interval model is more complex than a one-IPC model, it also runs slower, however, the slowdown is limited as we will detail later in this section.

Benchmark	'small' input size	'large' input size
barnes	16384 particles	32768 particles
cholesky	tk25.O	tk29.O
fmm	16384 particles	32768 particles
fft	256K points	4M points
lu.cont	512×512 matrix	1024×1024 matrix
lu.ncont	512×512 matrix	1024×1024 matrix
ocean.cont	258×258 ocean	1026×1026 ocean
ocean.ncont	258×258 ocean	1026×1026 ocean
radiosity	–room –ae 5000.0 –en 0.050 –bf 0.10	–room
radix	256K integers	1M integers
raytrace	car –m64	car –m64 –a4
volrend	head-scaleddown2	head
water.nsq	512 molecules	2197 molecules
water.sp	512 molecules	2197 molecules

Table 2.2: Benchmarks and input sets.

2.6.1 Core model accuracy

Reliable and accurate microarchitecture comparisons are one of the most important tools in a computer architect's tool-chest. After varying a number of microarchitecture parameters, such as branch predictor configuration or cache size and hierarchy, the architect then needs to accurately evaluate and trade-off performance with other factors, such as energy usage, chip area, cost and design time. Additionally, the architect needs to be able to understand these factors in order to make the best decisions possible with a limited simulation time budget.

Figure 2.5 shows accuracy results of interval simulation compared with the one-IPC model given the same memory hierarchy modeled after the Intel X7460 Dunnington machine. We find that the average absolute error is substantially lower for interval simulation than for the one-IPC model in a significant majority of the cases. The average absolute error for the one-IPC model using the large input size of the SPLASH-2 benchmark suite is 114% and 59.3% for single and 16-threaded workloads, respectively. In contrast, the interval model compared to the X7460 machine has an average absolute error of 19.8% for one core, and 23.8% for 16 cores. Clearly, interval simulation is substantially more accurate for predicting overall chip performance than the one-IPC model; in fact, it is more than twice as accurate.

Figure 2.6 shows a more elaborate evaluation with a variety of one-IPC models for a select number of benchmarks. These graphs show how execution time changes with increasing core counts on real hardware and in simulation. We consider five simulators, the interval simulation approach along with four variants of the one-IPC model. These graphs reinforce our

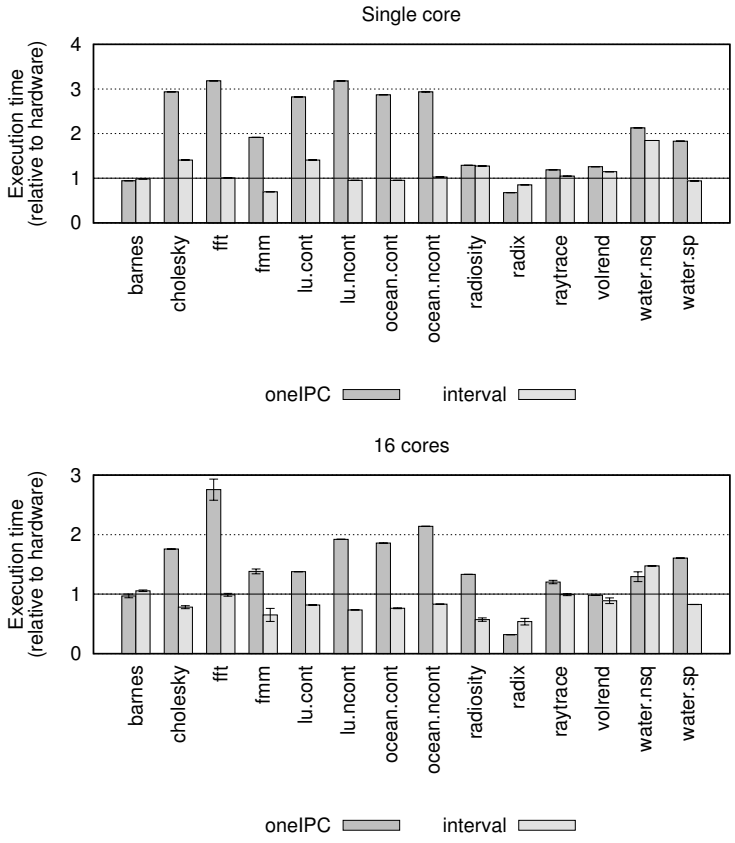


Figure 2.5: Relative accuracy for the one-IPC and interval models for a single core (top graph) and 16 cores (bottom graph).

earlier finding, namely, interval simulation is more accurate than one-IPC modeling, and different variants of the one-IPC model do not significantly improve accuracy. Note that performance improves substantially for `fft` as the number of cores increases, whereas for `raytrace` this is not the case. The reason why `raytrace` does not scale is due to heavy lock contention, as mentioned earlier. Our OS modeling improvements to the Graphite simulator, much like the updated memory hierarchy, benefit both the interval and one-IPC models.

2.6.2 Application scalability

So far, we focused on absolute accuracy, i.e., we evaluated accuracy for predicting chip performance or how long it takes for a given application to execute on the target hardware. However, in many practical research and development studies, a computer architect is more interested in rela-

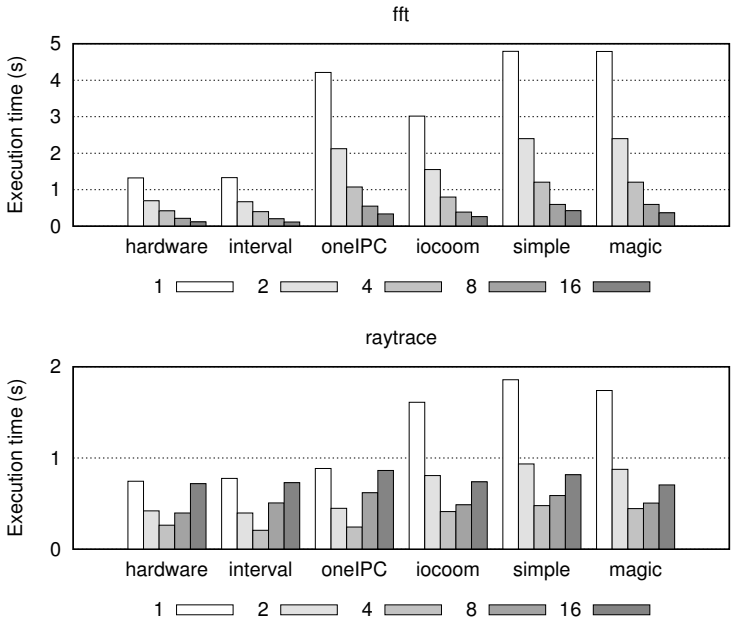


Figure 2.6: Absolute accuracy across all core models for a select number of benchmarks: `fft` (top graph) and `raytrace` (bottom graph).

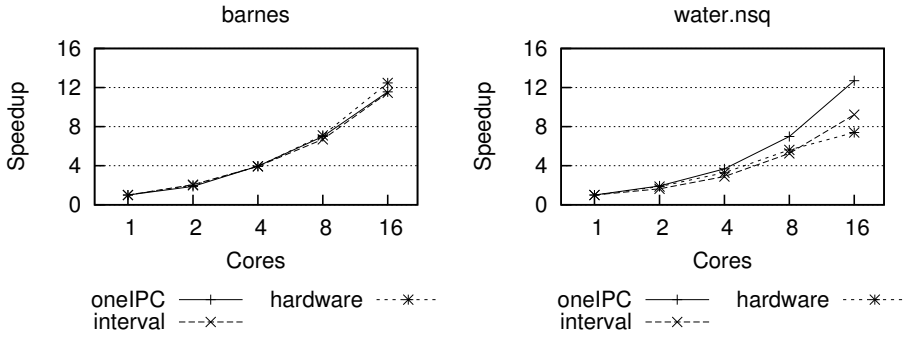


Figure 2.7: Application scalability for the one-IPC and interval models when scaling the number of cores.

tive performance trends in order to make design decisions, i.e., a computer architect is interested in whether and by how much one design point outperforms another design point. Similarly, a software developer may be interested in understanding an application's performance scalability rather than its absolute performance. Figure 2.7 shows such scalability results for a select number of benchmarks. A general observation is that both interval and one-IPC modeling is accurate for most benchmarks, as exemplified by `barnes` (left graph). However, for a number of benchmarks, see

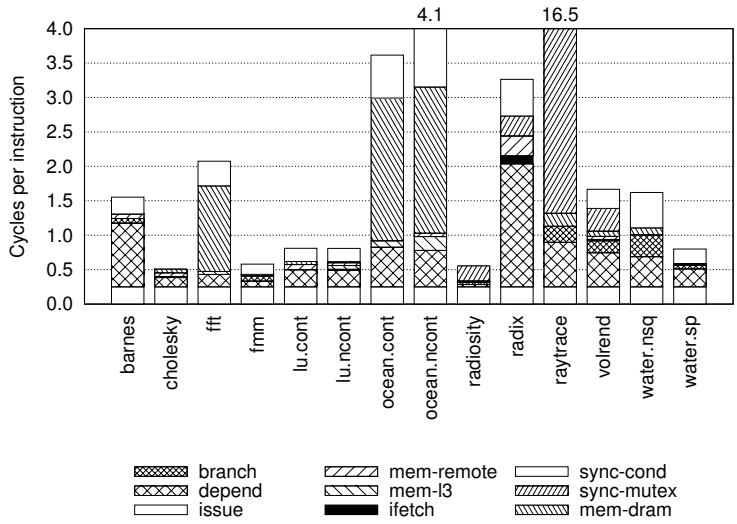


Figure 2.8: Detailed CPI stacks generated through interval simulation.

the right graph for `water.nsq`, the interval model accurately predicts the scalability trend, which the one-IPC model is unable to capture. It is particularly encouraging to note that, in spite of the limited absolute accuracy for `water.nsq`, interval simulation is able to accurately predict performance scalability.

2.6.3 CPI stacks

A unique asset of interval simulation is that it enables building CPI stacks which summarize where time is spent. A CPI stack is a stacked bar showing the different components contributing to overall performance. The base CPI component typically appears at the bottom and represents useful work being done. The other CPI components represent 'lost' cycle opportunities due to instruction interdependencies, and miss events such as branch mispredictions, cache misses, etc., as well as waiting time for contended locks. A CPI stack is particularly useful for gaining insight in application performance. It enables a computer architect and software developer to focus on where to optimize in order to improve overall application performance. Figure 2.8 shows CPI stacks for all of our benchmarks.

As one example of how a CPI stack can be used, we analyzed the one for `raytrace` and noted that this application spends a huge fraction of its time in synchronization. This prompted us to look at this application's source code to try and optimize it. It turned out that a `pthread_mutex` lock was being used to protect a shared integer value (a counter keeping track of global ray identifiers). In a 16-thread run, each thread increments this value

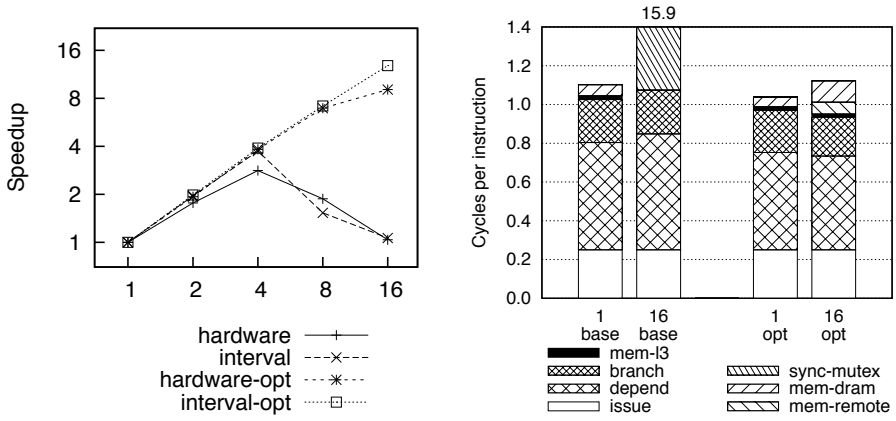


Figure 2.9: Speedup (left) and CPI stacks (right) for `raytrace`, before and after optimizing its locking implementation.

over 20,000 times in under one second of run time. This results in a huge contention of the lock and its associated kernel structures (see also Section 2.4.3). By replacing the heavy-weight pthread locking with an atomic `lock inc` instruction, we were able to avoid this overhead. Figure 2.9 shows the parallel speedup (left) and CPI stacks (right) for `raytrace` before and after applying this optimization.

2.6.4 Heterogeneous workloads

So far, we considered the SPLASH-2 benchmarks, which are all homogeneous, i.e., all threads execute the same code and hence they have roughly the same execution characteristics. This may explain in part why one-IPC modeling is fairly accurate for predicting performance scalability for most of the benchmarks, as discussed in Section 2.6.2. However, heterogeneous workloads in which different threads execute different codes and hence exhibit different execution characteristics, are unlikely to be accurately modeled through one-IPC modeling. Interval simulation on the other hand is likely to be able to more accurately model relative performance differences among threads in heterogeneous workloads.

To illustrate the case for heterogeneous workloads, we consider the `dedup` benchmark from the PARSEC benchmark suite [7]. Figure 2.10 displays the CPI stacks obtained by the interval model for each of the threads in a four-threaded execution. The first thread is a manager thread, and simply waits for the worker threads to complete. The performance of the three worker threads, according to the CPI stack, is delimited by, respectively, the latency of accesses to other cores' caches, main memory, and branch misprediction. A one-IPC model, which has no concept of overlap between

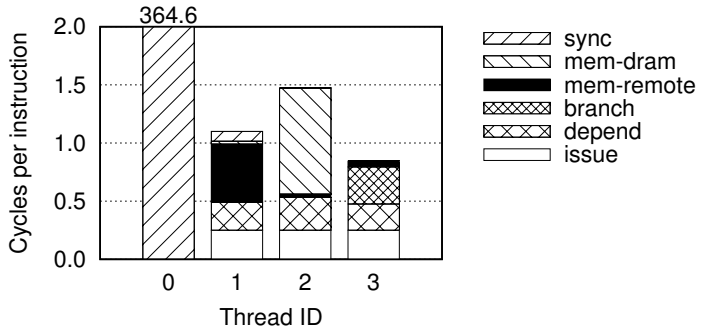


Figure 2.10: CPI stack for each of the four thread types spawned by `dedup`. (Core 0's very high CPI is because it only spawns and then waits for threads.)

these latencies and useful computation, cannot accurately predict the effect on thread performance when changing any of the system characteristics that affect these latencies. In contrast to a homogeneous application, where all threads are mispredicted in the same way, here the different threads will be mispredicted to different extents. The one-IPC model will therefore fail to have an accurate view on the threads' progress rates relative to each other, and of the load imbalance between cores executing the different thread types.

2.6.5 Simulator trade-offs

As mentioned earlier in the chapter, relaxing synchronization improves parallel simulation speed. However, it comes at the cost of accuracy. Figure 2.11 illustrates this trade-off for a 16-core `fft` application. It shows the three synchronization mechanisms in Graphite, barrier, random-pairs and none, and plots simulation time, measured in hours on the vertical axis, versus average absolute error on the horizontal axis. No synchronization yields the highest simulation speed, followed by random-pairs and barrier synchronization. For accuracy, this trend is reversed: barrier synchronization is the most accurate approach, while the relaxed synchronization models can lead to significant errors.

In addition, we explore the effect of various architectural options. Figure 2.11 also shows data points in which the branch predictor or the instruction caches were not modeled. Turning off these components (i.e. assuming perfect branch prediction or a perfect I-cache, respectively) brings the simulation time down significantly, very near to that of the one-IPC model (which includes neither branch prediction or I-cache models).

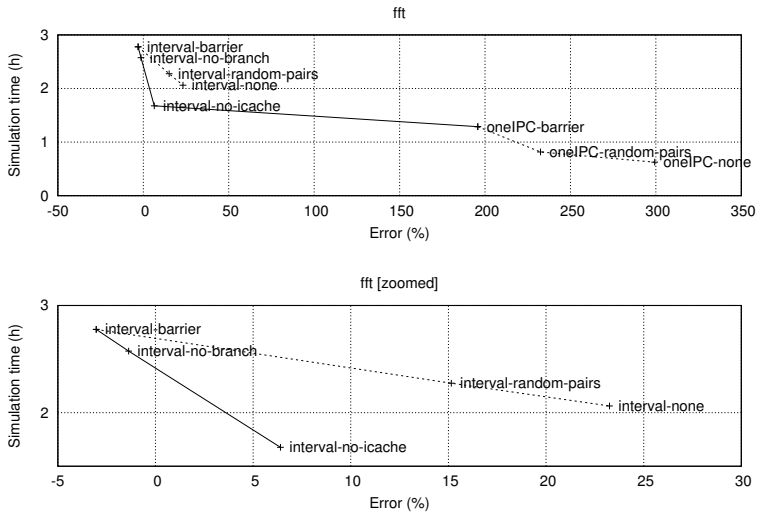


Figure 2.11: Accuracy vs. speed trade-off graphs comparing both synchronization mechanisms for parallel simulation.

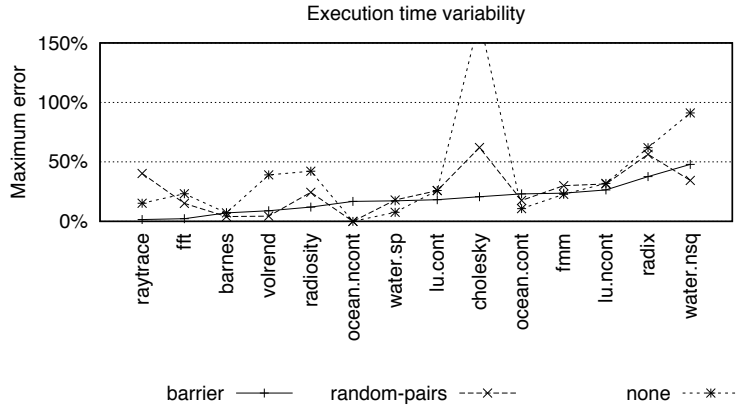


Figure 2.12: Maximum absolute error by synchronization method in parallel simulation for simulating a 16-core system.

2.6.6 Synchronization variability

It is well-known that multi-threaded workloads incur non-determinism and performance variability [1], i.e., small timing variations can cause executions that start from the same initial state to follow different execution paths. Multiple simulations of the same benchmark can therefore yield different performance predictions, as evidenced by the error bars plotted on Figure 2.5. Increasing the number of threads generally increases variability. Different applications are susceptible to this phenomenon to different

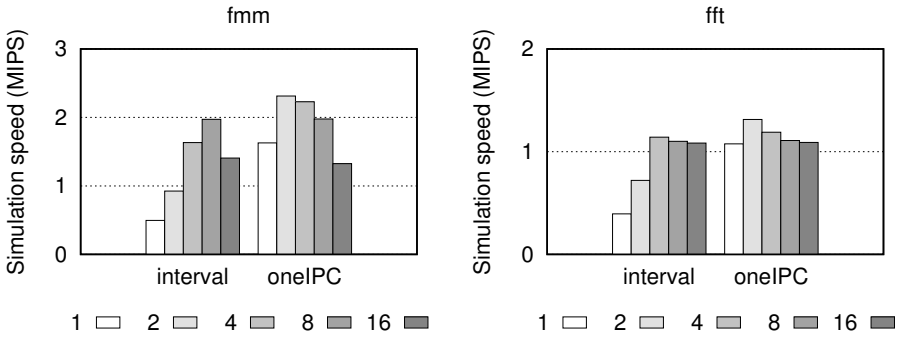


Figure 2.13: Simulation speed of 1–16 simulated cores on an eight-core host machine.

extents, based on the amount of internal synchronization and their programming style — applications employing task queues or work rebalancing often take severely different execution paths in response to only slight variations in thread interleaving. On the other hand, applications that explicitly and frequently synchronize, via pthreads for example, will maintain consistent execution paths.

An interesting observation that we made during our experiments is that performance variability generally is higher for no synchronization and random-pairs synchronization compared to barrier synchronization. This is illustrated in Figure 2.12 which shows the maximum error observed across five simulation runs. The benchmarks are sorted on the horizontal axis by increasing *max* error for barrier synchronization. The observation from this graph is that, although sufficient for some applications, no synchronization can lead to very high errors for others, as evidenced by the `cholesky` and `water.nsq` benchmarks. Whereas prior work by Miller et al. [51] and Chen et al. [16] conclude that relaxed synchronization is accurate for most performance studies, we conclude that caution is required because it may lead to misleading performance results.

2.6.7 Simulation speed and complexity

As mentioned earlier, the interval simulation code base is quite small; consisting of about 1000 lines of code. Compared to a cycle-accurate simulator core, the amount of code necessary to implement this core model is several orders of magnitude less, a significant development savings.

A comparison of the simulation speed between the interval and one-IPC models can be found in Figure 2.13. Here we plot the aggregate simulation speed, for 1–16 core simulations with no synchronization and perfect branch predictors and instruction cache models. As can be seen on

Figure 2.11, adding I-cache modeling or using barrier synchronization increases simulation time by about 50% each, which corresponds to a 33% lower MIPS number. These runs were performed on dual socket Intel Xeon L5520 (Nehalem) machines with a total of eight cores per machine. When simulating a single thread, the interval model is about 2–3× slower than the one-IPC model. But when scaling up the number of simulated cores, parallelism can be exploited and aggregate simulation speed goes up significantly — until we have saturated the eight cores of our test machines. Also, the relative computational cost of the core model quickly decreases, as the memory hierarchy becomes more stressed and requires more simulation time. From eight simulated cores onwards, on an eight-core host machine, the simulation becomes communication-bound and the interval core model does not require any more simulation time than the one-IPC model — while still being about twice as accurate.

Note that the simulation speed and scaling reported for the original version of Graphite [51] are higher than the numbers we show for Sniper in Figure 2.13. The main difference lies in the fact that Graphite, by default, only simulates private cache hierarchies. This greatly reduces the need for communication between simulator threads, which enables good scaling. For this study, however, we model a realistic, modern CMP with large shared caches. This requires additional synchronization, which affects simulation speed.

2.7 Other Related Work

This section discusses other related work not previously covered.

2.7.1 Cycle-level and cycle-accurate simulation

Architects in industry and academia heavily rely on detailed cycle-level simulation. In some cases, especially in industry, architects rely on true cycle-accurate simulation. The key benefit of cycle-accurate simulation obviously is accuracy, however, its slow speed is a significant limitation. Industry simulators typically run at a speed of 1 to 10 kHz. Academic simulators, such as gem5 [8] and PTLSim [72] are not truly cycle-accurate compared to real hardware, and therefore they are typically faster, with simulation speeds in the tens to hundreds of KIPS (kilo simulated instructions per second) range. Cycle-accurate simulators face a number of challenges in the multi-core era. First, these simulators are typically single-threaded, hence, simulation performance does not increase with increasing core counts. Second, given its slow speed, simulating processors with large caches becomes increasingly challenging because the slow simulation

speed does not allow for simulating huge dynamic instruction counts in a reasonable amount of time. Since the original publication of this work, follow-on work has been able to improve simulation speeds using a number of new techniques, such as one that localizes cache behavior to speed simulation of the common case, and then check for potential timing violations and interactions in a follow-on step [58].

2.7.2 Sampled simulation

Increasing simulation speed is not a new research topic. One popular solution is to employ sampling, or simulate only a few simulation points. These simulation points are chosen either randomly [20], periodically [71] or through phase analysis [61]. Ekman and Stenström [24] apply sampling to multi-processor simulation and make the observation that fewer sampling units need to be taken to estimate overall performance for larger multi-processor systems than for smaller multi-processor systems in case one is interested in aggregate performance only. Barr et al. [6] propose the Memory Timestamp Record (MTR) to store microarchitecture state (cache and directory state) at the beginning of a sampling unit as a checkpoint. Sampled simulation typically assumes detailed cycle-accurate simulation of the simulation points, and simulation speed is achieved by limiting the number of instructions that need to be simulated in detail.

While the vast majority of prior work in sampled simulation deals with single-threaded applications running on single-core processors, sampling of multi-threaded applications poses some unique challenges that have not been addressed in prior work. See Chapter 4 and Chapter 5 for two novel methods to apply sampling to multi-threaded applications.

Higher abstraction simulation methods use a different, and orthogonal, method for speeding up simulation: they model the processor at a higher level of abstraction. By doing so, higher abstraction models not only speed up simulation, they also reduce simulator complexity and development time.

2.7.3 FPGA-accelerated simulation

Another approach that has gained interest recently is to accelerate simulation by mapping timing models on FPGAs [18; 67; 54]. The timing models in FPGA-accelerated simulators are typically cycle-accurate, with the speedup coming from the fine-grained parallelism in the FPGA. A key challenge for FPGA-accelerated simulation is to manage simulation development complexity and time because FPGAs require the simulator to be synthesized to hardware. Higher abstraction models on the other hand are easier to develop, and could be used in conjunction with FPGA-accelerated

simulation, i.e., the cycle-accurate timing models could be replaced by analytical timing models. This would not only speed up FPGA-based simulation, it would also shorten FPGA-model development time and in addition it would also enable simulating larger computer systems on a single FPGA.

2.7.4 High-abstraction modeling

Jaleel et al. [39] present the CMP$im simulator for simulating multi-core systems. Like Graphite, CMP$im is built on top of Pin. The initial versions of the simulator assumed a one-IPC model, however, a more recent version, such as the one used in a cache replacement championship[3], models an out-of-order core architecture. It is unclear how detailed the core models are because the simulator internals are not publicly available through source code.

Analytical modeling is one method used to predict workload performance without the need for simulation. Sorin et al. [62] present an analytical model using mean value analysis for shared-memory multi-processor systems. Lee et al. [45] present composable multi-core performance models through regression. Additionally, the interval model [25] is a model used to estimate the performance of a balanced, out-of-order core. While interval modeling successfully models single-core performance, multi-core performance is not modeled by this method.

2.8 Conclusions

Exploration of a variety of system parameters in a short amount of time is critical to determining successful future architecture designs. With the ever growing number of processors per system and cores per socket, there are challenges when trying to simulate these growing system sizes in reasonable amounts of time. Compound the growing number of cores with larger cache sizes, and one can see that longer, accurate simulations are needed to effectively evaluate next generation system designs. But, because of complex core-uncore interactions and multi-core effects due to heterogeneous workloads, realistic models that represent modern processor architectures become even more important. In this work, we present the combination of a highly accurate, yet easy to develop core model, the interval model, with a fast, parallel simulation infrastructure. This combination provides accurate simulation of modern computer systems with high performance, up to 2.0 MIPS.

Even when comparing a one-IPC model that is able to take into account attributes of many superscalar, out-of-order processors, the benefits of the

[3] JWAC-1 cache replacement championship. `http://www.jilp.org/jwac-1`.

interval model provide a key simulation trade-off point for architects. We have shown a 23.8% average absolute error when simulating a 16-core Intel X7460-based system; more than half that of our one-IPC model's 59.3% accuracy. By providing a detailed understanding of both the hardware and software, and allowing for a number of accuracy and simulation performance trade-offs, we conclude that interval simulation and Sniper is a useful complement in the architect's toolbox for simulating high-performance multi-core and many-core systems.

Chapter 3

An Evaluation of High-Level Mechanistic Core Models

In the previous chapter, we provide an introduction to the Sniper Multi-Core Simulator and compare it to the one-IPC core model. We show that the interval simulation models used in Sniper provide higher accuracy while maintaining good performance.

In this chapter, we explore, analyze and compare the accuracy and simulation speed of a new high-abstraction core model. We introduce the instruction-window centric core model, a mechanistic core model that bridges the gap between interval simulation and cycle-accurate simulation by enabling high-speed simulations with higher levels of detail. In addition, we describe a number of enhancements to interval simulation to improve its accuracy while maintaining simulation speed.

3.1 Introduction

In this chapter, we provide an overview of interval simulation, and present improvements that both build on interval simulation as well as extend it in a new direction for higher simulation accuracy. We extend the original interval simulation model to take into account limited execution units, and improve its handling of overlapping memory accesses through a more detailed dependency analysis of memory accesses. These modifications improve accuracy for a range of workloads at a minimal increase in complexity. In addition, we present a new core model that uses the insights from interval modeling, and combines them with the detailed model of the instruction window, or reorder buffer (ROB). We call this methodology instruction-window centric (IW-centric) simulation, where the reorder buffer of an application is at the center of the amount of core-level performance that we can extract. While the original interval simulation method-

ology calculates the ILP of an application analytically, IW-centric simulation models micro-op dependency and issue timing in detail, providing additional accuracy with respect to fine-grained events. The cost of this additional level of detail is a somewhat lower simulation speed. IW-centric simulation therefore represents a different point on the speed versus accuracy trade-off, and can be a good middle ground between interval simulation (as described in Chapter 2) and cycle-accurate modeling.

More specifically, we make the following contributions in this work:

- We present issue contention for interval simulation. Issue contention takes into account core-level instruction execution limitations (e.g., a limited number of functional units) to more accurately predict core performance.

- We present an improvement to the interval model whereby we improve dependency analysis tracking by differentiating between instructions or micro-ops that are dependent on long-latency loads, to those that are not.

- We introduce the instruction-window centric core model, which is a new speed-vs.-accuracy trade-off point between interval simulation and traditional detailed cycle-level core model simulations.

- We present a detailed analysis of the trade-offs between the one-IPC, interval-core and the instruction-window centric simulation models, and provide both single-core and multi-core analysis across a number of simulated hardware configurations to demonstrate the effects that each of the core models have on the scaling and accuracy of the simulations.

- We validate these core models against real hardware and show single-core average errors of 11% for the instruction-window centric model and 24% for interval simulation.

This chapter is organized as follows. We first discuss high-level core models that are used to provide a speed versus accuracy trade-off for microprocessor simulation. Next, we present improvements to interval simulation, and introduce a new core model, instruction-window centric simulation, that improves accuracy with respect to hardware. Finally, we provide an evaluation of these core models for both speed and for accuracy, and discuss how core model resolution affects microarchitecture conclusions.

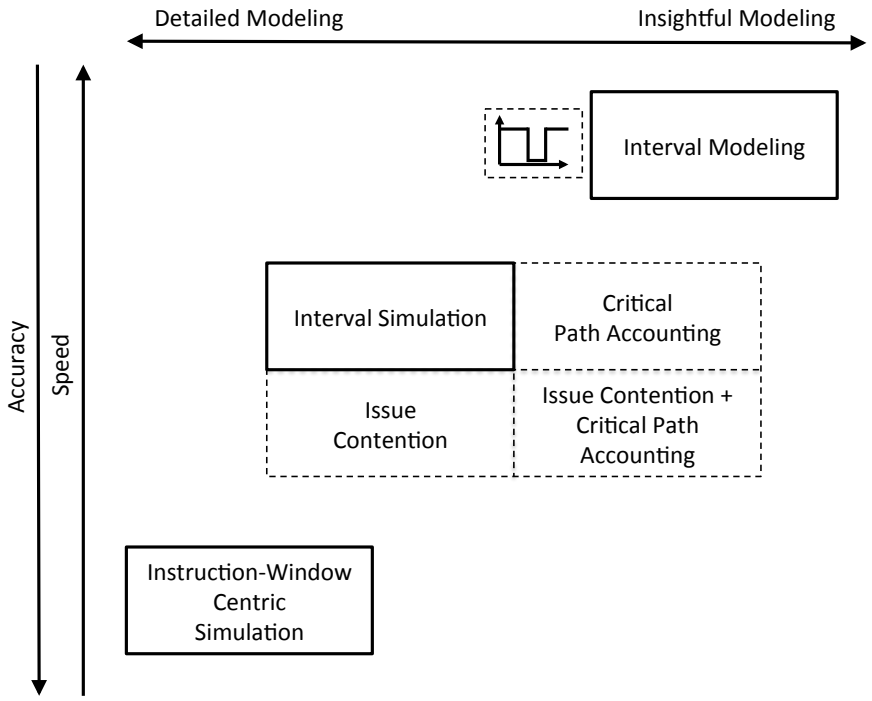


Figure 3.1: Interval modeling and simulation technique taxonomy. Interval modeling uses mechanistic performance modeling techniques to provide application performance estimates. Interval simulation and instruction-window centric simulation build on these insights to provide a more accurate representation of core timing, including timing for multi-core simulation.

3.2 Core-level Abstractions

There are a variety of high-level core abstractions, from a simple one-IPC model to the instruction-window centric core model introduced in this work. See Figure 3.1 for a taxonomy breakdown of the interval model-derived simulation techniques. Below we provide an overview of one-IPC, interval modeling, interval simulation and instruction-window centric simulation. Typically these core models are integrated in a functional-first simulator, where the functional model executes instructions and generates a dynamic instruction stream, potentially broken up into micro-ops. The core model receives this instruction stream and computes the time required to execute these instructions, querying branch prediction and memory hierarchy simulators to discover miss events; see Figure 3.2 for a schematic overview.

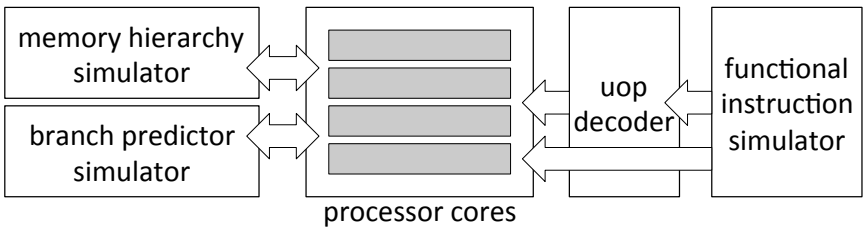


Figure 3.2: A diagram of the main components of a functionally-directed simulator with 4 processor core models.

3.2.1 One-IPC Models

A one-IPC core model is a cycle-by-cycle simulation technique where the core model is set to report a performance of a single instruction per cycle in the absence of long-latency miss events, like long-latency loads or front-end cache misses and branch mispredictions.

A one-IPC model attempts to simulate the performance of a typical out-of-order, multi-issue processor, but at a much higher level of abstraction, and therefore can simulate a core's performance with very little overhead. The absolute performance of the processor (ILP) is not modeled, and the focus is placed on the memory hierarchy of the microprocessor. These models, therefore, provide a faster way to simulate large, multi- and many-core systems that would otherwise not be feasible when using cycle-level core detail.

Unfortunately, there are a number of limitations that occur because of the use of one-IPC core models. one-IPC models do not faithfully model out-of-order memory-level parallelism (MLP) [32; 19] effects where multiple outstanding memory accesses are sent to the memory subsystem. Additionally, these models do not evaluate the amount of exposed ILP during the execution of the application itself. Issuing independent memory accesses in advance is very important to attempt to hide the effects of long-latency memory requests that would normally stall an out-of-order processor's performance. Also, a processor's ILP will dictate the request rate that the memory subsystem sees from the core.

Because of these limitations, there has been an effort to develop both modeling techniques for single-core processors (the interval model) as well as more advanced multi-core simulation techniques that use the insight from interval modeling to provide fast, but accurate simulation.

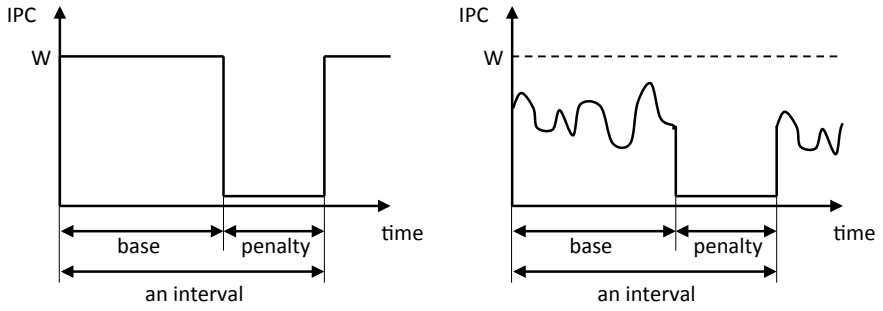


Figure 3.3: A comparison between the estimation of IPC with interval modeling (left) and interval simulation (right). The simulation-based models take a dynamic sequence of micro-ops and adjust the core IPC over time.

3.2.2 Interval Modeling

The interval model [25] is an analytical core modeling technique designed to estimate the performance of applications using the most important factors of a balanced out-of-order core. The interval model exposes the effects of both instruction-level parallelism (ILP) and memory-level parallelism (MLP), and has shown good accuracy as compared to simulated program execution.

The interval model models core performance as a collection of intervals, composed of a period of executing instructions followed by a stalling event that causes the normal flow of execution to halt. In addition, it models the microprocessor as seen from the dispatch stage, providing a model that looks at this point in time, instead of at the issue stage which previous work had focused on [42]. For each miss event, the interval model estimates the resulting penalty to apply to the intervals (See Figure 3.3 for an example). By combining a number of microarchitectural parameters with application characteristics, such as miss rates and the interval lengths between them, and the average window drain time after a branch misprediction, a performance prediction can be made. Using these parameters, the interval model was shown to have low error, around 7% for a 4-issue machine, compared to cycle-level simulation of a single core in SimpleScalar [25].

Interval modeling is a good way to understand application performance of single-core, single-threaded applications on out-of-order machines. Nevertheless, interval modeling does not allow for modeling multi-program and multi-threaded workloads running on multi- and many-core processors.

3.2.3 Interval Simulation

Interval simulation is an extension of the interval model to allow for the simulation of multi-core processors using the insights of the interval model.

The insight gained from interval modeling is that modern processors tend to be balanced with respect to their dispatch and commit widths, their ROB sizes and other supporting structures such as issue buffers and load/store queues. The major way to extract core performance in out-of-order processors is to improve the ability of the core to extract ILP and MLP from the running application. The main limiting factor in extracting ILP and MLP is the ROB size of the processor itself. When engineering an optimized, balanced processor, its components are sized appropriately to allow most of the ILP to be extracted by the ROB. For example, in the Nehalem microarchitecture from Intel, the processor balances out the dispatch rate of 4 micro-ops per cycle with a large number of load and store queue entries, reservation station entries and other supporting structures.

The major advantage of interval simulation over interval modeling is the ability to simulate multi-core platforms. There is a tight entanglement between core performance and the performance of shared resources as the core performance affects the rate at which requests are sent to shared resources which in turn affects core performance. This can be difficult to capture in a stand-alone offline analytical model. Interval simulation bridges this gap. Multi-processor interval simulation is enabled by combining the insights from interval simulation for core performance with a detailed performance model of shared resources. An additional advantage of interval simulation is that the instructions (or micro-ops) are held and processed just one time and in program order. This simulation technique allows for speedups of more than $10\times$ when compared to M5 [31].

New and Old Windows

In Figure 3.4, middle, we show a graphical representation of the core for the interval simulation model. On the right side of the figure, micro-ops enter the new window which holds upcoming micro-ops (the number equal to the size of the ROB) that are to be dispatched. On the left hand side is the old window. The old window holds the micro-ops that have previously been dispatched. The dispatch width of the processor determines the maximum rate at which instructions are moved into the old window from the new window. Using both of these windows, it is possible to both determine the ILP of an application, and extract available MLP to the memory hierarchy. In the absence of miss events, such as long-latency loads, the ILP exposed by the ROB of an out-of-order processor will determine the current performance of the core. In interval simulation, the application ILP is de-

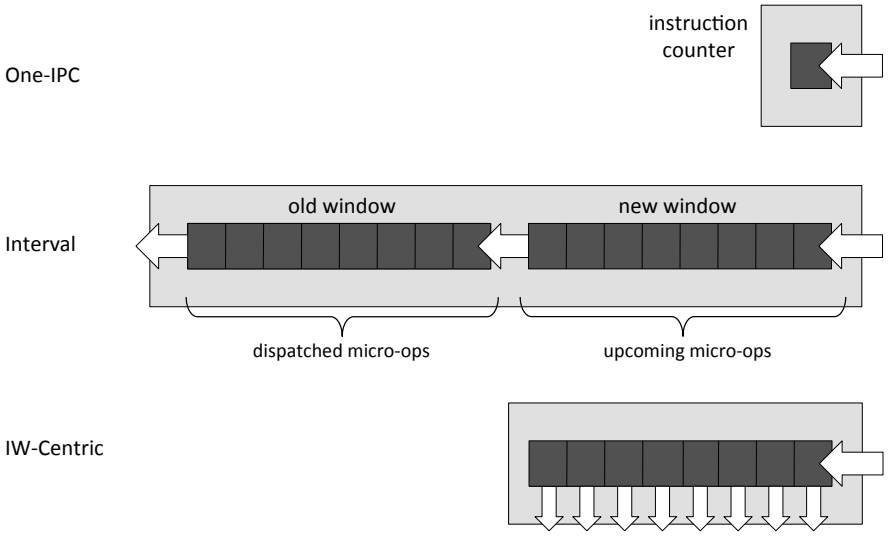


Figure 3.4: A diagram of the different core performance models. The one-IPC model (top) is the simplest model that only keeps track of instruction count. More advanced models, such as interval simulation (middle) and the instruction-window centric model (bottom) provide accuracy at the cost of simulation speed.

termined using the old window, while MLP and other overlap effects (e.g., a branch miss hidden under long-latency load miss) are extracted from the new window. The new window, representing the upcoming micro-ops, remains full at all times to allow for the identification of MLP. Each window contains the number of micro-ops that would exist in an ROB of an out-of-order processor. For the Nehalem microarchitecture, with an ROB size of 128 micro-ops, we would model this as a window with 128 micro-op entries, and an old window also with 128 micro-op entries.

To determine how much progress the ROB has been able to make on the most recent ROB-sized collection of micro-ops, we approximate instantaneous ILP of the upcoming micro-ops in the ROB with Little's Law [47]. The instantaneous IPC is calculated as the number of instructions in the entire old window divided by the latency of the instructions on the critical path [31]. By repeating this process for each micro-op (and accumulating the left-over non-integer amounts of work for future micro-ops) we can accurately estimate the application's ILP during the non-penalty portion of an interval.

In addition to baseline application ILP, the performance impact of miss events needs to be calculated. The interval model distinguishes between front-end and back-end miss events and in the following sections we will discuss how to account for each type of event.

Front-end Stalls

Front-end stalls, according to the original interval model, can be approximated by adding a penalty equal to the time it takes to resolve an I-TLB, or I-cache miss. Therefore, if a level-1 and level-2 miss occurs, resulting in an N cycle L3 hit penalty, we estimate the penalty of the miss on the core as stalling dispatch for N cycles. This resulting delay occurs because the processor is unable to dispatch additional useful micro-ops while waiting for the next instruction stream to be fetched and decoded.

For branch mispredictions, we can estimate the performance penalty as the sum of two components. The first is the latency from dispatch up to the execution of the branch to detect the wrong prediction, and the second component is the time it takes to refill the front-end with instructions from the correct path. In interval simulation, we approximate the latency to determine the correct branch target as the critical path in the old window at the time that we dispatch the branch instruction. The front-end refill time is a constant defined by the microarchitecture itself.

Back-end Stalls

Back-end stalls, such as serialization and long-latency loads (LLC misses and D-TLB misses, etc.) are modeled by interval simulation using the insights from the interval model. A long-latency load instruction is one that causes the processor to stall because the core is not able to commit the load that has not yet received a result from memory. When this occurs, the core experiences a penalty that is equivalent to the load instruction's access penalty [25]. MLP, or memory-level parallelism, is an important effect of out-of-order processors [32; 19]. In order to extract MLP in interval simulation, we scan the new window after dispatching each long-latency load to determine if there are upcoming independent loads that could be issued, representing application MLP. Any dependent load in the window (e.g., in pointer chasing applications) would not be issued until the resulting load has completed. Nevertheless, the out-of-order core can expose additional loads, issuing those loads whose inputs do not depend on the result of the issuing load. By iterating over the new window we can issue independent loads, exposing application MLP. The processor penalty for multiple independent overlapping long-latency loads exposed as MLP is treated as the penalty of a single long-latency load. Serialization instructions incur a penalty equal to the processor window drain time. Other second-order effects, such as overlapping I-cache/I-TLB and independent branch mispredictions with long-latency loads are also taken into account by interval simulation. Additionally, serialization instructions that are encountered while handling MLP halt the detection of additional overlapped

long-latency loads and drains the old window.

3.3 Interval Simulation Improvements

Through the mechanistic modeling of modern out-of-order microprocessors, we are able to both better understand how they operate, and use those assumptions to improve microprocessor simulation. In the next sections we introduce a number of interval simulation enhancements, above what was introduced in Chapter 2, to improve simulation accuracy.

3.3.1 Functional Unit Contention Modeling

Interval simulation [31] assumes that a microprocessor is balanced with respect to its ROB size and dispatch width. What this means is that the front-end, back-end and supporting structures, such as execution units and load and store buffers have been sized to be large enough for a typical application.

Unfortunately, there exist classes of applications that fall outside of this realm. For example, on Intel's Nehalem architecture, there is only a single issue port for 64-bit floating-point multiply instructions. With a micro-benchmark tailored to use only 64-bit multiplications, the maximum number of independent multiplies that can be maintained will be just 1 per cycle, and not the 4 micro-ops that the processor was designed to dispatch. Therefore, when using interval simulation to model the performance of this micro-benchmark, the result will not be what is expected. Interval simulation will report that the performance is 4 times higher (corresponding to the dispatch width of 4 micro-ops per cycle) than the machine can actually perform.

Although this effect applies to floating-point applications, there are other instructions that, when dominating the dynamic application instruction mix, can also affect performance in the same way. Large numbers of branches, or load and store micro-ops also behave similarly because each of these micro-ops can only be issued to a single issue port. Additionally, generic micro-ops can only be issued at a rate of 3 per cycle, and not a maximum of 4, which is the maximum micro-op dispatch rate of the processor.

To be able to account for the discrepancy between dispatch width and issue capabilities, and therefore reduce simulation error, we propose resource contention modeling for interval simulation to improve the resulting accuracy with a negligible reduction in simulation performance.

Issue contention is modeled in the interval simulation model by extending the old window analysis in a new direction. Instead of only using Lit-

tle's law to determine the current IPC of the application, we also take into account the utilization of other resource-sensitive units, such as pipelined and non-pipelined execution units and issue ports. By updating the effective dispatch rate calculation to take into account these additional restrictions, one can determine if we are using more execution units or ports than are available. Although out-of-order processors are able to schedule events whenever there is a time slot available, if one uses too many resources, on average, this can limit the speed of the processor.

The dispatch rate of a processor in the absence of miss events is estimated with interval simulation using Little's law. The dispatch rate of the processor is therefore the number of instructions in the reorder buffer (typically the entire ROB in the absence of miss events) divided by the critical path length.

$$R_{dispatch} = N/L_{criticalpath}$$

We model this in interval simulation as the number of micro-ops in the old window divided by the critical path length. This provides us with our instantaneous micro-ops/cycle with an infinite dispatch width. Taking into account the dispatch width allows us to model a realistic architecture. To maintain higher accuracy, leftover non-integer portions of the dispatch rate are collected and used for dispatch at a later time.

$$R_{dispatch} = min(W_{dispatch}, N/L_{criticalpath})$$

Finally, to take into account resource contention in a processor, we also keep track of the resource utilization of each micro-op. We extend interval simulation by keeping track of the number of resources (issue slots, etc.) that are used in the old window. The number of resources used in the old window puts a limit to the number of micro-ops that can be issued along the critical path. Each S_n is equal to the minimum number of cycles required to simulate the number of instructions for each contention component. For example, if we would like to model an issue port for 64-bit multiplies for the Nehalem microarchitecture, then for an ROB with 32 64-bit multiply operations, the minimum execution time for those operations will be 32 cycles, or one cycle per instruction per issue port. By collecting a number of different microarchitectural restrictions, we can build a minimum execution time that would be necessary to execute the instructions. The effective critical path is then extended by the number of resources required by the processor for that time.

$$L_{criticalpathcontention} = max(L_{criticalpath}, S_1, S_2, \ldots, S_n)$$
$$R_{dispatch} = min(W_{dispatch}, N/L_{criticalpathcontention})$$

In Figure 3.5, we provide a more detailed example of issue port contention in the interval simulation model. In this example, the ROB window

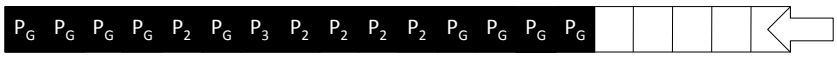


Figure 3.5: An example of port-based issue contention in the updated interval simulation model. The window above represents the interval simulation's old window for a ROB size of 20. The old window contains 15 micro-ops, with the port number that this micro-op can issue from is shown as P_n where n is the port number and G represents ports 0, 1 and 5. Micro-ops are inserted from the right.

size (W) of the system is 20, and the number of micro-ops in the old window (N) is 15. Let's start with a precomputed value for the critical path ($L_{criticalpath}$) of 4. Additionally, we have a number of micro-ops that issue to a number of different ports. Five issue to port 2 (P_2), one issues to port 3 (P_3), and there are nine generic instructions (P_G) that can issue to ports 0, 1 or 5. These three ports can process a large number of generic micro-ops

$$
\begin{aligned}
W &= 20 \\
N &= 15 \\
W_{dispatch} &= 4 \\
L_{criticalpath} &= 4 \\
N_G &= 9 \\
N_2 &= 5 \\
N_3 &= 1
\end{aligned}
$$

First, we calculate the effective dispatch rate (instantaneous micro-ops per cycle, or μPC) using the original method provided by interval simulation. We then redo the calculation taking into account issue contention.

$$
\begin{aligned}
R_{dispatch} &= min(W_{dispatch}, N/L_{criticalpath}) \\
R_{dispatch} &= min(4, 15/4) \\
R_{dispatch} &= min(4, 3.75) \\
R_{dispatch} &= 3.75
\end{aligned}
$$

$$
\begin{aligned}
L_{criticalpathcontention} &= max(L_{criticalpath}, S_1, S_2, S_3, \dots) \\
L_{criticalpathcontention} &= max(4, ceil(N_G/3), N_1, N_2) \\
L_{criticalpathcontention} &= max(4, 3, 5, 1) \\
L_{criticalpathcontention} &= 5 \\
R_{dispatch} &= min(W_{dispatch}, N/L_{criticalpathcontention}) \\
R_{dispatch} &= min(4, 15/5) \\
R_{dispatch} &= 15/5 = 3
\end{aligned}
$$

The resulting improvement in accuracy is apparent. In this small example, the instantaneous performance with issue contention is 3 μPC, while without issue contention we would see a core performance of 3.75 μPC, a 25% faster result. In interval simulation [31], a very low error of 4.6% compared to the M5 simulator was shown compared for a 4-wide Alpha processor. This processor contains a sufficient number of execution units to prevent the issue stage from becoming a bottleneck, and is therefore a balanced microarchitectural design. In recent microarchitectures such as Intel's Nehalem, however, trade-offs are made in the number of execution units which prevents some combinations of micro-ops from being executed in a single clock cycle. When comparing simulation results to real hardware, simulation models must be able to take these microarchitectural imbalances into account if good absolute accuracy is to be expected.

3.3.2 Refilling the window after front-end miss events

In steady state, according to Little's law, the rate at which instructions enter the ROB (dispatch) equals the rate at which they leave the ROB. The interval model computes the latter by analyzing the critical path which determines the rate of execution, and applies Little's law to state that the effective execution rate equals the effective dispatch rate. After a miss event, the old window is flushed to denote the fact that the ROB no longer has independent instructions to operate on. As instructions are dispatched into the ROB, the parallelism exposed by it increases, and the effective execution rate picks up.

For applications where miss events are few, or spaced many instructions apart, this assumption holds and core performance can be estimated with good accuracy. However, in some applications miss events and the associated start-up effects start to dominate. This especially becomes a problem when miss events are spaced so closely that the ROB did not have a chance to be refilled, a situation that caused relatively high errors in the original interval model.

The solution to this problem is to recognize that after front-end miss events, Little's law is not applicable and dispatch can occur at the maximum rate, until the ROB (at this point represented by the contents of the old window) is filled up. We therefore modify the interval simulation model to dispatch at the machine width, rather than the effective dispatch rate computed through analysis of the critical path, as long as the old window is not full. This change proved most helpful for the *fft* benchmark, where it reduces error by 25.3% (to a remaining error of 34.8%, caused mostly by the store buffer filling up which is not taken into account by the interval model).

3.3.3 Modeling of overlapped memory accesses

One drawback of using the interval model is that all memory accesses are sent to the memory hierarchy in program order, which may not represent the actual ordering seen by the memory hierarchy when out-of-order execution re-orders loads and stores with respect to each other. In addition, many higher-abstraction level simulators such as Sniper or M5's atomic memory access mode complete the simulation of each memory access in a single function call, updating all structures (e.g., cache tags) immediately. Both make it impossible for the memory hierarchy to see exact time stamps, which in turn makes it difficult to properly detect overlapping memory accesses in the memory subsystem itself.

We update the interval simulation model to improve modeling of overlapping accesses in two ways. Both changes affect the marking of overlapped accesses and occur when scanning the window of upcoming micro-ops, when the model searches for independent loads that can be hidden under an initial long-latency load.

Pending hits

A first case arises when two independent loads access the same cache line, which was initially not present in the cache. The first access will be a long-latency event as the data has to be fetched from DRAM. The second access, if made in short succession, will be a pending hit; i.e., the cache line in question has already been requested from the next-level cache but the request has not yet completed. An atomic memory hierarchy, however, will return this second access as a hit (simulation of the first access was already completed, leaving the cache line allocated in the L1 cache). A similar problem was addressed by adding support for pending hits in an off-line analytical performance model [17]. To properly delay the second load, when marking micro-ops in the window looking for overlapping accesses, we check for independent loads that access the same cache line and mark the second load

as dependent. This way, keeping the second load's latency as the L1 hit latency, it will complete a few cycles after completion of the first long-latency load.

Dependents of independent long-latency loads

The second case occurs when a chain of dependent loads, that is in itself independent of the initial long-latency load, contains a long-latency load inside of it. Consider the situation where a long-latency load A stalls the processor. The new window contains two additional loads B and C, where C depends on B but both are independent of A. In the original interval simulation model, both B and C would be marked as independent loads and would hence have been allowed to be fully hidden underneath resolution of the original load A.

However, if load B is itself also long-latency, it will not complete until after A completes. Load C, and all instructions depending on it, could therefore also not be assumed to be hidden under the long-latency event caused by A. The required modification made to the interval model here is to, when walking the window looking for overlapping loads, stop marking loads as overlapped once they depend on a newly found long-latency load.

3.4 Instruction-Window Centric Simulation

Interval simulation has been shown to be both a fast and accurate way to simulate the effects of microarchitectural changes on performance [11]. Nevertheless, there are some limitations that make extending the interval simulation model difficult. For example, extending interval simulation to support functional unit contention (Section 3.3.1) required the development of a new core modeling methodology to accurately estimate the timing of the microprocessor. Evaluating a system that consists of a variety of core configurations would become much more difficult if new modeling advances were needed for each core type. Therefore, a new core model that provides both higher fidelity with respect to core and cache hierarchy models would benefit the community if it was able to more accurately model a microarchitecture while still providing performance speedups compared to detailed cycle-level processor models. We therefore introduce instruction-window centric simulation as this middle ground that provides more detail to allow for future core microarchitectural changes without modeling updates, while continuing to provide faster simulation speeds than traditional cycle-level simulators.

3.4.1 Overview

Instruction-window centric simulation builds on many of the insights of interval simulation [31]. While the cache hierarchy and branch predictors continue to be simulated in detail (as is done in interval simulation), many structures such as the fetch and decode logic, additional hardware structures for issuing instructions such as the issue queue, and register renaming, as well as the commit stage are not simulated in detail because of the assumption of a balanced processor microarchitecture. The key change required to go from interval simulation to IW-centic simulation (which enables higher accuracy) is how we model the ability of the hardware to extract application ILP. Instead of approximating the ILP based on Little's Law, where the out-of-order performance is estimated by processing instructions in order, the instruction-window centric model estimates performance by processing micro-ops out of order, in a way similar to how a real processor would issue them. (See Figure 3.4).

The instruction-window centric model replaces the new and old windows of the interval simulation methodology with a new structure that is sized as large as the ROB. Each dispatched micro-op is contained in this structure and is awaiting the results of the operations that they depend on. As the results are completed, additional micro-ops are issued to each functional unit, potentially out-of-order. A major difference between the IW-centric model and the interval model is that the complexity of the issue logic increases as the size of a processor's ROB grows. Additionally, the IW-centric model needs to monitor all input dependencies of each micro-op, with the cost of potentially checking micro-ops multiple times during their lifetime, increasing simulation costs over that of the interval simulation model. Because of this added complexity, the simulation time for IW-centric simulation increases, but at the same time accuracy is also improved.

3.4.2 Implementation Details

Many of the assumptions in the instruction-window centric core model use or extend those that have been established by interval simulation, but allow for a more detailed simulation of application ILP. IW-centric modeling continues to be a dispatch-oriented model, where the dispatch stage of the processor is modeled in detail, and all penalties relate to this stage in the pipeline. Front-end events are handled in a similar way to interval simulation. When there is a branch misprediction or an I-cache or I-TLB miss, the processor waits for the front-end refill to complete before dispatching additional instructions. Interval simulation estimates the branch penalty as the sum of the branch resolution time and the front-end refill penalty. In

instruction-window centric simulation, the branch resolution time is modeled naturally by issuing the branch at the correct time, and only the front-end refill penalty needs to be added. Instruction decode is not modeled directly, but is assumed to be able to keep up with the maximum dispatch rate in the absence of front-end miss events. The maximum number of micro-ops to dispatch per cycle is properly handled and is defined by the core microarchitecture. Loads and stores are executed at issue time, and no special handling needs to take place for long-latency loads. Because the time of the next event can be easily tracked, the core can fast-forward time when there are no events to be processed. All overlapping miss events are handled directly through register and memory dependency analysis. Finally, the commit rate of the processor is fixed to a maximum number of micro-ops per cycle as defined by the microarchitecture.

Issue port contention is modeled directly in the instruction-window centric model. Issue port occupancies are monitored and instruction issue is delayed until a free issue slot is available.

CPI stacks [73] are a first-order method for understanding the causes of performance loss in an out-of-order processor without requiring additional simulation runs. Interval simulation allows for the computation of CPI stacks through the insights of interval modeling. Nevertheless, while instruction-window centric simulation borrows many insights from interval simulation, the method for CPI stack computation needs to be modified because it is no longer possible to attribute the causes of all miss events (especially back-end events) directly. We therefore calculate CPI stacks in a similar way to a model that is suited to performance analysis on real hardware [26]. If the microarchitected dispatch width is limiting the forward progress of the core, we attribute the loss of cycles to this CPI stack component. For front-end miss events, such as a branch misprediction, we also attribute the number of cycles that it takes to restart the core directly to the component that caused the delay. This occurs because the out-of-order processor can no longer make forward progress and the reason for the delay is clear. For back-end miss events, the true cause of the delay is not as straight forward to determine. Therefore, we approximate the cause of the stall to the type of instruction present at the head of the window (serialization, load/store, floating-point, etc.). The rationale for this choice is that the most likely cause of the delay is the instruction at the head of the window.

In addition to timing model details, there are other factors that can contribute to the results of the simulation. The multi-core simulation takes place by first functionally executing the instructions for each core and then by feeding the instructions into each individual core model. This results in a timing model that is slightly behind the natively executed instructions by a maximum of the number of instructions in the ROB of the target microarchitecture. Additionally, because of this direct functional execution,

Component	Configuration
Processor	1 and 2 sockets, 4 cores per socket
Core	2.66 GHz, 4-way issue and commit, 128-entry ROB
Branch predictor	Dothan [64], 8 cycles penalty
L1-I	32 KB, 4 way, 4 cycle access time
L1-D	32 KB, 8 way, 4 cycle access time
L2 cache	256 KB per core, 8 way, 8 cycle
L3 cache	8 MB per 4 cores, 16 way, 30 cycle
Main memory	65 ns access time, 8 GB/s per socket
Inter-processor bus	QPI, 12.8 GB/s per direction

Table 3.1: Micro-architectural configuration

cache and branch predictor pollution that normally occur in out-of-order processors because of wrong-path instruction execution is not present in this model.

3.5 Evaluation and Methodology

3.5.1 Simulation infrastructure

We implemented the three core models described above, one-IPC, interval simulation and the instruction-window centric core model, as well as the enhanced interval model with issue contention, in the Sniper multi-core simulator [11]. Sniper is an execution-driven, user-level simulator that uses functional-first simulation with timing feedback. It implements parallel simulation to improve simulation speed, keeping threads synchronized using a quantum-based barrier synchronization approach which allows for some level of causality violations in exchange for much greater simulation speed (similar to SlackSim [16]).

We then use these core models to simulate the execution of a variety of benchmarks. All core models connect to the same branch predictor and cache models, making a direct comparison between them possible. We also compare simulated results to running the same application on real hardware, which allows us to evaluate the accuracy of each core model in addition to its simulation performance.

We model a dual-socket, quad-core configuration that approximates an Intel Nehalem based server machine. Processor cores are 4-wide and have a 128-entry ROB, and run at 2.66 GHz. Each core has private L1 instruction and data caches in addition to a unified private L2 cache, while all four cores in a package share a L3 cache and DRAM controller. Two quad-core processors are connected using a coherent QPI connection and make up a

Component	Configuration
core models	Instruction-Window Centric, Interval and One-IPC
core issue contention models	enabled, disabled
bus contention model	history-list
DRAM contention model	history-list
synchronization method	time-based barrier
synchronization interval	100 ns
OS emulation	futex replacement
Reschedule cost (cycles)	1k @ 1 core, 10k @ 2 cores 15k @ 4 cores, 25k @ 8 cores

Table 3.2: Simulator configuration options

single shared-memory machine. Other microarchitectural parameters can be found in Table 3.1. When modeling issue contention, we assume the architecture has a given number of issue ports that each accept a subset of all micro-ops. Each issue port can accept a single micro-op for execution per clock cycle. In the Nehalem microarchitecture, there are five issue ports in total. One is dedicated to loads, a second one can be used only by stores. The other three ports are specialized for branches, floating-point additions and floating-point multiplications, respectively; and in addition accept all types of integer instructions. The complete mapping of micro-ops to issue ports is configured according to prior work [28].

The core processor models are configured as either the one-IPC core model, interval simulation core model, or the instruction-window centric core model. Both the interval and instruction-window centric models support optionally enabling functional-unit issue-contention, by enabling the modeling described in Section 3.3.1 for the interval model or by directly accounting for execution unit occupancy on a cycle-by-cycle basis in the instruction-window centric model. By comparing simulations with and without this option we will be able to gauge the increase in accuracy of this additional modeling step. Both the bus and DRAM contention models are configured to use history-list based contention [51]. The simulation models and model-specific parameters are configured as listed in Table 3.2.

3.5.2 Hardware validation

Hardware validation of Sniper with its respective core models was performed on a dual-socket server based on the Intel Xeon X5550 processor. In contrast to Chapter 2, this study uses a more modern processor configuration and therefore the results are not directly comparable. We configure the software and hardware as follows. Benchmark threads are each pinned to

Benchmark	Input set
barnes	32768 particles
cholesky	tk29.O
fmm	32768 particles
fft	4M points
lu.cont	1024×1024 matrix
lu.ncont	1024×1024 matrix
ocean.cont	1026×1026 ocean
ocean.ncont	1026×1026 ocean
radiosity	–room
radix	1M integers
raytrace	car –m64 –a4
volrend	head
water.nsq	2197 molecules
water.sp	2197 molecules

Table 3.3: Benchmarks and input sets

their own core using the *pthread_setaffinity_np()* API. SpeedStep and Turbo Boost are disabled to ensure the processor cores always run at the intended 2.66 GHz frequency. Finally, we disable both Hyperthreading (SMT) and hardware prefetchers as these are also not modeled in Sniper.

3.5.3 Benchmarks

The benchmarks that we use for validation and evaluation are the SPLASH-2 benchmarks [70]. SPLASH-2 is a well-known benchmark suite that represents high-performance, scientific codes. See Table 3.3 for more details on these benchmarks and the inputs that we have used. The benchmarks were compiled with GCC 4.3.2 in 64-bit mode with –O3 optimization and with the SSE, and SSE2 instruction set extensions enabled. On real hardware, we measure the length of time that each benchmark took to run its parallel section (region of interest, ROI) through the use of the Read Time-Stamp Counter (*rdtsc*) instruction. A total of 30 runs on hardware were completed, and the average was used for comparisons against the simulator. In addition, performance counter information was collected using the *perf stat* infrastructure. This allowed us to validate micro-architectural characteristics such as branch misprediction and cache miss rates.

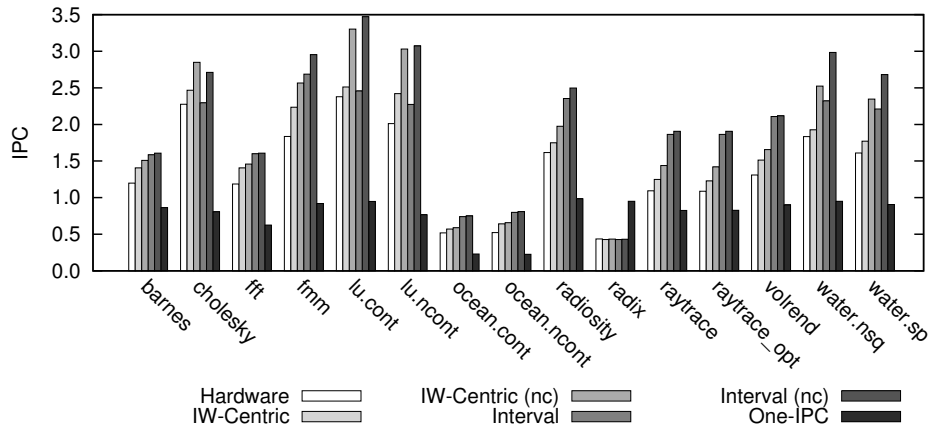


Figure 3.6: Single-core IPC on real hardware and simulated using a variety of core models and benchmarks. Models without issue-contention enabled are labeled (nc) for no issue contention.

	runtime		MPKI avg abs. difference	
core model	avg abs err	max abs err	bp	l3
IW-Centric	11.11	18.76	0.17	0.09
IW-Centric (nc)	21.83	33.62	0.17	0.09
Interval	24.29	41.61	0.17	0.09
Interval (nc)	31.76	42.89	0.17	0.09
One-IPC	92.05	182.04	0.19	0.09

Table 3.4: Single-core average absolute runtime errors and average absolute differences for each simulation model

3.6 Simulation Accuracy Comparison

In this section, we characterize Sniper's simulation accuracy when compared to our real dual-socket Intel Nehalem server. We will vary the core model, comparing one-IPC modeling to interval simulation and the instruction-window centric core model while keeping the memory hierarchy and branch predictor constant. This comparison allows us to keep all of our infrastructure the same to isolate the differences between the core models.

3.6.1 Absolute Accuracy Comparison

Starting with single-core results, Figure 3.6 compares the IPC obtained by real hardware with that predicted by the different core models. The *(nc)* variants of IW-centric and Interval disable modeling of issue contention.

Unsurprisingly, the one-IPC core model incurs large errors (92.0% on average with a maximum of 182.0%, see Table 3.4) and generally underestimates performance on benchmarks where instruction-level parallelism can be exploited to obtain an execution speed of more than one instruction per cycle. On applications with memory-level parallelism such as *ocean*, the one-IPC model does not allow for multiple simultaneous outstanding memory requests and serializes their latency. Alternatively, performance of the *radix* benchmark suffers because of dependency chains through instructions with non-unit latency, here the one-IPC model *overestimates* execution speed. In contrast, the more advanced IW-centric and interval simulation models can provide a much more accurate estimation of execution speed (24.3% for interval simulation and just 11.1% for IW-centric on average, with maximum errors of 41.6% and 18.8%, respectively).

For some benchmarks, including *lu.cont* and *lu.ncont*, pressure on execution units is quite high so taking issue-contention modeling into account significantly improves accuracy for these applications. This is especially important for benchmarks with high IPC which are unrestricted by other hardware components such as memory latency. On average, enabling issue contention modeling improves average accuracy of interval simulation from 31.8% to 24.3%.

In addition to IPC error, Table 3.4 shows a comparison of simulated branch misprediction and cache miss rates as compared to real hardware. These do not depend on the core model used, and are in any case quite low.

3.6.2 Multi-core scaling comparison

Moving on to multi-core results, Figure 3.7 plots the speedup obtained for different core counts; both on real hardware and as predicted using the different core models. Most benchmarks, including for instance the *barnes* and *fmm* applications, exhibit good scaling on real hardware, as an increase in core count leads to an almost linear increase in performance. This behavior is predicted correctly by all core models.

Notable exceptions are the two variants of *ocean*. This application has a large data set and is DRAM bandwidth bound, running this benchmark with more than two threads does not provide an additional benefit in performance. This fact is predicted correctly by the IW-centric and interval simulation core models. In contrast, the one-IPC model does not take memory-level parallelism into account but stalls the core on each DRAM access, underestimating effective DRAM bandwidth pressure and hence overestimating the application's scalability.

Other benchmarks, such as *lu.ncont*, scale well up to four cores, while the eight-core version sees only limited gains. This is because beyond four

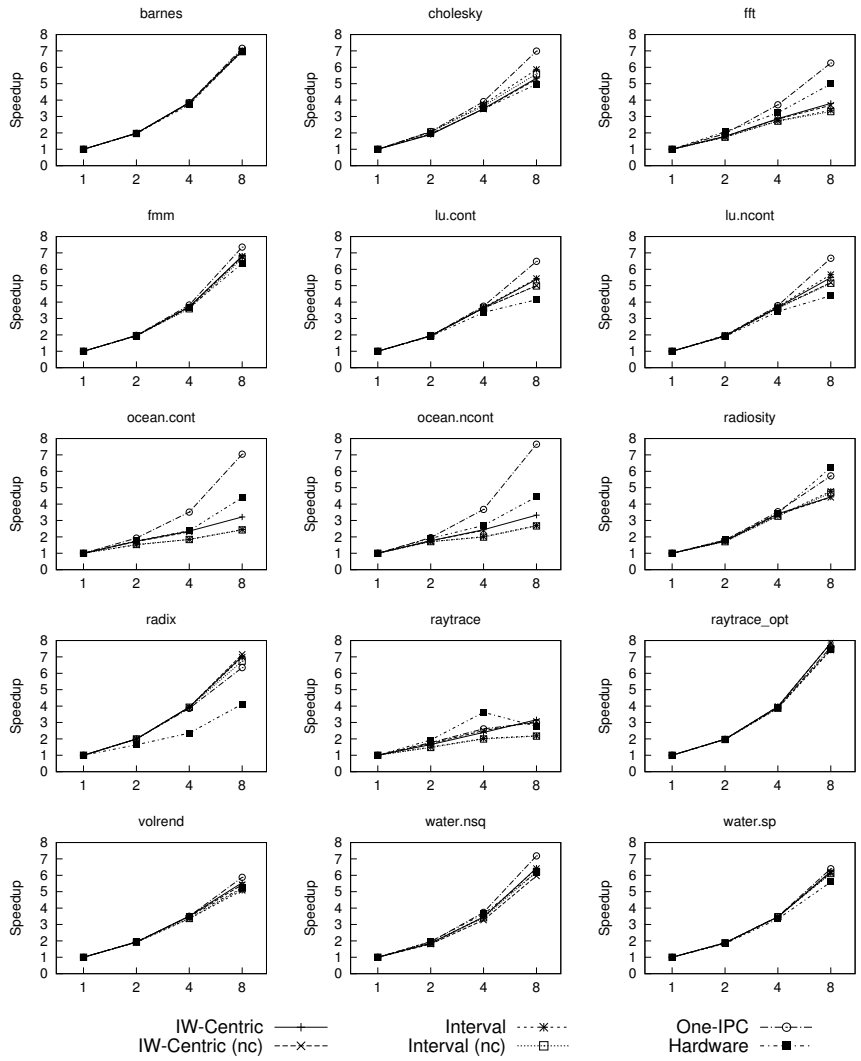


Figure 3.7: Relative performance speedup predictions for the SPLASH-2 applications from 1 to 8 cores.

cores the second processor chip is being used, which incurs inter-socket communication over the QPI links. On real hardware, contention on these links limits performance. The *interval* and *IW-centric* core model predict this correctly. However, the one-IPC core model predicts perfect scaling. This is again because the one-IPC model predicts per-core performance too low (by a factor of 2.5×, see Figure 3.6), which in turn leads to the simulated cores generating a request rate made to the QPI that is too low. The QPI bus is therefore not saturated when being driven by the one-IPC model, which incorrectly leads to the prediction of favorable scaling on the dual-socket run of *lu.ncont*.

	IW-Centric		IW-Centric (nc)		Interval		Interval (nc)		One-IPC	
num cores	avg	max	avg	max	avg	max	avg	max	avg	max
1	11.11	18.76	21.83	33.62	24.29	41.61	31.76	42.89	92.05	182.04
2	9.68	18.79	20.56	34.64	22.15	41.50	29.52	42.57	90.26	162.78
4	14.77	39.19	22.38	39.96	21.76	40.72	28.32	42.37	78.40	150.06
8	20.79	40.32	27.06	43.89	25.91	41.56	31.11	45.08	54.58	100.26

Table 3.5: Average and maximum absolute errors across the simulation models for different core counts (n).

This difference in relative (scaling) accuracy can in fact be correlated with each core model's absolute accuracy. Even though architects usually claim to require only *relative* simulation accuracy, scaling is often largely dependent on contention of shared resources such as DRAM and QPI bandwidth, which in turn requires request rates and thus core performance to have a certain level of *absolute* accuracy. Analyzing the summary of absolute accuracy provided in Table 3.5, we can see that the IW-centric core model has very good accuracy (11.1% on average, with a maximum of 18.8%) for single-core results. Error goes up with increasing core count, mostly because of modeling errors in Sniper's memory hierarchy which dominates performance for four- and eight-core results; leading to a 20.8% average error with a maximum of 40% for the *radix* benchmark, where large numbers of TLB misses and queueing delays caused by the write-back of evicted dirty cache lines push the limits of the level of detail provided in the memory hierarchy. Looking at the other core models, interval simulation starts off at an average 24.3% error for single-core results. With increasing core count the memory subsystem again starts to dominate results, reducing the contribution of the core model somewhat and leading to an eight-core error of 25.9% with a maximum, again for the *radix* benchmark, of 41.6%. Finally, the one-IPC core model starts of with a 92.0% average error (up to 182.0%) for single-core results; while the eight-core error is slightly lower, but due the one-IPC model's inaccurate pressure on the memory subsystem average error is still high at 54.6%.

3.7 Simulation Speed Comparison

In Figure 3.8, we show the average simulation speed (in 1,000 simulated instructions per second of wall-time, or KIPS, aggregated over all simulated cores) across a number of different core models, while changing the size of the simulated system from single-core to dual-socket quad-core (eight cores in total). All simulations were done on an Intel Xeon E5-2650L (Sandy Bridge) based system running at 1.80 GHz. This machine has 16 cores so parallelism in the simulator can optimally be exploited as each simulator

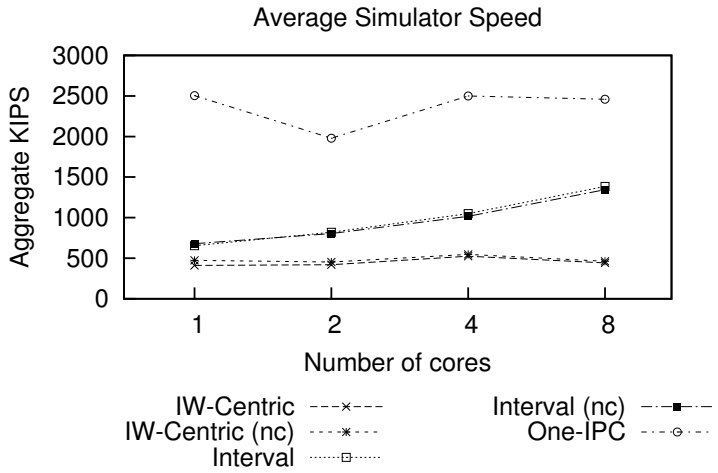


Figure 3.8: Average simulator speed, in KIPS, for a variety of simulation models. Models without issue-contention enabled are labeled (nc) for no issue contention. Core-level issue contention adds very little simulation overhead, yet can greatly improve accuracy.

thread (running the timing model for one simulated core) can run on its own private host core.

Keeping in mind the accuracy of each core model, we can see a clear trade-off of simulation speed versus accuracy. Concentrating on single-core simulations first, the one-IPC core model runs the fastest at over 2.5 MIPS on average, and up to 5.5 MIPS for the *fmm* benchmark. The interval model's much greater accuracy comes at a cost in simulation speed, but still reaches 680 KIPS on average; while the yet more detailed IW-centric core model reaches a single-core simulation speed of around 450 KIPS. An interesting observation is that issue contention, when implemented as described in Section 3.3.1, does not affect simulation speed much, yet can significantly improve accuracy on several benchmarks. This can be clearly seen in Figure 3.9 which plots simulation speed against accuracy (average absolute error to real hardware, for all single-threaded workloads): disabling issue contention modeling (the *(nc)* variants) adds around 10% in additional modeling error, yet does not significantly improve simulation speed. Going from IW-centric over interval to one-IPC modeling does provide the simulator user with a clear choice of core models with different simulation speed over accuracy trade-offs.

Moving to multi-core simulations, the memory hierarchy and synchronization bottlenecks inside the simulator start to become important — both for modeling shared resources such as LLC and DRAM components, and for keeping local clocks of each simulated core synchronized. Using the in-

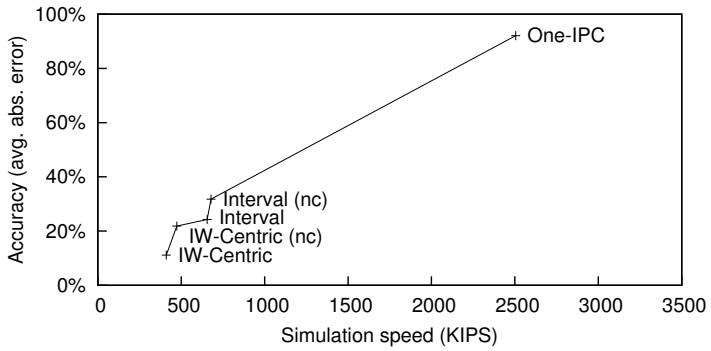


Figure 3.9: Simulation speed versus modeling error of all core models for single-core runs.

terval simulation core model, a speedup of over $2\times$ can be achieved when simulating an eight-core system where the simulation model of each core can run on its own host core. When using the one-IPC model, the core model itself is too simple to have any effect on runtime; here simulation speed is limited by the memory subsystem, which is shared between the simulated cores and therefore can provide only limited parallel speedup. Finally, the IW-centric core model shows limited speedup because of bottlenecks in memory allocation. Whereas the one-IPC and interval simulation core models do not dynamically allocate memory, the IW-centric model — as currently implemented in Sniper — relies heavily on dynamic memory allocation which, in combination with Pin's memory allocator, leads to poor scalability on multiple host cores. Using better allocation techniques such as circular buffers or pool allocation should be able to alleviate this problem.

3.8 Core model resolution affects microarchitecture conclusions

While one-IPC core models are often used to reduce simulation time during micro-architectural evaluations, this increase in performance comes with a trade-off. The one-IPC models do not model a number of key core properties that can be crucial to being able to make accurate design decisions when comparing a number of different microarchitectural design choices. The key differences of a one-IPC model and a modern out-of-order core model are the ability to model both application ILP, and therefore the memory request rate appropriately, as well as the MLP, or the amount of memory parallelism of an application.

To demonstrate some of the limitations of using one-IPC models to eval-

Component	L2 configuration	
	private	*shared*
Size	256 KB per core	1 MB per 4 cores
Associativity	8-way	16-way
Access latency	8 cycles	30 cycles

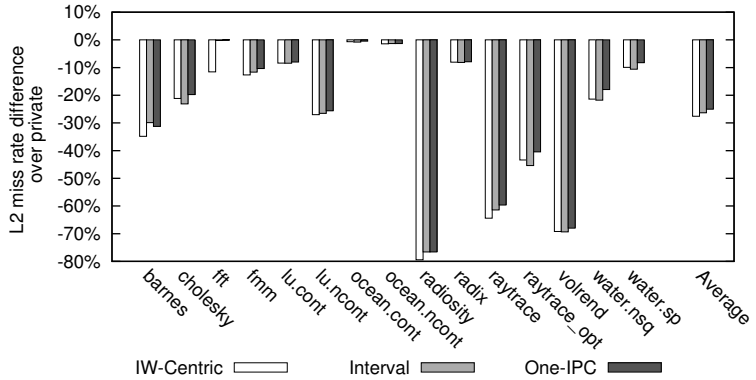
Table 3.6: Micro-architectural configuration for private and shared L2 cache configurations used for one-IPC vs. detailed core model comparisons.

uate processor performance, we show that even when modifying only the caches of a system (a memory hierarchy study), we still need the resolution in ILP and MLP to be able to accurately predict micro-architectural performance trends (relative accuracy).

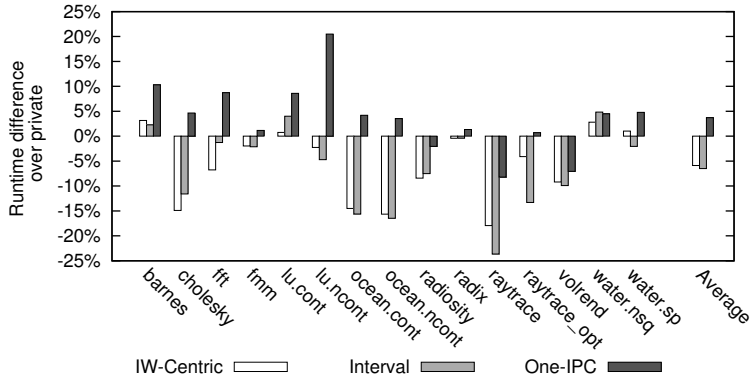
Our experimental configuration is based on the original configuration in Table 3.1, but with slight modifications. For our baseline configuration, we assume four cores, with two levels of cache using a private hierarchy, with the L2 of each core at 256 kB (Table 3.6). We compare the configuration with a shared L2 cache across all 4 cores (Table 3.6). In this configuration, the total cache capacity remains the same, at 256 kB per core, but the latency to access the shared L2 cache goes up to 30 cycles from 8.

Figure 3.10(a) shows the percentage change in miss rates between the private and shared cache systems, where negative (lower) changes show fewer misses in the shared L2 configuration. We see that across all benchmarks, a shared L2 cache configuration reduces the number of L2 misses significantly when compared to the private L2 cache configuration, up to a 75% reduction in some cases. This is caused by the fact that the shared configuration can avoid the data duplication that is present in the private caches, and therefore has a higher effective capacity. More importantly, we see that the cache miss rate changes are stable across all of the core models considered.

When taking application runtime into account, however, the core models no longer agree. Moving from private to shared caches, a reduction in cache miss rate avoids expensive DRAM accesses, but this comes at the cost of a significant increase in latency of (much more common) L2 hits. How this trade-off affects total application runtime, will depend on the relative occurrence of both events, and on how much of the corresponding latency can be overlapped by the core. In Figure 3.10(b), the relative runtime changes are presented between the private L2 and shared L2 cache configurations. A negative value represents a runtime decrease (or improved application performance). According to both the interval and IW-centric core models, most applications prefer the shared L2 cache configuration which reduces their runtime by over 5% on average, with a maximum of around 20%. This indicates that most of the latency increase incurred in the com-



(a) L2 miss rate



(b) Runtime

Figure 3.10: A comparison of L2 miss rates and application runtime of shared versus private caches, as predicted by the one-IPC, interval and instruction-window centric core models.

mon case (i.e., shared L2 hits) is overlapped by the out-of-order core, while the reduction in DRAM misses (which are too long to be fully overlapped with useful work) directly corresponds to improved application performance. However, the one-IPC model predicts a very different scenario, with only a few applications showing slight performance improvements. On average, the results show that applications will run *longer* when using a shared L2 cache, a completely different conclusion. This is because the one-IPC model cannot determine correctly how much of the L2 latency can be overlapped, and overestimates its impact on application runtime.

In other words, the one-IPC core model concludes private caches are best, as the increase in L2 hit latency for shared caches cancels out the reduction in latency caused by avoiding DRAM accesses. In contrast, the more detailed models show that shared caches are the better option since the increased L2 hit latency can be hidden almost completely by out-of-

order execution. This experiment clearly shows that, even when performing experiments that seem to affect the memory subsystem only, certain aspects of the core cannot be ignored but need to be modeled faithfully. As indicated by the results of Figure 3.10, interval simulation poses the right balance for this type of research, as it is able to make the same conclusions yet eschews modeling the more intricate but less important details of the core that are included in the IW-centric core model.

3.9 Conclusion

The micro-architectural trends in modern computer architecture continue to push the limit for simulation technology. With larger caches, larger numbers of cores, and increasingly complex memory hierarchies, the complexity and therefore simulation time required to accurately simulate these architectures has increased. There is a desire for faster simulation throughput, and therefore, faster core models are one way to move closer to these goals.

Nevertheless, the accuracy of the core model is important, even when conducting memory hierarchy studies. Out-of-order cores are able to extract ILP and MLP, which for some applications results in high memory request rates from the individual cores. Accurately simulating an out-of-order core means taking into account each one of its characteristics as seen by the memory hierarchy.

To meet ever-demanding time constraints, there could be a push to move toward faster core models, such as one-IPC models, to determine the best next-generation microarchitectural configurations. Even though cache miss rates tend to be modeled accurately with simple core models, our results show that one-IPC models can be misleading. These simple models do not properly take into account individual core ILP or MLP, which in turn leads to large discrepancies both from an absolute accuracy and a relative accuracy perspective.

Through the use of interval simulation and instruction-window centric core models, one can speed up microarchitectural simulation while maintaining accuracy of the resulting architectural performance evaluation. Both core models provide good absolute accuracy (11.1% for IW-centric and 24.3% for interval simulation) and provide fast simulation speeds (with IW-centric performing just 1.5$\times$ slower than interval simulation), even when combined with parallel simulation techniques.

Chapter 4

Sampled Simulation of Multi-threaded Applications

In the two previous chapters, we provide an overview of two simulator core models (and a parallel simulator infrastructure) that allow for faster simulation of workloads in general. In this chapter, we describe an orthogonal technique to allow for the reduction of the amount of an application that we need to simulate in detail. This technique is called sampling, and it is a well-known workload reduction technique that allows one to speed up architectural simulation while accurately predicting performance.

Previous sampling methods have been shown to accurately predict single-threaded application runtime based on its overall IPC. However, these previous approaches are unsuitable for general multi-threaded applications, for which IPC is not a good proxy for runtime. In this chapter, we propose a time-based sampling approach that takes into account application periodicity and inter-thread synchronization as they play a significant role in determining how best to sample these applications. Through the use of the proposed methodology, we can simulate less than 10% of the total application runtime in detail, significantly speeding up multi-threaded application simulation.

4.1 Introduction

There has been extensive research done with respect to application workload reduction. The reduction of workloads to their relevant components allows for large, realistic input sets to be used both for future architecture exploration as well as application evaluation. Through sampling, architects can perform detailed simulation on a small percentage of the application, covering its most relevant portions [20; 61; 71]. Compared to simulating a complete application with hundreds of billions of instructions, being able

to simulate just a few hundred million instructions in a detailed fashion, while keeping accuracy high, allows architects to explore new architectures with different core and un-core components in much shorter time.

Recent sampling algorithms assume that a number of restrictions are placed on the applications to study. For single-threaded applications, the most important limitation to using sampling to predict application runtime is the assumption that the sampled IPC can act as a proxy for an entire application's runtime [61; 71]. In recent multi-threaded sampling techniques [24; 69], each thread needs to be treated as independent, where they do not explicitly affect the progress of other threads' execution. In other words, per-thread behaviors must be uncorrelated. But for multi-threaded applications that use synchronization primitives, such as locks or barriers, IPC is no longer sufficient and the threads can now directly affect the progress of one another. Threads in these applications can be idle or spinning, unable to make any additional forward progress until one or more other threads reach a synchronization point. During this time, other threads will continue to advance, introducing a gap that represents this thread's idle time. Because of these gaps, it is not possible to solely use IPC and instruction counts to approximate runtime for general multi-threaded applications [2].

In addition to issues involving a large class of multi-threaded applications, program phase behavior is also an important aspect for application sampling. In existing work, the SMARTS methodology [71] is able to accurately predict application IPC by simulating a large number of very small intervals, whereas others [15; 61] use phase behavior to guide sample selection. Without an understanding of an application's phase behavior during sample creation, the result could end up containing periods that alias the original application. Predicting runtime behavior with a sample that aliases the original application has the potential to provide inaccurate results.

To overcome these issues, we propose a multi-threaded application sampling methodology with the following key features: (i) it performs detailed application synchronization during fast-forwarding while keeping track of per-thread performance, and (ii) it uses application phase behavior to select appropriate sampling parameters. We demonstrate that accurately estimating per-thread performance and simulating thread interactions during fast-forwarding is required to maintain high runtime accuracy in tightly synchronized multi-threaded applications. Additionally, through the use of fast pre-simulation application analysis, we take into account application periodicity to allow for accurate multi-threaded application sampling. To the best of our knowledge, this proposal is the first to offer a methodology for performing sampled simulation of multi-threaded applications while maintaining high predicted runtime accuracy.

In this chapter we detail the following contributions:

- We provide a methodology for sampling multi-threaded workloads that provides up to a 5.8× simulation time reduction with an average absolute error of 3.5% while simulating less than 10% of the application in detail.

- We show that both computing per-thread IPC as well as handling inter-thread interactions even during fast-forwarding increases accuracy significantly.

- We demonstrate that application phase behavior needs to be understood and properly taken into account when sampling multi-threaded applications where threads can not be assumed to run independently. To accomplish this, we propose a microarchitecture-independent methodology to determine application phases and how to select the appropriate options for the required speed and accuracy trade-offs.

- We show that large, realistic input sets are needed to make correct design decisions, which in its turn requires accurate sampling for making the simulation of these inputs feasible.

4.2 Fast-Forwarding Parallel Applications

4.2.1 Requirements for Accurate Parallel Fast-Forwarding

Existing techniques for sampled simulation of single-threaded applications, or those treating each thread of a multi-threaded program independently, use purely functional simulation to fast-forward through non-sampled regions [61; 71], or use checkpoints to avoid simulating them at all [65].

These techniques do not directly apply to multi-threaded applications where synchronization or explicit interactions occur. In parallel applications, threads interact through shared memory and synchronization events, influencing the timing of neighboring threads. The use of tracing, or other forms of checkpointing of microarchitectural state, such as used in PinPlay [53] or Flex Points [69], further constrains the absolute ordering of threads in an application. This in turn limits the ability of an architect to view new thread orderings — and their resulting load (im)balance or ability of overlapping communication and computation — that would otherwise occur in an execution on different micro-architectural configurations. Therefore, functional and timing simulation cannot be completely separated during fast-forwarding, but instead, one must take care to preserve the timing of synchronization events. Additionally, the different threads of

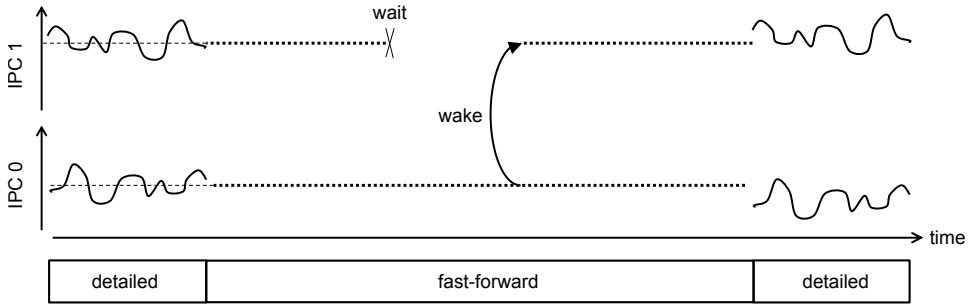


Figure 4.1: Proposed mechanism of fast-forwarding during multi-threaded sampled simulation.

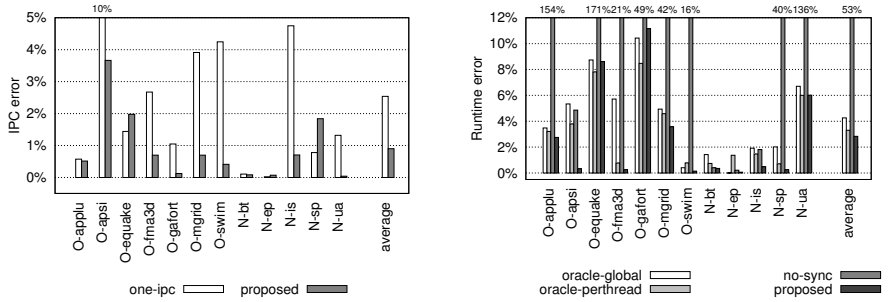
parallel applications can make progress at different speeds — either because they run different code, or they exhibit data-dependent behavior where distinct memory access patterns cause threads to experience different cache miss rates and thus have unequal performance. Considering these effects on a per-thread basis is therefore necessary.

4.2.2 Accurate Multi-Threaded Fast-Forwarding

In our proposed technique, we employ functional simulation of the complete benchmark to capture a sufficient level of accuracy for multi-threaded applications. Sampling is done by periodically simulating detailed performance models during intervals of a predetermined length (the *detailed interval* length D), separated by periods of non-detailed simulation (*fast-forward intervals* of length F). In contrast to single-threaded simulation, we keep track of simulated time, and maintain inter-thread dependencies through shared memory and synchronization events, even while fast-forwarding. We also base sample selection on time, not instruction count, as the latter — due to differences in performance and idle periods among threads — is not comparable across cores.

Figure 4.1 illustrates our fast-forwarding mechanism. Intervals of a fixed length of simulated time are simulated in detail. For each thread, we record the number of instructions executed, and the time this thread did *not* sleep waiting for synchronization events (locks, barriers, etc.) or spin-locks. This allows each thread's execution speed, in instructions per cycle (IPC) to be calculated for the non-idle periods. This non-idle IPC value summarizes the hardware's performance for the section of code executed during the detailed period. Although it is possible to automatically detect and account for spin-locks [46], we chose to use the OpenMP passive wait policy in this work.

While fast-forwarding, the non-idle IPC, along with the current instruc-



(a) Non-idle IPC error (b) Runtime error

Figure 4.2: Accuracy of sampled IPC (left graph) and estimated runtime (right graph) for simulations using different fast-forwarding mechanisms.

tion count is used to keep track of each thread's elapsed time. Most importantly, synchronization events are simulated as normal; i.e., when a thread goes to sleep, functional execution is halted for that thread, and once the thread wakes up it is provided with the current time and functional simulation continues.

In this work, we keep functional cache simulation enabled at all times, and focus on the sampling methodology itself. The use of more efficient warmup techniques, such as Barr et al.'s memory timestamp record [6], would allow for additional speedups and could be a potential direction for future work.

4.2.3 Comparison of Fast-Forwarding Techniques

The key aspects of our proposed fast-forwarding mechanism are to show how one can best preserve inter-thread synchronization and its effect on simulated time, and how accurate knowledge of per-thread IPC variations through time improves accuracy. In this section, we will evaluate the importance of each of these aspects.

Prior work [4; 57] ignored synchronization events during fast-forwarding periods (we call this *no-sync* fast-forwarding). In addition to using synchronization during fast-forwarding, we evaluated a number of alternatives for determining the IPC to use during each fast-forwarding interval. The approaches evaluated either account for time very simply (*one-ipc*), or require up-front knowledge about an application's performance on the architecture before the sampled run commences (*oracle-global* and *oracle-perthread*).

The *one-ipc* mechanism fast-forwards each thread at a fixed IPC of one, so — except when threads are idle — each thread is fast-forwarded by the same number of instructions. The *oracle-global* and *oracle-perthread* fast-

forward mechanisms use IPC information from a fully-detailed simulation, rather than from the previous detailed interval. This allows for a comparison with a theoretical situation where the IPC error caused by sampling is removed, but through-time IPC variations are not taken into account. The *oracle-global* mechanism uses a single fast-forward IPC (the harmonic mean of the IPC for the complete application over all threads), whereas *oracle-perthread* uses the per-thread average. Additionally, both *oracle-global* and *oracle-perthread* use non-idle periods to determine fast-forwarding IPC in the same way that the proposed method does. Finally, the *no-sync* fast-forwarding method does not model the timing of synchronization events during fast-forwarding. Instead, it uses the fast-forward IPC as measured during the preceding detailed interval, similarly to the proposed method, but now the fast-forward IPC lumps together idle and non-idle periods.

In Figure 4.2, we contrast different fast-forwarding mechanisms on a simulated 8-core, shared memory machine. (See Section 4.4 for additional micro-architectural details.) The non-idle IPC, the IPC that occurs when the core is not blocked, waiting for other threads, of the *one-ipc* mechanism and our proposed approach is shown in Figure 4.2(a). Here we see that for most cases, and on average, the proposed fast-forwarding method is more accurate at predicting IPC than the *one-ipc* model. Graph (b) of Figure 4.2, shows how the prediction of total application runtime is affected by different modeling components. Our proposed fast-forwarding technique predicts a simulated runtime with under a 3% average absolute error compared to an average absolute error of 53% for *no-sync*. This shows that taking synchronization into account during fast-forwarding is essential for high accuracy. In addition, the proposed technique's average absolute runtime prediction error is slightly better than both oracle mechanisms, showing that through-time IPC variations are important as well.

4.3 Sample Selection in Parallel Applications

In addition to being able to fast-forward a multi-threaded application while accurately keeping track of the threads' relative progress, there is a critical concern about appropriate sample selection. We make the case that an understanding of application periodicity is crucial for effective sample selection. First, we will show how application periodicity leads to sampling errors, and how this problem applies to our multi-threaded sampling methodology. We then show how one can determine application periodicity in microarchitecture-independent ways, and go on to use this information to build a methodology that constructs reliable sampling parameters based on these application characteristics.

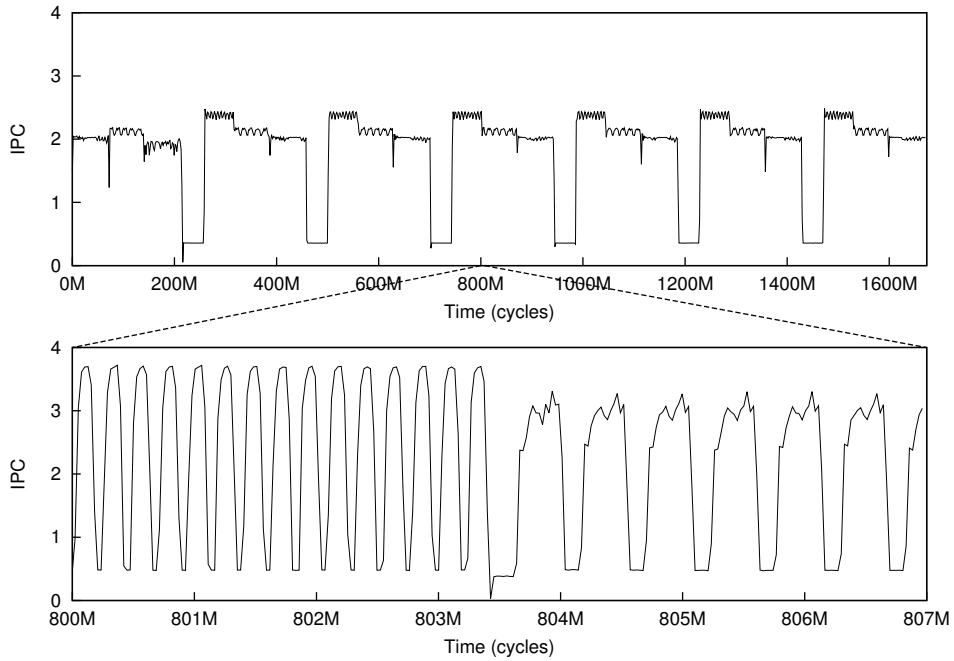


Figure 4.3: IPC trace of `N-ft` (thread 0 out of 8, class A input set): full run (top) and zoomed in (bottom). Several periodicities are visible.

4.3.1 The Effect of Periodicity on Sampling

Many applications exhibit inherent periodicity or phase behavior [55; 61]. Figure 4.3 plots the IPC variation through time of one of the eight threads for the class A input set of the `N-ft` benchmark from the NAS Parallel Benchmarks (NPB) suite. At the macro scale, seven iterations can be seen of a single main period, which has a length of approximately 220M clock cycles. In Figure 4.3 (bottom), the periodicities can be observed at a different scale, with two phases, each with their own iteration length (at 230k and 550k cycles).

Previous work has shown that these inherent application periods can be used to guide sampling. For instance, Casas et al. [15] sample hardware performance counters for exactly an integer number of periods. In this situation, it is feasible to measure the length of one period exactly, since the application runs on actual hardware at native speed. In simulation, however, the application's performance on a given architecture, and hence its periodicity, is a priori unknown.

Casas et al. also note that if the sampling period does not exactly match the size of the periodicity of an application (or an integer multiple), aliasing can occur which can significantly increase the error and variability. In our framework, the IPC sample from the collection of detailed intervals

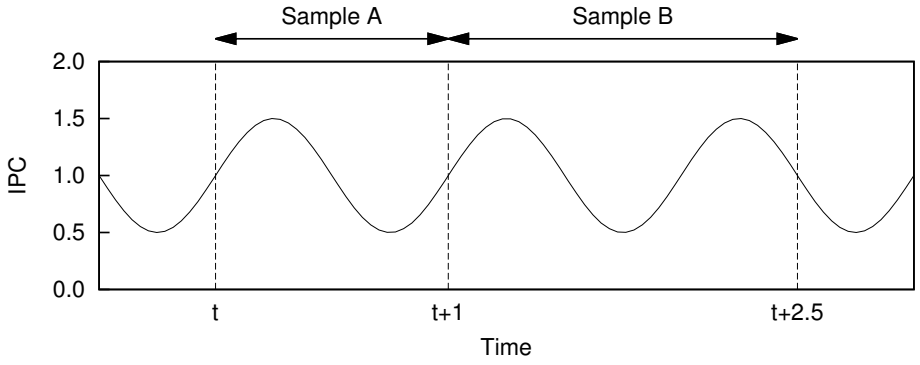


Figure 4.4: Sampling with intervals of exactly one period yields a correct IPC average; when application period and detailed length do not match, sampling errors occur.

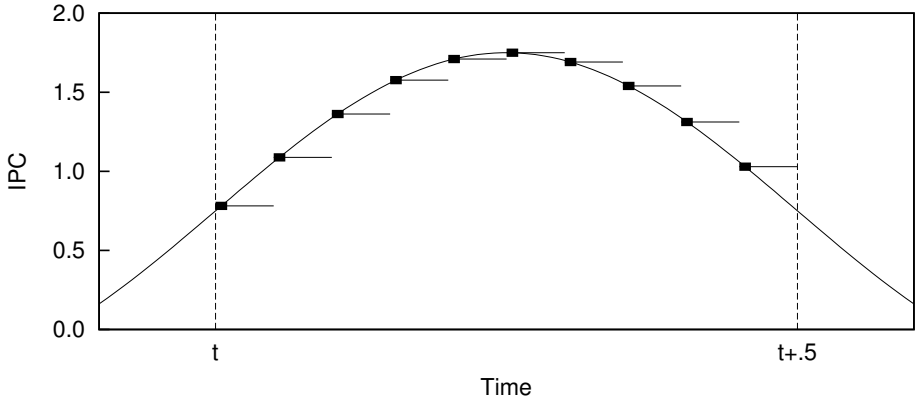


Figure 4.5: When sampling inside of an application's period, a sufficient number of intervals need to be collected to ensure that fast-forwarding IPC accurately tracks actual IPC.

needs to have a high degree of accuracy, as the detailed application performance is used to determine the progress each thread makes during fast-forwarding. Consider for instance the example IPC trace of Figure 4.4. Interval A contains exactly one period; its IPC is therefore equal to the global average. However, interval B, which has a length close to but not exactly equal to the periodicity, has a measured IPC that can be incorrect.

A related problem occurs when sampling intervals are taken *inside* one (much larger) application period, see Figure 4.5. Here, solid squares represent detailed regions, their average IPC is projected forward during fast-forward phases (horizontal lines). Taking a small number of intervals within each iteration can yield inaccurate results, as the instantaneous IPC

will change too much in-between intervals. We therefore want to maximize the number of intervals taken inside one period, so the shape of its IPC curve can be accurately described.

In other sampling methodologies, this type of aliasing is not an issue since the IPC of many small intervals inside a sample is averaged, which because of the central limit theorem yields an accurate estimate of the IPC of the whole application. In our run-time prediction methodology, however, we rely on the detailed regions to be an accurate representation of that region and extrapolate it during fast-forwarding, in order to compute the total program run-time. In contrast to the IPC of single-threaded applications, where positive and negative errors can cancel out during averaging, run-time of parallel applications with inter-thread synchronization behaves differently: positive errors (overestimation of run-time) are often propagated when other threads wait on the slow thread; whereas negative errors (underestimation) are masked for non-critical threads. Therefore, for parallel applications with a substantial amount of synchronization, total run-time is not simply the mean or sum of run-times of the detailed periods, and the central limit theorem does not directly apply to it.

To avoid these aliasing problems, we determine the sampling parameters on a per-application basis, given the application's periodicities. By taking application periodicity into account, one can avoid introducing sampling errors that are caused by the aliasing of the application's periodicities with the detailed sampling period. The next step in our proposed methodology determines the periodicities which allows one to generate the necessary sampling parameters to avoid aliasing.

4.3.2 Determining Application Periodicity

While there are multiple methods to determine application periodicity, we chose to look at those that are micro-architecture independent to allow for up-front calculation of application periodicities regardless of the simulated architecture or the simulation infrastructure used. We use signal analysis techniques in a similar fashion compared to prior work [23; 38] to allow us to capture micro-architecture independent application characteristics.

Our primary method to determine application periodicity is through the collection of basic-block vectors (BBVs) as outlined in [61]. We then perform a windowed auto-correlation on these BBVs. We have created a parallel Pin [48] tool that both generates BBVs and performs the auto-correlation step with minimal application slowdown (around $10\times$ vs. native execution).

An auto-correlation $A(d)$ of the time series of BBVs $B(t)$ is the comparison of a vector of BBVs (a call history) with a version of itself that is at a

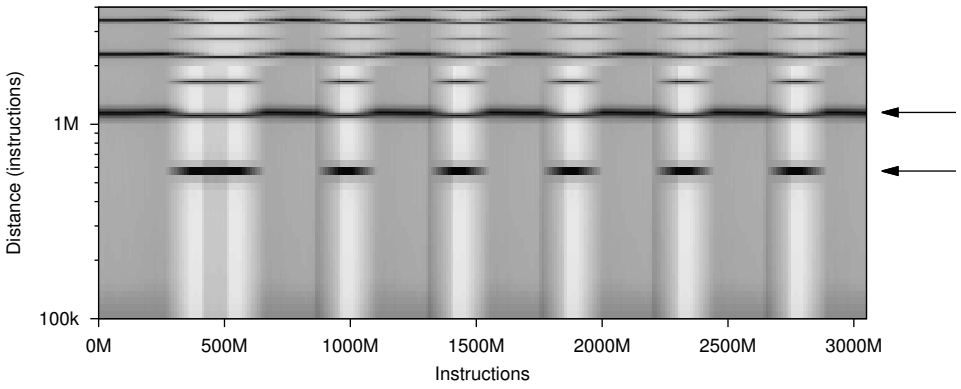


Figure 4.6: BBV autocorrelation for `N-ft` (thread 0 out of 8, class A input set). Strong correlations (dark bands), pointing to periodic behavior, are visible at 550k and 1.14M instructions and their harmonics.

given offset in time d:

$$A(d) = \sum_t \|B(t) - B(t+d)\|$$

By comparing the vector with itself, the sum $A(0)$ is zero which denotes a perfect match. As the offset d between the two vectors is increased, one is able to measure the similarity of the BBVs seen with those at a later point in time. By detecting the points of highest correlation between the two shifted vectors, we can obtain the periodicities of the application. Typically, an auto-correlation is done with the entire vector against itself; by using a windowed auto-correlation in which the summation runs over a localized window around t, rather than over the length of application, changes in periodicity throughout the application's run can be made visible.

An example of the output can be seen in Figure 4.6. Application runtime, measured in instructions, is on the horizontal axis; the vertical axis represents the offset d. Light colors denote a low similarity between the BBVs at a given point in the program with those a distance d away, whereas dark colors denote strong correlation — which implies similar execution behavior [44]. In this case, we can see that the `N-ft` benchmark, running the class A input set with 8 threads, has one periodicity at 550k instructions that occurs for a part of the application runtime, and another which occurs at 1.14M instructions and exists during the entire application execution. These periodicities correspond directly to the 230k and 550k cycle periodicities, respectively, in Figure 4.3.

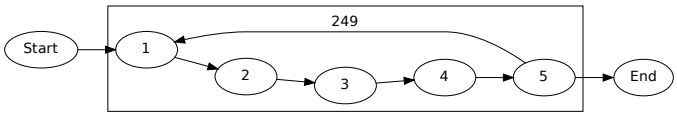
4.3.3 Detecting Large Application Variability over Long Periods

The proposed sampling methodology for multi-threaded workloads makes the assumption that we can detect, and therefore avoid aliasing of the periodicities in an application. Some applications, however, have irregular behavior which can be difficult to determine using the BBV autocorrelation technique alone. This behavior affects the quality of the detailed sample, potentially causing aliasing which can lead to large errors. We therefore augment the BBV-based analysis with a second technique which allows detection of irregular behavior early; this analysis will indicate which applications are not amenable to be sampled reliably.

This technique works by detecting loops in the application's call graph, and comparing the instruction counts of each iteration. For example, by using the OpenMP runtime library, we can monitor the high-level application periodicity. The OpenMP functions are usually called as the result of `#pragma omp` directives in the source code and are therefore closely related to the high-level structure of the application. Using different sets of marker functions, this technique can be applied to most parallel programming models.

The call structure of each thread is derived using a separate Pin tool, which at near-native execution time records a call graph of the application limited to the set of marker functions. This call graph is annotated with both a per-thread and per-loop instruction count. We then use Tarjan's algorithm [63] to determine nested loops inside of the graph. This makes the high-level structure of the application apparent; comparing instruction counts for different iterations provides insight into the application's level of irregularity. For regular applications, the instruction counts thus obtained can be used to verify or augment those obtained through BBV-based analysis.

From this analysis, irregular behavior that can potentially alias the detailed sampling of applications can be detected early. In the case of `N-lu`, the instruction count variability as observed in Figure 4.7 are such (>10%) that we can a priori say that sampled simulation will not provide accurate runtime predictions; we have experimentally verified that errors for this application are indeed as high as 20 to 30%, while the maximum error for applications that show regular behavior is around 8%, with an average absolute error just below 3%. The same holds for `O-ammp` which shows high variability both in the BBV and OMP analysis methods. We therefore leave generalizing the proposed sampling method to irregular applications for future work.



Node	Avg	Δ/μ
1	37.14 M	12.0%
2	38.97 M	16.1%
3	1.96 M	36.6%
4	17.45 M	<1%
5	9.83 M	<1%

Figure 4.7: Loop structure (top) with application call points as edges for the `N-lu` application (thread 3 of 8, class A input set). Node instruction counts (below) with relative spread Δ/μ, defined as $\frac{i_{max}-i_{min}}{i_{count}}$.

4.3.4 Deriving Optimal Sampling Parameters

As noted in Section 4.3.1, using a detailed interval length that is close to an application's periodicity can lead to large sampling errors. For short periods, we will therefore want to sample using a detailed interval length D that is significantly larger than the periodicity P. Similarly, for long application periods we want to take a sufficient number of intervals to accurately describe the IPC changes during this period. Note that in this case, the period at which intervals are taken is of size $D+F$, so the number of intervals taken within a period P equals $P/(D+F)$.

Figure 4.8 shows the results of a complete set of runs across all sampling parameters for the `N-bt` benchmark, showing the runtime error compared to a full-detailed simulation at each (D, F) combination. The graph also shows, for the three different periodicities that occur in this application, the iso-lines where $D = P$ (vertical lines) and $D + F = P$ (diagonal lines). On the right side of the graph, for $D > 50$ ms, the detailed region length is longer than all of the application's periodicities, here sampling works well as the error is low (<3%). Moving from the middle towards the lower left corner also increases accuracy, as this moves $D + F$ farther away from the two larger periodicities and thus increases the number of intervals taken inside each period. On the other hand, sampling parameters close to $D = P$ or $D + F = P$ yield much higher errors, up to 15% even for conservative amounts of fast-forwarding. Additionally, the area in between the $P = D$ and $P = D + F$ region also does poorly. Here, an interval is measured that represents part of the application's main period, but this interval is subsequently used in fast-forwarding across multiple iterations of this period. This can introduce large errors when the IPC of the collected interval is not representative of the IPC of the whole period. It therefore makes sense that

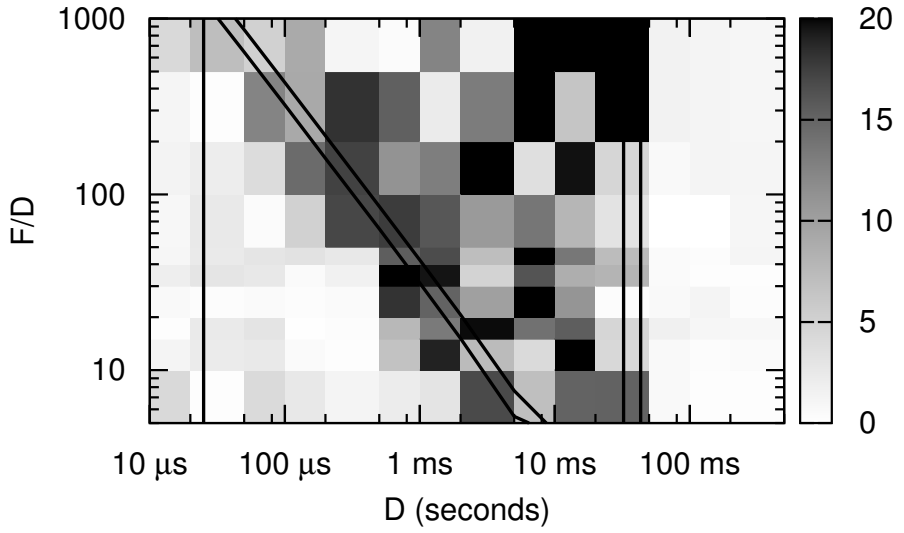


Figure 4.8: Sampling error versus application periodicity for `N-bt`, class A input set with 8 threads. Also shown are the periodicities of the application (solid lines).

we would not want to collect intervals using parameters in this region.

Converting Instruction Periodicities to Time

Although both methods for determining instruction periodicity described in the previous section yield a (micro-architecture independent) result expressed in instructions, our method of multi-threaded sampling requires its parameters to be expressed in time. Since we will only use these periodicities to define *forbidden zones* for these parameters, we do not need the periodicity's exact length in cycles. Instead, we assume the benchmark will be executed at an IPC of between 0.5 and 2.0, which are typical long-term average IPC values for the benchmarks used. Each periodicity P thus becomes a range of $[0.5P \ldots 2.0P]$. Note that the four-way issue out-of-order processor core we model regularly achieves an IPC close to four; this is usually only for short periods whereas the long-term averages can be much lower, see also Figure 4.3.

Table 4.1: Simulated system characteristics.

Component	Parameters
Processor	2 sockets, 4 cores per socket
Core	2.66 GHz, 4-way issue, 128-entry ROB
Branch predictor	Pentium M [64], 17 cycles penalty
L1-I	32 KB, 4 way, 4 cycle access time
L1-D	32 KB, 8 way, 4 cycle access time
L2 cache	256 KB per core, 8 way, 8 cycle
L3 cache	8 MB per 4 cores, 16 way, 30 cycle
Main memory	65 ns access time, 8 GB/s per socket

4.4 Experimental Setup

4.4.1 Simulation Configuration

For the results shown in this chapter, we used the Sniper multi-core simulation infrastructure [11]. We configured it to model a multi-core out-of-order processor resembling the Intel Nehalem processor, see Table 5.1 for its main characteristics. The benchmark suites used in this chapter are the SPEC OpenMP (medium) suite (*train* inputs) [5], the NAS Parallel Benchmarks version 3 with OpenMP parallelization (*class A* inputs) [41], and the PARSEC 2.1 benchmark suite (*simlarge* inputs) [7]. We refer to the benchmarks from these suites using the `O-*`, `N-*` and `P-*` notations, respectively. Only the parallel Region of Interest (ROI) of each application is included in our measurements; fast-forwarding (with functional modeling of caches and branch predictors enabled) was used to skip over the (sequential) initialization and cleanup phases. The passive OpenMP wait policy was used for thread synchronization. All benchmarks were compiled with GCC 4.3 for x86_64 with SSE2 extensions enabled.

4.4.2 Implementing Sampled Simulation in Sniper

We implemented multi-threaded sampling as detailed in Section 4.2 in Sniper, building on its existing detailed and cache-only simulation modes. Sniper uses the Pin dynamic instrumentation framework [48] as its functional simulation front-end. Pin is instructed to add analysis routines, which send detailed instruction information to Sniper's timing models. By changing which analysis routines are enabled, one can efficiently switch into a functional simulation-only mode which runs at near-native execution speed (by adding no analysis routines), or simulate just caches and branch predictors for functional warming (by instrumenting only memory operations and branch instructions). The latter mode is used in this chapter during the fast-forward phases between detailed intervals.

A sampling director, which we added to Sniper, takes the length of the detailed and fast-forward intervals as input. Once the region of interest begins, it starts out in detailed mode and runs the simulation for the required amount of simulated time to complete one detailed interval. Note again that all intervals are expressed in absolute time (seconds), which is required to keep track of simulation modes consistently across threads when they execute instructions at different speeds, or even run at different clock frequencies. When the detailed interval completes, the simulation director computes the non-idle IPC over the preceding interval for each thread based on the instructions it executed and the time it did *not* sleep waiting for synchronization events.

The simulator is then switched into functional warming mode. Only a small amount of instrumentation is needed here (one analysis function per basic block) to be able to keep track of instruction counts; these are used to increment each thread's time (using its current fast-forward IPC). Synchronization events (`pthread_mutex`, futex system calls, etc.) still happen as before, i.e., threads waiting on them do not execute instructions and do not advance time, but inherit the time of the thread which later wakes them up. Once all (non-sleeping) threads have advanced in this way to the end of the fast-forward interval, a new detailed interval starts.

During fast-forwarding, as in detailed mode, barrier synchronization is used periodically to ensure threads make forward progress at roughly the same pace. This is especially important when keeping cache simulation enabled, to make sure the ordering of memory references — and their resulting performance impact due to for instance associativity conflicts among threads sharing a last-level cache — are simulated accurately.

At the end of the simulation, since every thread has kept track of time, the time of the last thread to finish will be equal to the application's total runtime; no further computation or extrapolation on the results is needed. In addition, per-thread idle times can be kept and the application's synchronization overhead or load imbalance can be derived without extra effort.

4.4.3 Selecting Sampling Parameters

Sample selection during simulation is done periodically using fixed parameters for the detailed (D) and fast-forward interval lengths (F), both are expressed in seconds. These parameters are determined up-front based on application periodicity obtained from BBV and call structure information.

The methodology described in Section 4.3 defines forbidden ranges (D and $D + F$ close to one of the application's periods or its end). We start by converting all periods P_i that were found, and the application length L (in-

struction count for the longest thread), into a range of cycle counts in which we expect these values to lie using an expected IPC range of $[0.5 \ldots 2.0]$. Multiplying this range with the clock frequency of the simulated processor yields a lower $\underline{\bullet}$ and an upper bound $\overline{\bullet}$ expressed in seconds for each of these application characteristics. We then enumerate all possible D and F combinations, and remove those which do not satisfy the following conditions:

$$\begin{array}{l} D > \alpha \cdot \overline{P_i} \quad \vee \quad (D+F) \cdot \beta < \underline{P_i}, \quad \forall P_i \\ (D+F) \cdot \gamma < \underline{L} \end{array} \tag{4.1}$$

The constants α, β and γ used were $\alpha = 2$ (for *outside* sampling, where the detailed interval is larger than one period: at least two iterations per detailed interval), $\beta = 25$ (for *inside* sampling, where multiple intervals lie inside one application period: at least 25 intervals per iteration) and $\gamma = 10$ (at least 10 intervals in total).

We rank all of the remaining options, maximizing their distance from the closest periodicity or the end of the application, since a maximum distance yields the lowest potential for error as discussed in Section 4.3.4. We then select two points: *predicted fastest* which is the one with the highest ratio of F/D, and *predicted most-accurate* which is the point with the largest minimum distance and a fast-forward interval of at least $F \geq 5 \cdot D$.

Since the potentials for error of *inside* and *outside* sampling are not easily comparable, we select a set of parameters for each of them. For results in Section 4.6, where only one parameter set is used, we prefer *outside* sampling for those benchmarks where a valid point is available, and use *inside* sampling otherwise. *Outside* sampling is preferred because it increases an individual sample's accuracy by averaging the IPC over a number of application periods.

It can be the case that appropriate sampling parameters cannot be found for a particular combination of benchmark, input size and core count. In these cases, the resulting configuration options would provide either minimal speedup or too high of an expected error. We consider these configurations not able to be sampled with the proposed methodology.

4.5 Results

In this section, we will evaluate the proposed sampling methodology with respect to fully-detailed (non-sampled) runs. We first review the entire sampling space for a single application. Next, we review the accuracy and performance trade-offs for all applications that have valid sampling parameters and compare the parameter sets selected by the *predicted fastest* and *predicted most-accurate* methods.

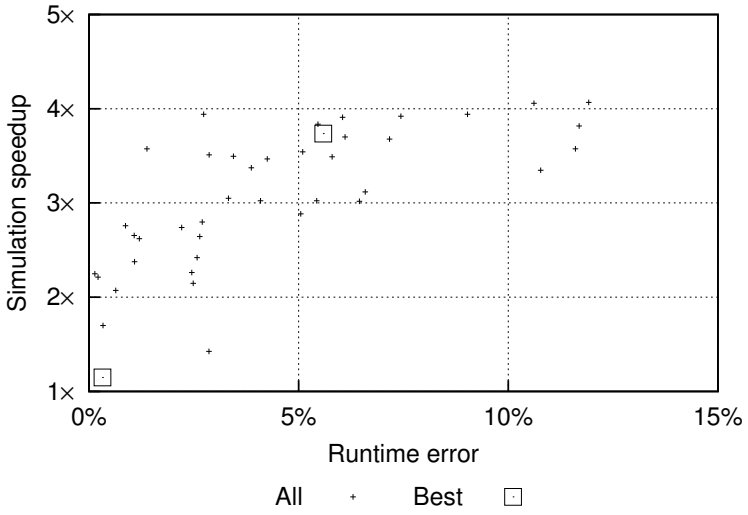


Figure 4.9: Simulation speedup versus accuracy for all valid sampling parameters, with those selected by the methodology marked. `O-apsi`, train input set, 8 threads.

4.5.1 Sampling Parameter Space

Figure 4.9 compares the simulated runtime error and simulation speedup, both compared to a fully-detailed simulation of the `O-apsi` application, for a wide range of sampling parameters. The `O-apsi` benchmark was chosen here because it has a large number of sampling opportunities available. In this graph, the methodology selected two points as the *predicted fastest* and *predicted most-accurate* options. The fastest option, as defined above, resulted in a 3.74× speedup with a 5.59% error. The most accurate result was chosen to be conservative and achieved a 1.15× speedup with an error of 0.32%. For this application, our selection comes close to predicting Pareto-optimal results.

4.5.2 Predicting Optimal Sampling Parameters

In Figures 4.10 and 4.11, we detail the results when selecting the best options as predicted by the methodology for both *inside* and *outside* sampling for the *predicted most-accurate* and *predicted fastest* parameter sets respectively. Note that not all of the benchmarks have valid sampling options either because of their internal periodicities or a short application runtime. The average absolute error for applications with valid sampling periodicities using the *predicted most-accurate* method is just 3.5% with an average speedup of 2.9×. The maximum speedup achieved is 5.8× faster than full-detailed simulation for an 8-core architecture. The *predicted fastest*

Table 4.2: Overview of all benchmarks, their periodicities, the chosen sampling parameters and their speed and accuracy.

Application	periodicities (ins)	length (ins)	in/outside	D	F/D	error	speedup	sampled simulation time
O-ammp	*non-periodic behavior*							
O-applu	603k, 104M	38.1B	outside	500 ms	5×	-2.75%	3.37×	21.96 h
O-apsi	676M	48.8B	inside	10 μs	5×	-0.32%	1.15×	69.89 h
O-art	1.40M	110M	*no valid range*					
O-equake	3.66M, 9.00M	17.3B	outside	200 ms	5×	-8.61%	1.63×	19.88 h
O-fma3d	354k, 99.8M	36.2B	outside	500 ms	5×	-0.26%	3.49×	35.27 h
O-gafort	17.0k, 34.3M	10.2B	outside	100 ms	5×	11.15%	2.80×	4.85 h
O-galgel	3.36M, 5.60M, 548M	64.4B	*no valid range*					
O-mgrid	60.5M	61.8B	outside	500 ms	10×	-3.90%	3.83×	41.28 h
O-swim	26.4M	21.8B	outside	200 ms	5×	0.94%	2.18×	72.19 h
N-bt	140k, 180M, 241M	52.7B	outside	500 ms	5×	-0.36%	2.38×	26.81 h
N-cg	2.20M, 56.8M	860M	*no valid range*					
N-ep	420k, 14.2M	7.04B	outside	100 ms	5×	0.06%	4.37×	2.13 h
N-ft	550k, 1.14M, 449M	3.11B	*no valid range*					
N-is	25.0M	333M	inside	10 μs	5×	0.49%	1.88×	0.33 h
N-lu	*non-periodic behavior*							
N-mg	95.0k, 146M, 292M	1.26B	*no valid range*					
N-sp	60.4M	27.0B	outside	200 ms	10×	-1.61%	1.98×	27.91 h
N-ua	1.89M	30.0B	outside	200 ms	10×	-6.02%	4.39×	11.13 h
P-blackscholes	4.08M, 4.56M, 5.60M, 6.36M	712M	outside	10 ms	5×	0.56%	2.58×	0.30 h
P-bodytrack	138k	2.74B	outside	20 ms	10×	-11.52%	3.22×	0.70 h
P-canneal	200k	250M	outside	2 ms	10×	-1.31%	1.61×	0.38 h
P-dedup	—	17.5B	outside	200 ms	5×	3.29%	2.79×	4.10 h
P-facesim	6.36M, 19.2M, 32.0M	3.44B	outside	50 ms	5×	-8.52%	1.78×	2.75 h
P-ferret	13.2M	12.7B	outside	100 ms	10×	-0.07%	3.08×	2.66 h
P-fluidanimate	584M	3.03B	inside	10 μs	5×	0.71%	1.02×	2.40 h
P-freqmine	—	6.01B	outside	50 ms	10×	1.83%	4.59×	1.52 h
P-raytrace	50.0k	1.22B	outside	10 ms	10×	-5.45%	5.76×	0.23 h
P-streamcluster	—	2.93B	outside	20 ms	10×	7.47%	2.96×	1.00 h
P-swaptions	200k	2.47B	outside	20 ms	10×	3.82%	3.73×	0.93 h

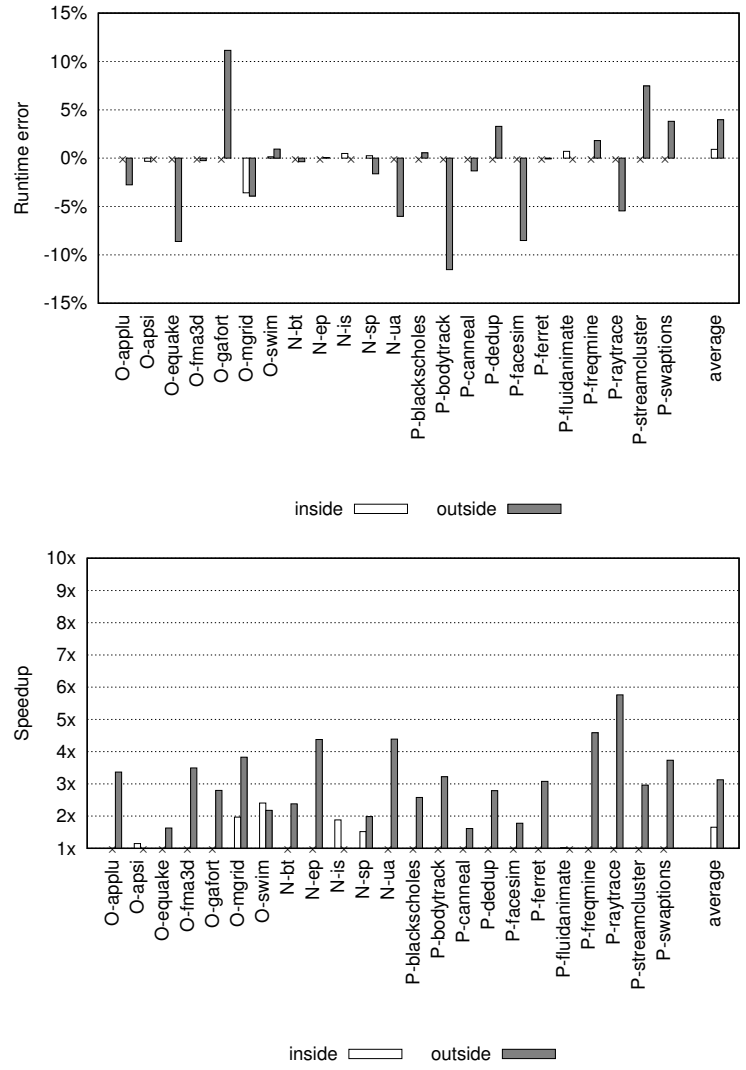


Figure 4.10: Overview of sampling accuracy and speedup using the *predicted most-accurate* parameter set, for both *inside* and *outside* sampling when available.

method achieves a maximum speedup of 8.4× with an average speedup across applications with valid sampling parameters of 3.8× and an average absolute error of 5.1%. With many applications seeing a 10× F/D fast-forwarding ratio, we are therefore simulating just 9.1% of the application in detail (one detailed period followed by 10 fast-forwarding periods). While other single-threaded sampling techniques can achieve a much larger speedup, the speedup in our methodology is limited by two factors: the complexity of the multi-core memory hierarchy models, which is enabled both during fast-forwarding and detailed intervals, and the relatively

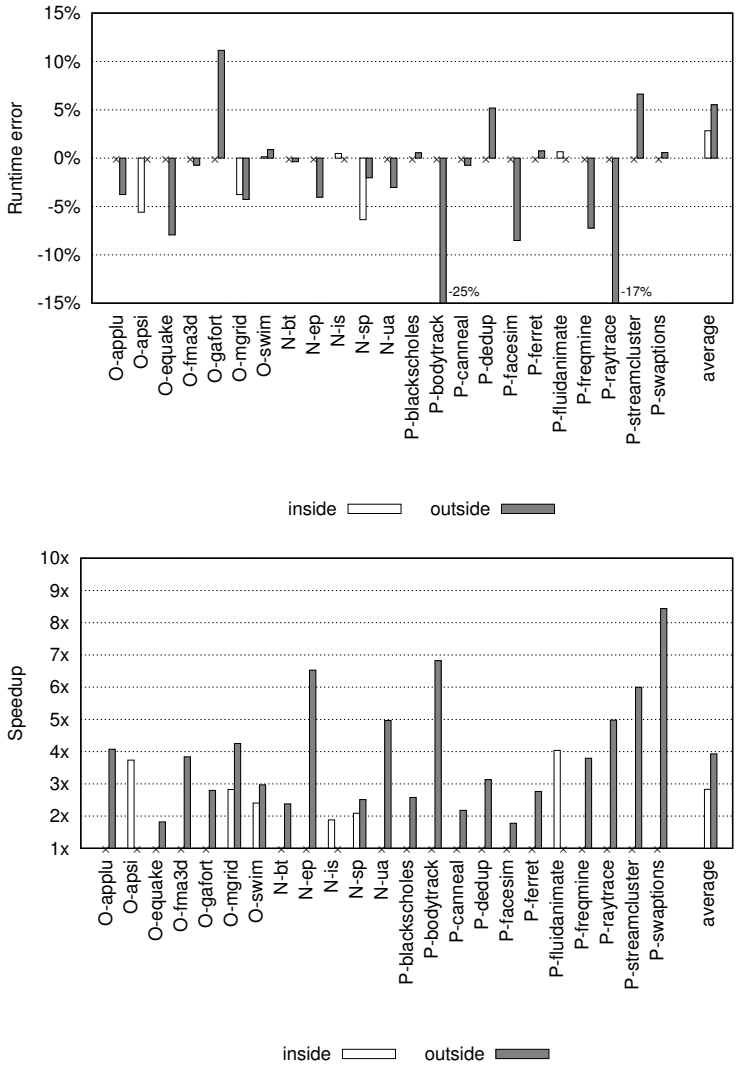


Figure 4.11: Overview of sampling accuracy and speedup using the *predicted fastest* parameter set, for both *inside* and *outside* sampling when available.

high speed of our core model. See Section 4.5.5 for a detailed discussion on speedup potential.

Table 4.2 lists application periodicities found for each benchmark, and the parameters that were used when sampling them for the *predicted most-accurate* case. Two benchmarks, `O-ammp` and `N-lu` were excluded a priori according to the analysis made in Section 4.3.3, while for five more benchmarks their periodicities were such that no valid sampling parameters could be found that satisfied the constraints of Equation 4.1.

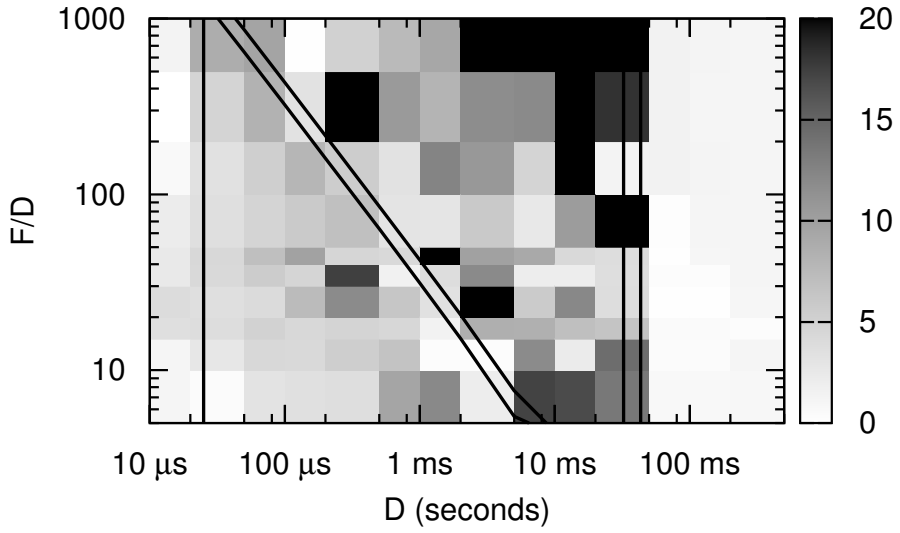


Figure 4.12: Sampling error versus application periodicity for `N-bt`, class A input set with 8 threads, and random placement of the detailed interval within each D+F region.

4.5.3 Random Sampling

When sampling periodic signals, random sampling is often used to avoid aliasing. The underlying idea is that each sampling interval covers just part of the period, but collectively, the average of all intervals approaches the average of the signal. In our methodology, however, we require each single detailed interval to be representative for the current IPC as this IPC is used during the subsequent fast-forwarding phase. When the application synchronizes, it is the slowest thread which determines progress — application runtime is therefore not determined by the sum of all intervals (allowing high and low estimates to cancel each other out), but has a more complex relationship in which at several points the maximum value of a set of intervals determines progress. The central limit theorem is therefore not applicable, as discussed in Section 4.3.1.

Figure 4.12 revisits the experiment of Figure 4.8, but implements random sampling. The execution is divided into intervals of size $D + F$, the detailed intervals (again of size D) are placed at a random position within this interval. This randomizes the sampling period, while making sure that no fast-forward interval becomes larger than $2F$ (larger effective F lengths would extrapolate the detailed IPC for too long, causing additional error).

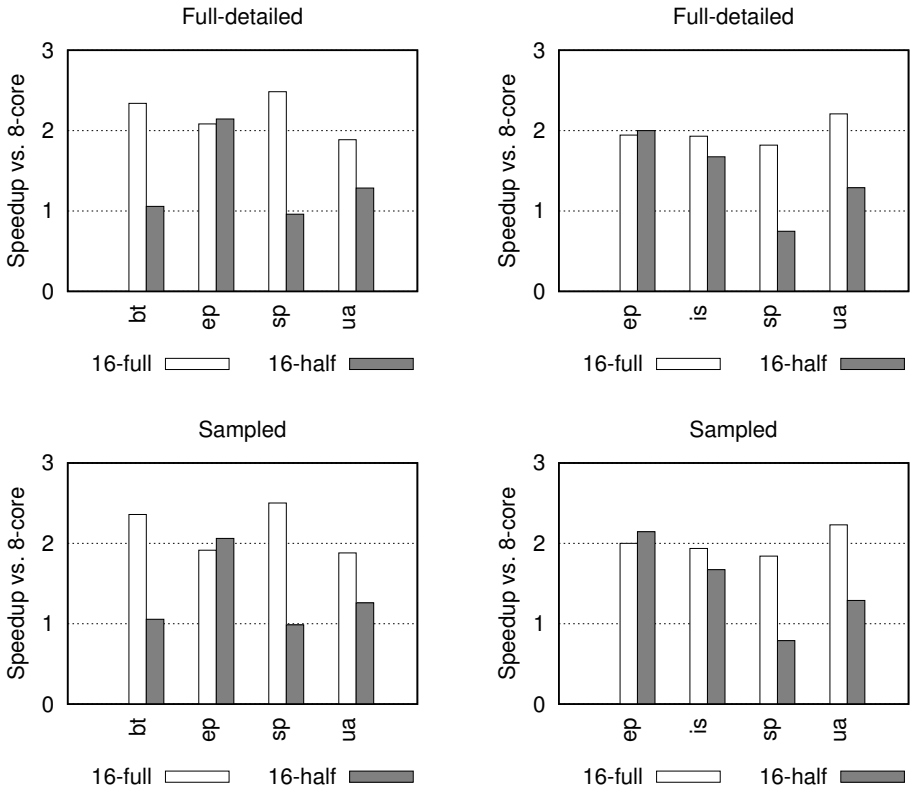
Although some regions in the (D, F) design space have become slightly more accurate when compared to periodical sampling in Figure 4.8, accuracy is still largely dependent on the relation between the sampling parameters and application periodicity. Random sampling can therefore be used as an extra component to increase overall accuracy, but it does not relieve one from knowing about application periodicities and designing the sampling parameters accordingly.

4.5.4 Detailed Warmup

In the SMARTS methodology [71], in addition to continuous functional warming of caches and branch predictors, a detailed core warmup step was required to minimize the cold-start bias of the core model. Following their analysis, the maximum life-time of stale state inside the core could be computed as the product of store-buffer depth, memory latency in cycles, and the maximum IPC. For our configuration, this upper bound is 25,600 (32 $\times$ 200 $\times$ 4) instructions. In our methodology for sampling multi-threaded applications, however, detailed intervals much longer than SMART's 1,000 instructions are favored. This makes the detailed region (very) long in comparison to the potential cold-start effects, negating the need for a separate detailed warming phase. Simulation results confirmed this: even for scenarios with a 10 µs detailed period, the shortest considered, adding a detailed warmup period of as much as 10 µs (approximately 10k-100k instructions) did not cause a change in run-time predictions beyond the expected run-time variability.

4.5.5 Potential for Simulator Speedup

Simulation results presented in this study use Sniper's *interval* core model, which is significantly faster to simulate than detailed core models found in other academic simulators [31]. In addition, its memory model is relatively complex as it supports shared caches in a parallelized simulation platform. This makes the ratio of execution speed in detailed mode versus that of functional warmup rather low, around 5–10$\times$ depending on the application. In SMARTS, this ratio was much higher (around 50–100$\times$), due to its complex (8- and 16-way issue) core models and a simpler, single-core memory hierarchy. This ratio directly affects the potential speedup that can be obtained from sampled simulation: as the fraction of the application simulated in detail is reduced, the simulation speed asymptotically reaches that of functional warming. Any additional gains in simulation speed will have to be made by relaxing the continuous functional warmup requirement, which is an open research problem [6].



(a) Class A input set (b) Class B input set

Figure 4.13: Results of the architectural exploration study: speedup over the baseline architecture for all benchmarks, A (top) and B (bottom) input sets, comparing full-detailed (left) with sampled (right) simulation.

4.6 Application: Architectural Exploration

In architectural exploration, a simulator needs to have high fidelity with respect to architectural changes, whereas absolute accuracy against any given architecture is less important. Figure 4.13 shows the results of an experiment where we compare our baseline, 8-core architecture with two 16-core architectures: one is a straightforward doubling of cores and cache sizes (*16-full*), whereas the alternative architecture keeps cache sizes constant but splits each core into two dual-issue out-of-order cores (*16-half*). For those applications in the NPB benchmark suite that had valid sampling parameters, we simulated the class A and B input sets on all architectures using both full-detailed and sampled simulation and plot the simulated speedup achieved over the baseline architecture.

Comparing the full-detailed (top) with the sampled (bottom) graphs, it is clear that our sampling methodology has a good fidelity with respect to

architectural changes, as it preserves the performance differences between the architectures for all benchmarks and input sets shown.

On the other hand, attempting to speed up simulation by using smaller input sets does not have the same fidelity. For instance, using the A input set, the `N-sp` benchmark achieves a superlinear speedup of around 2.5× when going from the baseline 8-core architecture to the 16-full architecture — whereas the `N-ua` benchmark sees a normal 2× speedup. However, on the larger class B input set, this trend is reversed: here `N-ua` has a (slightly) superlinear speedup whereas `N-sp` achieves less than linear scaling. Clearly, reducing input set size is not an accurate method when doing architecture exploration studies. However, running the larger class B input sets in full-detailed mode takes several weeks, whereas our sampling methodology can bring down this simulation time by a factor of 2.6× while still allowing the correct conclusions to be made.

4.7 Related Work

Below we discuss prior work that is most closely related to this work.

4.7.1 Single-Threaded Sampling

The SMARTS [71] methodology constructs a sample consisting of a large number of intervals (10,000) of a relatively small number of instructions (1000) per detailed region. They are able to estimate IPC very well because with large numbers of intervals per sample, the average IPC error for the application as a whole decreases — even when each individual interval may suffer from aliasing and is therefore by itself not reliable (the central limit theorem applies here). In our sampling methodology, we require each detailed region to be an accurate representative of its $D + F$ time slice, and extrapolate run-time from it. Because of thread synchronization, we cannot rely on averaging to counter aliasing and the central limit theorem is not applicable, see Section 4.3.1.

SimPoint [61] clusters large intervals, on the order of 100M instructions, using BBVs to represent common chunks of an application in a microarchitecture-independent way. Although SimPoint allows one to accurately predict overall application IPC, it does not allow for the prediction of multi-threaded application run-time, nor does it take application synchronization into account during fast-forwarding.

COTSon [4] uses dynamic sampling to speed up simulation. It uses feedback from the JIT engine of the SimNow simulator to react to changes in the executed code and to switch out of fast-forwarding back into detailed simulation when necessary. Their implementation was used and evaluated

on single-threaded applications, and is similar to the SimPoint methodology with respect to running long intervals (100 million instructions) without functional-warming during fast-forward phases.

4.7.2 Multi-threaded Sampling

Ekman et al. [24], propose matched-pair comparison as a way to reduce the number of simulation points required to gain an accurate understanding of multi-threaded workloads. Matched-pair comparison relies on the assumption that threads are independent and not synchronized to be able to reduce the sample size. Ekman et al. showed that for their methodology, synchronized applications, such as those in the Splash-2 suite, do not see a significant sample size reduction.

Wenisch et al. [69] propose Flex Points as a way to increase simulator performance for multi-threaded commercial workloads. Van Biesbrouck et al. [66] propose the Co-Phase Matrix as a reduction technique for multi-program workloads. Both of these techniques depend on the fact that each thread is independent, and therefore over time, a sample will contain a number of thread interleavings that can act as a representation for the overall system execution. Explicit thread synchronization through OS or architected instructions prevents these thread interleavings from occurring, which violates their assumptions. Additional details are discussed in Section 4.2.3.

Perelman et al. [55] cluster multi-threaded applications by looking at each thread in isolation. They predict the IPC and cache hit rates of clustered application phases, but do not evaluate runtime error.

Ryckbosch et al. [57] use interval simulation as a core model in the COTSon [4] simulator and compare their sampled simulation results directly to hardware. COTSon's sampling mechanism throttles functional simulation during fast-forwarding to ensure that relative thread progress, at a ratio corresponding to each thread's IPC, is observed during detailed simulation. The timing of synchronization events is considered to be part of the fast-forward IPC and is not considered independently during fast-forwarding. In Figure 4.2 we show that not taking into account the detailed interactions between threads during fast-forwarding can lead to significant runtime estimation errors. Our proposed implementation provides for thread-to-thread synchronization and shared cache hierarchy interactions that should be represented to obtain accurate runtime results. Additionally, COTSon does not perform functional-warming of caches, meaning that detailed NUMA behavior, for example, or other interactions through shared caches will be lost during fast-forwarding.

Concurrently with our work, Ardestani and Renau [3] propose time-

based sampling with integrated power and temperature evaluation, resulting in the development of the ESESC simulator. They establish fixed sampling parameters to minimize the coefficient of variation of IPC across a number of benchmarks. In contrast, our work demonstrates that application periodicity needs to be taken into account when selecting parameters for multi-threaded time-based application sampling.

4.8 Conclusions

Previous sampling work has primarily focused on either single-threaded, IPC-based runtime prediction methods or multi-threaded workloads where per-thread behavior is uncorrelated. Synchronizing multi-threaded applications, where threads affect each other's behavior directly, pose a challenge when it comes to accurately predicting runtime as the traditional sampling methods do not apply to these workloads.

To address these limitations we propose a general-purpose multi-threaded application sampling methodology. We show that taking into account application synchronization during fast-forwarding while determining progress in a per-thread manner significantly improves the prediction of application run-time. In addition to synchronization, application periodicity needs to be taken into account to prevent detailed sampling intervals from aliasing with the application's periodic behavior, affecting both the fast-forwarding IPC and overall runtime prediction. Through the use of micro-architecture independent methods of detecting periodicity we derive sampling parameters up-front to allow for accurate run-time prediction.

Using our sampling method inside Sniper, we are able to achieve a maximum speedup of 5.8× and an average speedup of 2.9× when simulating parallel applications running on 8-core processors while being able to predict application runtime with an average absolute error of 3.5%.

Chapter 5

BarrierPoint: Sampled Simulation of Multi-Threaded Applications

In the previous chapter, we describe a general-purpose sampling methodology that combines time-based sampling with application knowledge to reduce the amount of an application that needs to be simulated in detail. While this technique can help to speed up applications in the general case, the maximum speedup of the simulation is ultimately limited by the speed of the memory subsystem during non-detailed fast-forwarding phases.

In this chapter we take a look at how we can use application-specific behavior to improve sampling even further. We propose BarrierPoint, a sampling methodology to accelerate simulation by leveraging globally synchronizing barriers in multi-threaded applications. BarrierPoint collects microarchitecture-independent code and data signatures to determine the most representative inter-barrier regions. BarrierPoint then estimates the total application execution time (and other performance metrics of interest) through detailed simulation of the representative regions alone, leading to substantial simulation speedups.

5.1 Introduction

Sampling is a widely used technique to dramatically reduce simulation time by simulating a select number of sampling units in detail and extrapolating the results for the entire workload execution. Sampled simulation is a mature technology for single-threaded workloads running on individual cores, and different approaches have been proposed for determining representative sampling units: random [20], periodic [71], and through program analysis [61]. Sampled simulation for multi-threaded workloads on

the other hand is much less mature and substantially more complicated.

The fundamental problem in sampled simulation for multi-threaded workloads is to make sure all threads are aligned (i.e., all threads are at the same point in their execution) at the beginning of each sampling unit as if we were to simulate the entire execution up until the sampling unit. This is non-trivial because of how slight timing variations during the execution may affect per-thread progress either through synchronization behavior (e.g., locking) or resource sharing (e.g., shared caches, off-chip bandwidth, interconnection network, etc.). Moreover, making sure the same thread alignment occurs across microarchitectures is even more problematic.

Some classes of multi-threaded workloads do not pose this fundamental problem. For example, commercial server throughput workloads can be accurately sampled by randomly selecting sampling units [69]. This principle more generally applies to multi-threaded workloads in which the individual threads do not synchronize [24]. However, synchronizing multi-threaded applications are more challenging to sample for the reason mentioned above. Time-based sampling has been proposed for synchronizing multi-threaded workloads (Chapter 4 and [3; 12]), which simulates X units of time in detail every Y units of time, and estimates per-thread progress in-between sampling units. The fundamental limitation of time-based sampling is twofold. It requires functional simulation of the entire program execution to determine the sampling units, which limits the amount of speedup that can be achieved through sampling. In addition, it leads to different sampling units being selected across different simulated processor architectures, which complicates performance analysis.

Barrier-synchronized multi-threaded applications are an important subset of synchronizing parallel workloads, especially in the high-performance scientific computing and data-parallel workload domains, for which this fundamental problem in sampling can be overcome by selecting sampling units using barrier semantics. Barriers denote points of global synchronization in the applications at which all threads are naturally 'aligned', i.e., all threads re-start the execution at the same time once the barrier is reached by all threads. Bryan et al. [9] leverage this observation to speed up simulation of barrier-synchronized applications by simulating multiple inter-barrier regions in parallel on a cluster of simulation machines. This approach requires massive simulation resources to achieve substantial simulation latency reductions. Additionally, because the number of simulations to be performed inevitably outstrips the supply of available machines, maximizing overall simulation throughput becomes the overall goal. The only way to make faster progress is to improve the total simulation throughput across the entire parameter and benchmark space. Bryan et al. does not solve this critical issue: it only reduces latency of an isolated

simulation run, but it does not reduce the number of resources required nor overall simulation throughput when many simulations need to be run.

In this chapter, we propose BarrierPoint, a methodology for sampled simulation of barrier-synchronized multi-threaded machines that simulates a select number of representative inter-barrier regions, called *barrierpoints*, and predicts total application execution time (and other metrics of interest) from these barrierpoints. BarrierPoint computes code and memory access signatures for all inter-barrier regions, clusters regions based on these signatures, and then selects a single representative region, called a barrierpoint, per cluster. Barrierpoints are selected in a microarchitecture-independent way, and can therefore be used across processor architectures. BarrierPoint overcomes several major limitations in prior work. (i) It does not require functional simulation of the entire application as in time-based sampling; barrierpoints can be simulated in parallel. (ii) It leads to well-defined and fixed units of work — unlike time-based sampling — which facilitate comparisons across microarchitectures. (iii) It requires far less (one to three orders of magnitude fewer) simulation resources while achieving similar simulation speedups compared to the approach by Bryan et al. [9].

Specifically, we make the following contributions in this chapter:

- We propose a methodology for selecting representative, microarchitecture-independent inter-barrier regions in barrier-synchronized multi-threaded applications for sampled simulation. Barrierpoints enable microarchitectures and processor architectures (including different core counts) to be compared through sampled simulation using well-defined and fixed units of work.
- We propose a method to extrapolate and estimate total application execution time, and other performance metrics of interest, from this select set of barrierpoints.
- We explore different methods for characterizing inter-barrier regions and find that signatures that incorporate both code and data behavior are most accurate at identifying representative barrierpoints.
- We evaluate the BarrierPoint methodology using a set of NPB and Parsec benchmarks on 8 and 32-core machines, and report average speedups of 24.7× (maximum speedup of 866.6×) while maintaining an average error of 0.6% and maximum error of 2.8%. BarrierPoint reduces the number of simulation machine resources needed by 78× on average, compared to simulating all inter-barrier regions.
- We propose an easy-to-implement, fast and accurate cache warmup technique for multi-threaded sampling that incurs a combined sampling and warmup error of 0.9% on average and 2.9% at most.

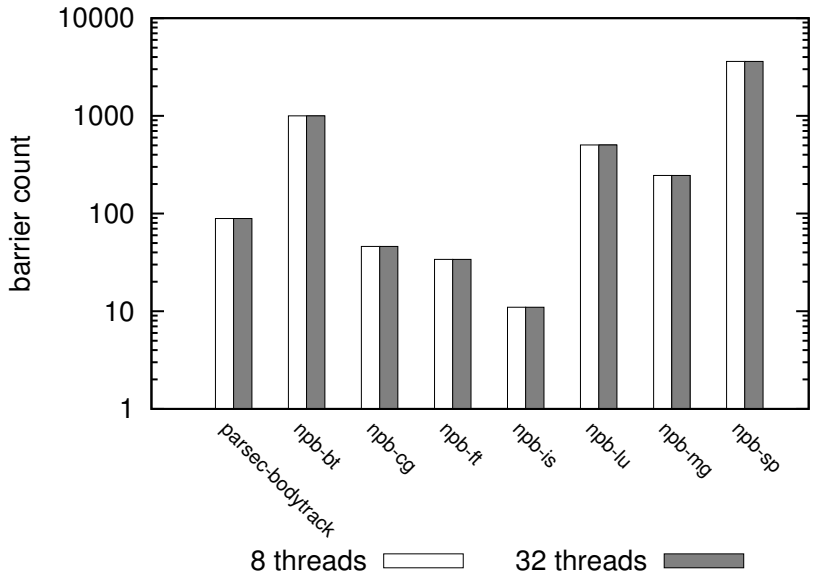


Figure 5.1: Total number of dynamically executed barriers.

5.2 Key Idea

Multi-threaded applications have been developed as a way to take advantage of the growing numbers of cores available in current machines. Prior work [3; 12] has shown that, as in single-threaded workloads, redundancy exists in the behavior of multi-threaded applications which allows detailed simulation of only part of the workload to be extrapolated to accurately predict execution time of the complete application. However, the only available techniques for accurate sampled simulation of synchronizing multi-threaded applications required functional simulation of the complete application, which limits the simulation speedup that can be obtained through sampling. In contrast, implementations of the popular SimPoint [61] and SMARTS [71] methodologies for sampled simulation of single-threaded applications are able to load the application's architected state at the start of each sampling unit from a checkpoint [65; 68]. This makes simulation of each sampling unit completely independent and potentially parallelizable, and negates the need for functional simulation of the complete application.

In synchronizing multi-threaded applications, however, it is not known a priori at what rate the execution of each individual thread will progress. Therefore, a checkpoint of architected state taken at a random time during execution of the application does not necessarily represent a valid situation that would occur when executing the same application on a different mi-

croarchitecture. Global synchronization barriers, on the other hand, represent points in each thread's instruction stream that denote a common point in time, and are therefore *safe* points at which a checkpoint can be taken. Figure 5.1 counts the number of barriers encountered during the execution of a number of applications in the NPB and Parsec benchmark suites. As is common for many data-parallel workloads, which are typically written in a structured manner using fork-join parallelism or other paradigms that lead to bulk-synchronous behavior, the absolute number of barriers is large, up to several 1000s in this case. Moreover, the number of barriers present remains constant even when changing the number of threads. This suggests that sections of code in between global barriers can be used as independent units of work.

By simulating an application separated into a number of inter-barrier regions in parallel, it is now possible to speed up simulation of an entire program. Assuming that a large enough amount of simulation resources are available, one can reduce the latency of executing a single application [9].

Taking this one step further, if we are able to classify — and therefore merge — similar inter-barrier regions into a single *representative* region, we could further reduce the number of computing resources required to obtain these speedups. We will call these representative regions *barrierpoints*, akin to *simpoints*, their single-threaded, instruction-delineated counterparts following the SimPoint [61] methodology. The final step that is required is to provide the warmup data for each barrierpoint. The end result is a complete simulation methodology that takes advantage of the ability to parallelize multi-threaded applications at a barrier-by-barrier granularity. With BarrierPoint, we have developed a methodology that allows for existing applications, without modification, to be simulated in parallel, using fewer simulation resources. Using global instruction count as a proxy for the amount of work in a simulation, we demonstrate a minimum, harmonic mean, and maximum speedup of 10.0×, 24.7× and 866.6×, respectively.

5.3 BarrierPoint Methodology

The BarrierPoint methodology (See Figure 5.2) provides a practical flow from a barrier-synchronized application to an estimation of performance metrics such as execution time or cache miss rates with a significant reduction in simulation time. The steps in the methodology include a one-time step where characteristics that represent inter-barrier regions of the application are collected, and clustering of the representative regions into barrierpoints for simulation occurs. These barrierpoints, together with the warmup data for each simulation is used to simulate each barrierpoint in

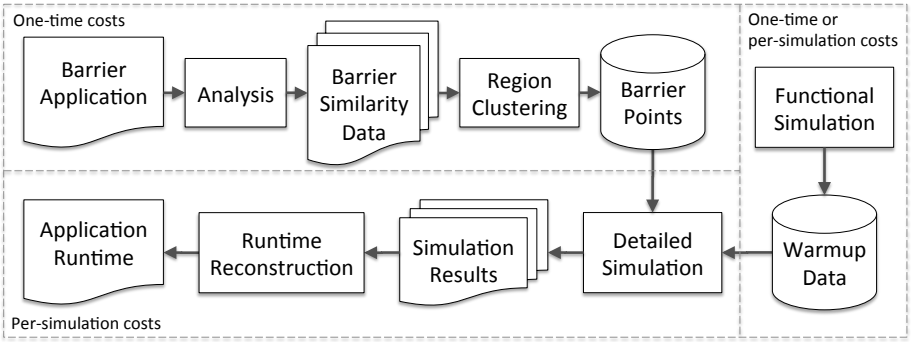


Figure 5.2: The BarrierPoint methodology flow diagram.

parallel. The final step is to reconstruct the performance metrics based on the representative barrierpoint simulation results. In the following sections, we will provide a detailed overview of each step in the methodology.

5.3.1 Barrier Region Similarity Metrics

Through the use of barriers, we have the ability to compare the instructions executed inside of inter-barrier regions for similarity. There are a number of different ways to classify the similarity between regions.

Basic Block Vectors

Traditionally, Basic Block Vectors (BBVs) [61] from fixed-instruction-length regions have been used as a way to easily classify or cluster regions of single-threaded applications. A basic block vector is a vector with an entry for the dynamic instruction count for each basic block in an application. During program execution, the number of instructions executed from each basic block is counted. After the completion of this region, the basic block vector is saved and a new vector is started for the following region. These BBVs form its fingerprint, or summary of the instructions that have executed during that region. Prior work has shown that BBVs relate strongly to region performance [44]. Therefore, by comparing the BBVs across inter-barrier regions, we can match the different regions to determine performance similarities for each application region.

LRU stack distance

LRU stack distance histograms [49] are another metric that can be used to classify program behavior. The LRU stack distance is the number of

unique address accesses that occur between two accesses to the same address. A histogram of these distance numbers represents a memory history footprint of a particular region of an application, and have been previously used to automatically detect phases in an application [60], as changes in the LRU stack distance profile is a way to evaluate the changing data reuse patterns of an application. LRU stack distances can therefore also be used for region classification. The intuition behind using LRU stack distances is that dynamic instruction regions, even though they are executing the same code and have the same BBV footprint, could experience different cache access latencies because of micro-architectural state. One example of this is the well-known cold-start effect, where the first few iterations of an application exhibit different progress than later, but BBV-similar, phases. Our goal, therefore, is to improve the accuracy of BBVs by combining them with LRU stack distance information when performing BarrierPoint clustering. To collect LRU stack distance information for the BarrierPoint methodology, we keep track of the reuse distances for each region of the application as it runs. The LRU stack distance data is stored in a power-of-two histogram, where we keep track of the frequency of each of the LRU stack distance accesses for each inter-barrier region. We will further refer to this histogram as the LRU stack distance vector (LDV).

Signature Vectors

To make an abstraction of the exact metric that is used, we define the Signature Vector (SV) as a representation of a region's phase behavior. Functionally, the Signature Vector consists of indices representing application characteristics, and the value of a vector element is the quantity or weight of that characteristic. In Section 5.6, we explore the clustering obtained by SVs consisting of BBV information only, LRU stack distance vectors (LDVs) only, and SVs that contain a combination of both. When both types of metrics are used, the BBV and LDV are normalized individually and then concatenated into a single, longer vector.

In addition, we evaluate the use of a weighing function to the LRU stack distance counters. Conceptually, longer stack distances will correspond to memory accesses that hit further away in the memory hierarchy and therefore have a larger impact on application performance. When comparing two regions with the aim of clustering them by having similar performance, we therefore want to give more importance to long-latency accesses. We will show results for unweighted distances, and variants where those elements corresponding to a distance of $2^n \ldots 2^{(n+1)} - 1$ are weighted by $2^{n/v}$, for different values of v.

Multi-threaded SVs

Both BBVs and LDVs are initially collected for each inter-barrier region on a per-thread basis. To combine these per-thread metrics into a single SV, two options are possible. One option is to sum the vectors, aggregating data from different threads into the same vector element. A second option is to concatenate vectors for each thread into a longer vector. This second option separates the behavior for each of the threads.

When threads behave in a homogeneous manner, both options will be equivalent. But, if threads have different behavior, a concatenated vector will expose these differences to the clustering phase which allows the relevant regions to be separated into different clusters. We therefore choose to use concatenation for combining SVs of different threads.

5.3.2 Region Clustering

To automatically determine the similarity among regions, we employ clustering. The first step is to normalize the SVs to allow the clustering phase to ignore each region's length, and to cluster regions based on their intrinsic characteristics. To reduce the computational demands of the clustering process, the dimensionality of the SVs is then reduced through random linear projection into 15 dimensions. In addition to the normalized SV, the region clustering process will use the region lengths, in aggregate number of instructions across cores, to weigh different regions such that more emphasis can be placed on those regions that represent a large fraction of total execution time. We then perform weighted k-means clustering on the randomly projected SVs to determine a reduced set of representative regions. When within a cluster multiple regions of similar characteristics but different length occur, weighing by instruction count will favor longer-running regions both in determining the cluster center, and in choosing the representative region.

Signature vector normalization, random projection and k-means clustering are compatible with the implementation of SimPoint [43] for variable-length regions. We are therefore able to leverage the existing SimPoint infrastructure when implementing BarrierPoint.

5.3.3 Detailed Region Execution

After we have defined the representative regions of the application, detailed simulation of each barrierpoint will provide the necessary information required to rebuild the application's execution time. Before detailed simulation can be started, both architected and microarchitectural state must be properly initialized. As the region starts with a barrier, it is possi-

ble to store the program's architected state in a checkpoint in a consistent manner across threads. Alternatively, functional simulation or direct execution can be used to fast-forward an application up to the barrier that marks the start of the region.

With respect to microarchitectural state, care must be taken to ensure caches are warmed sufficiently before the start of the detailed simulation. Several warmup options are possible and are compatible with the BarrierPoint methodology, see Section 5.4 for an overview of existing techniques, and the method used in our evaluation of BarrierPoint. Since barrierpoints are usually long (in the order of millions of instructions), detailed core or branch predictor warmup is not normally required.

5.3.4 Whole-Program Runtime Reconstruction

As variable-length barrierpoints are used as the basis for the identification of common regions, we need to add an additional step to properly map these representatives to each region to compute overall application execution time. This step is not required for the original SimPoint methodology as they are using fixed-length representative regions to determine application CPI, which in turn, can be used as a proxy for single-threaded application execution time. In contrast, the BarrierPoint methodology may cluster inter-barrier regions of differing lengths. Yet, as the clustering phase guaranteed that these regions have the same behavior, we can assume that their performance characteristics (when expressed as per-instruction metrics, e.g., cycles per instruction (CPI), cache misses per 1000 instructions (MPKI), etc.) are constant. We can therefore rebuild the original application by concatenating scaled (by relative global instruction count) versions of each barrier's representative region. We call the sum of scaling factors from each barrierpoint its *multiplier*. Put another way, the sum of the instruction counts from all of the inter-barrier regions that are represented by the barrierpoint is equal to the instruction count of the barrierpoint times the multiplier. This can be written as $\sum_{i=1}^{m} insncount_i = insncount_j \cdot mult_j$, where m is the number of regions that are represented by the jth barrierpoint. To calculate a metric of interest, we sum, over each barrierpoint, the metric weighted by the multiplier, $metric_{app} = \sum_{j=1}^{n} metric_j \cdot mult_j$, where $metric_j$ and $mult_j$ are the metric and multiplier, respectively, for the jth barrierpoint out of a total of n barrierpoints.

An example is provided in Figure 5.3 for `npb-ft`. The top graph plots aggregate IPC over time for the original, unsampled application. The bottom graph shows the different phases, and the result of the clustering step. The representatives for each region are marked in dark gray. The middle graph shows IPC as rebuilt by concatenating each region's representative. Aside from small differences, the representative is almost identical to the

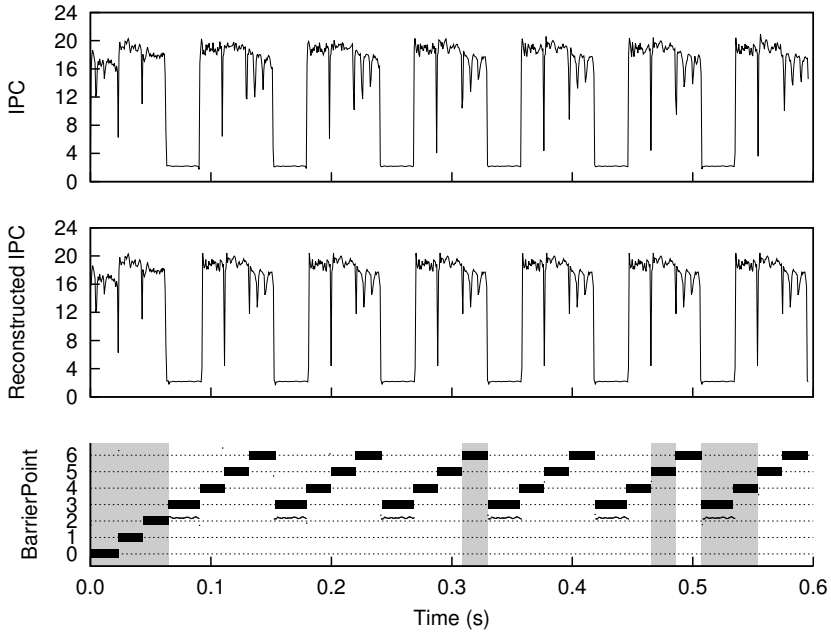


Figure 5.3: Aggregate application IPC (above), reconstructed IPC (middle) and the selected barrierpoints (below) for `npb-ft` using the *class A* input set on 32-cores. (Non-significant barrierpoints are omitted for clarity.)

original.

5.4 Micro-architectural State Reconstruction

There are a large number of micro-architectural warmup options available today [6; 21; 34; 65; 68]. Nevertheless, selecting a warmup strategy that is flexible, non-intrusive to the simulator and has a high speedup can be difficult. We propose a middle-ground for multi-threaded simulation cache warmup that maintains its speed and flexibility, while maintaining accuracy and being non-intrusive.

The two main micro-architectural warmup strategies are checkpointing and functional cache replay. Checkpointing tends to be the fastest warmup strategy as one only needs to load the amount of data that represents the state of the machine. But it tends to either be the least flexible, as naive checkpoints require a state snapshot for each micro-architecture and application combination, or require coherency-specific knowledge for accurately rebuilding the cache state. A more flexible but slower method is to run a functional simulation while updating microarchitecture state (e.g., issue memory requests to the cache hierarchy using a simple core timing model). This has the disadvantage that the execution time overhead is proportional

Component	Parameters
Processor	1 and 4 sockets, 8 cores per socket
Core	2.66 GHz, 4-way issue, 128-entry ROB
Branch predictor	Pentium M (Dothan) [64], 8 cycles penalty
L1-I	32 KB, 4 way, 4 cycle access time
L1-D	32 KB, 8 way, 4 cycle access time
L2 cache	256 KB per core, 8 way, 8 cycle
L3 cache	8 MB per 8 cores, 16 way, 30 cycle
Main memory	65 ns access time, 8 GB/s per socket

Table 5.1: Simulated system characteristics.

Parameter	Value
-dim (number of projected dimensions)	15
-maxK (maximum number of clusters)	20
-fixedLength (clusters are not normalized)	off
-coveragePct (percent coverage)	1 (100%)

Table 5.2: SimPoint parameters. Default values used for those options not specified.

to the number of instructions during the warmup period.

Instead of either of these extremes, we extended prior work [21; 68] that saves and then replays only the most recent unique address access requests. We first dynamically instrument an application and run them at near-native speeds (using a Pintool with only a 20× to 30× slowdown compared to native execution) to capture the most recently used data on a per-core basis. Each core keeps track of its most-recently used cache lines with a total capacity equal to the size of the shared LLC. Next, each thread replays their most recent access data in execution order to restore the state of the caches. The result is a significant reduction in simulated warmup replay time, as the size of the replayed cache state is on the order of the total LLC cache size and not based on the number of dynamically executed instructions up to this point.

5.5 Experimental Setup

In this chapter, we are using a modified version of the Sniper multi-core simulator [11], version 5.0, updated with the IW-Centric core model as described in Chapter 3. Each processor socket that we model is an octo-core processor with a three-level cache hierarchy, in which the L1 and L2 caches are private per core, and the last-level L3 cache is shared among all cores. Each core is 4-wide superscalar running at 2.66 GHz. We simulate both single-socket and four-socket shared-memory machines; we assume

Application	input size	num cores	total num barriers	significant barrierpoints	num insignificant barrierpoints, combined multiplier, and total weight	barrierpoint number and multiplier				
npb-bt	A	8	1001	11	3 / 12.0 / 8.9e-04	3 (1.0)	144 (190.0)	172 (100.0)	407 (46.0)	448 (98.0)
						463 (48.0)	521 (200.0)	537 (54.0)	595 (10.0)	980 (189.0)
						993 (53.0)				
		32	1001	11	4 / 13.0 / 9.2e-04	195 (10.0)	196 (100.0)	199 (189.0)	220 (189.0)	327 (46.0)
						408 (100.0)	552 (65.0)	733 (100.0)	771 (100.0)	787 (54.0)
						802 (35.0)				
npb-cg	A	8	46	3	7 / 29.1 / 4.4e-04	0 (1.0)	15 (12.0)	21 (2.0)		
		32	46	3	5 / 28.3 / 5.0e-04	3 (4.0)	6 (7.0)	30 (4.0)		
npb-ft	A	8	34	9	3 / 7.0 / 1.5e-05	0 (1.0)	1 (1.0)	2 (1.0)	3 (1.0)	11 (3.0)
						15 (6.0)	19 (6.0)	26 (3.0)	28 (5.0)	
		32	34	9	3 / 7.0 / 1.7e-05	0 (1.0)	1 (1.0)	2 (1.0)	3 (1.0)	16 (3.0)
						25 (6.0)	28 (5.0)	29 (6.0)	31 (3.0)	
npb-is	A	8	11	10	1 / 1.0 / 5.9e-07	0 (1.0)	1 (1.0)	2 (1.0)	3 (1.0)	4 (1.0)
						5 (1.0)	6 (1.0)	7 (1.0)	8 (1.0)	9 (1.0)
		32	11	10	1 / 1.0 / 7.0e-07	0 (1.0)	1 (1.0)	2 (1.0)	3 (1.0)	4 (1.0)
						5 (1.0)	6 (1.0)	7 (1.0)	8 (1.0)	9 (1.0)
npb-lu	A	8	503	7	1 / 2.0 / 8.8e-05	60 (10.0)	98 (68.0)	122 (53.0)	195 (250.0)	222 (16.0)
						282 (56.0)	332 (47.0)			
		32	503	2	0 / 0.0 / 0.0e+00	11 (250.1)	276 (250.0)			
npb-mg	A	8	245	8	0 / 0.0 / 0.0e+00	2 (2.0)	52 (4.6)	57 (9.0)	116 (4.6)	123 (4.6)
						179 (4.6)	182 (17.6)	230 (4.7)		
		32	245	10	5 / 112.2 / 9.8e-04	2 (2.0)	50 (4.8)	51 (4.6)	63 (4.0)	112 (4.6)
						116 (4.0)	117 (9.0)	179 (4.6)	185 (4.6)	238 (13.0)
npb-sp	A	8	3601	16	2 / 2.0 / 3.5e-04	0 (1.0)	159 (400.0)	238 (137.0)	607 (91.0)	1111 (56.0)
						1478 (399.0)	1813 (51.0)	1948 (65.0)	1970 (20.0)	1976 (399.9)
						2169 (399.0)	2465 (379.9)	2595 (399.9)	2746 (399.9)	3004 (202.0)
						3319 (198.0)				
		32	3601	12	0 / 0.0 / 0.0e+00	502 (400.0)	784 (94.0)	850 (400.0)	955 (200.0)	964 (106.0)
						2069 (400.0)	2157 (21.0)	2643 (379.0)	3360 (400.0)	3456 (400.0)
						3497 (400.0)	3584 (400.0)			
parsec-bodytrack	large	8	89	13	1 / 1.0 / 5.4e-04	12 (1.0)	21 (3.0)	25 (7.0)	29 (16.0)	39 (16.0)
						40 (5.0)	60 (2.0)	62 (8.0)	72 (9.0)	74 (12.0)
						77 (4.0)	80 (4.1)	87 (1.0)		
		32	89	7	1 / 1.0 / 5.4e-04	6 (16.0)	30 (12.0)	32 (16.0)	39 (16.0)	65 (4.0)
						77 (4.0)	86 (19.5)			

Table 5.3: Applications, input sizes used, total number of dynamically executed barriers, significant and insignificant barrierpoint information and the selected barrierpoints and their multipliers. Note that because of scaling a barrierpoint can represent a fraction of another inter-barrier region and therefore multipliers do not necessarily sum to the total number of regions.

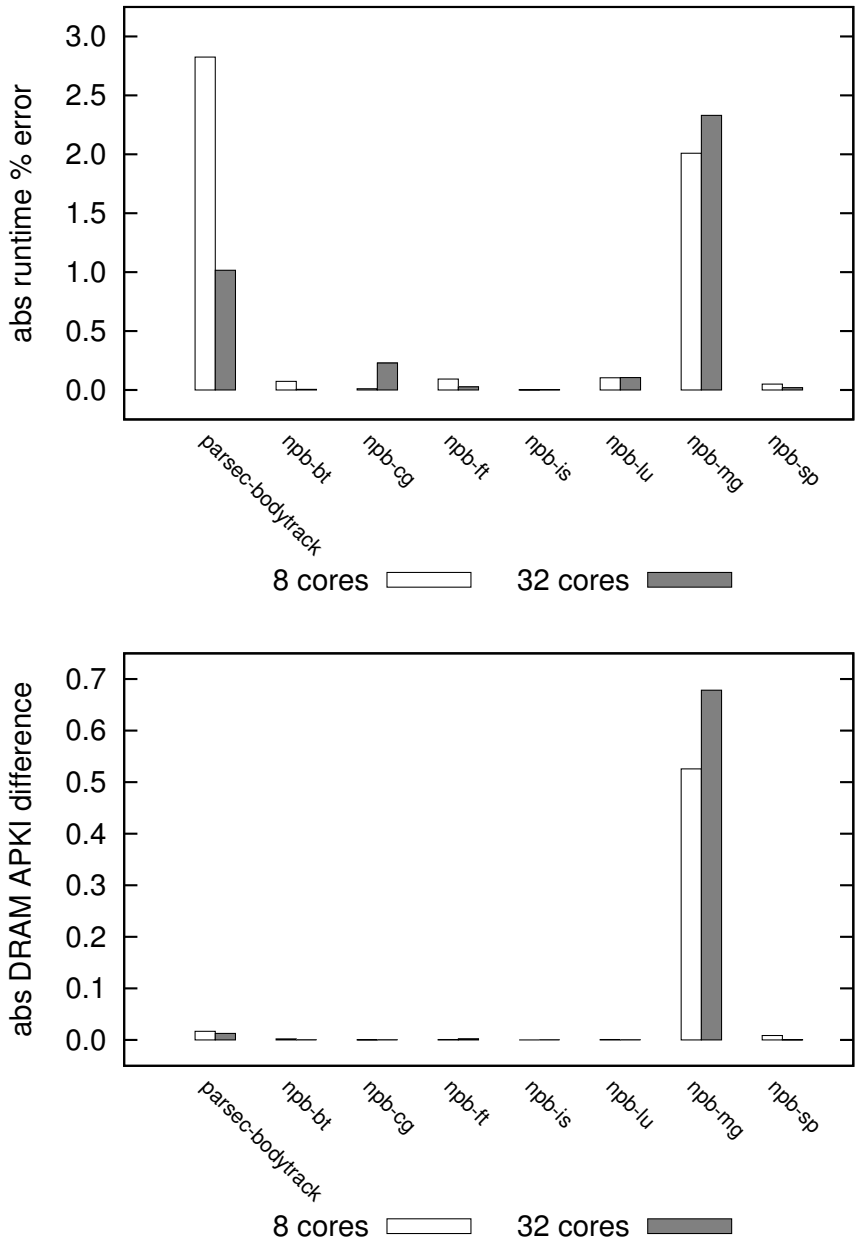


Figure 5.4: Percent absolute error for predicting application execution time (top) and absolute DRAM APKI difference (bottom) across all benchmarks using the BarrierPoint methodology, assuming perfect warmup.

an MSI directory cache coherency protocol. See Table 5.1 for the main characteristics of the simulated machines.

The benchmark suites used in this chapter are the NAS Parallel Bench-

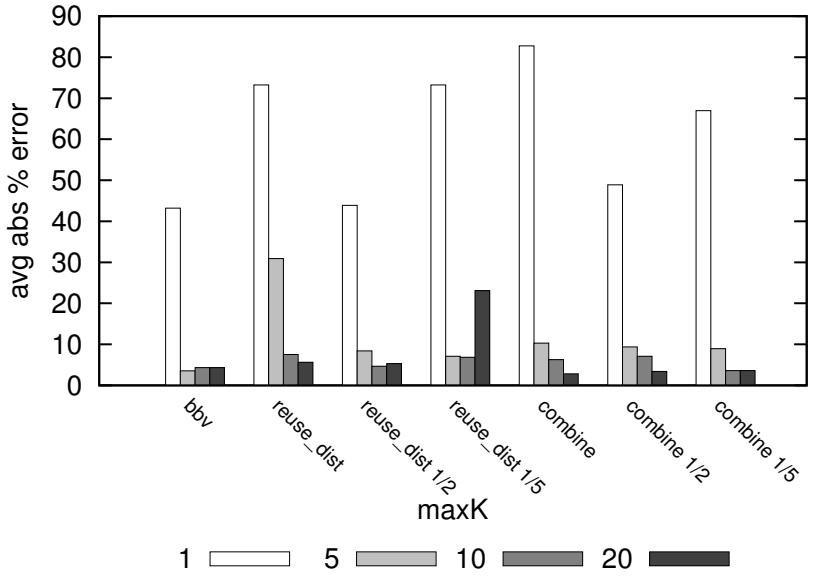


Figure 5.5: Average absolute error for application execution time prediction for different maxK and clustering methods. This data is averaged across 8 and 32-core runs assuming perfect warmup. The LDV vectors are weighted equally ($1/v = 1/1$) or according to the value indicated.

marks (NPB) version 3.3 with OpenMP parallelization (*class A* inputs) [41], and the PARSEC 2.1 benchmark suite (*simlarge* inputs) [7]. Of the 10 NAS Parallel Benchmarks, three were not used. We are unable to run the `npb-dc` (data cube) benchmark in our simulator because it produces a lot of output data that needs to be written to a hard drive; Sniper is a user-level simulator and does not model HDDs, for which reason it cannot accurately run the `npb-dc` benchmark. The `npb-ua` (unstructured adaptive mesh) benchmark generates a very large number of barriers which makes it difficult to analyze. Our current BarrierPoint implementation cannot handle that many inter-barrier regions. There is no fundamental reason to believe that BarrierPoint cannot be applied to this benchmark, however, it might need an extension to filter or combine regions before processing by the BarrierPoint methodology; we leave this for future work. Finally, `npb-ep` is an embarrassingly parallel benchmark, that only contains a single region between barriers. This type of workload does not apply to the BarrierPoint methodology. From Parsec, we use the `parsec-bodytrack` benchmark, as it is one of the three barrier-synchronized benchmarks that uses the OpenMP infrastructure. The other two benchmarks use pthread-based barrier synchronization. The current BarrierPoint implementation works with OpenMP-parallelized applications only, however, this is not a fundamental limitation to the methodology, and it should therefore be applicable

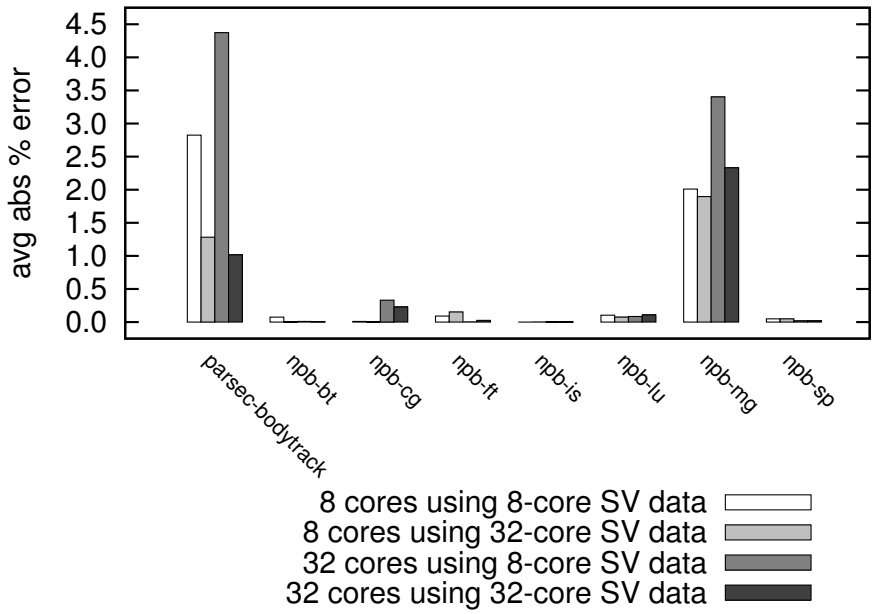


Figure 5.6: Barrierpoint selection cross-validation. Results for 8 and 32-core runs are interchangeable, and therefore we can use the same regions as representative regions.

for pthread barrier-synchronized applications as well.

Only the parallel Region of Interest (ROI) of each application is included in our timing measurements. We do not consider the serial fractions of these benchmarks; sampled simulation of sequential code running on individual cores has been studied extensively in prior work and is considered mature.

Other methodological settings are as follows. We use the passive OpenMP wait policy for thread synchronization, which specifies that waiting threads do not consume CPU resources. All benchmarks were compiled with GCC 4.3 for x86-64 with SSE and SSE2 extensions enabled. BBV and LRU stack distance profiles of inter-barrier regions are collected using a custom Pintool [48]. We use the SimPoint clustering software version 3.2 for identifying representative inter-barrier regions. Non-default parameters used for SimPoint are listed in Table 5.2.

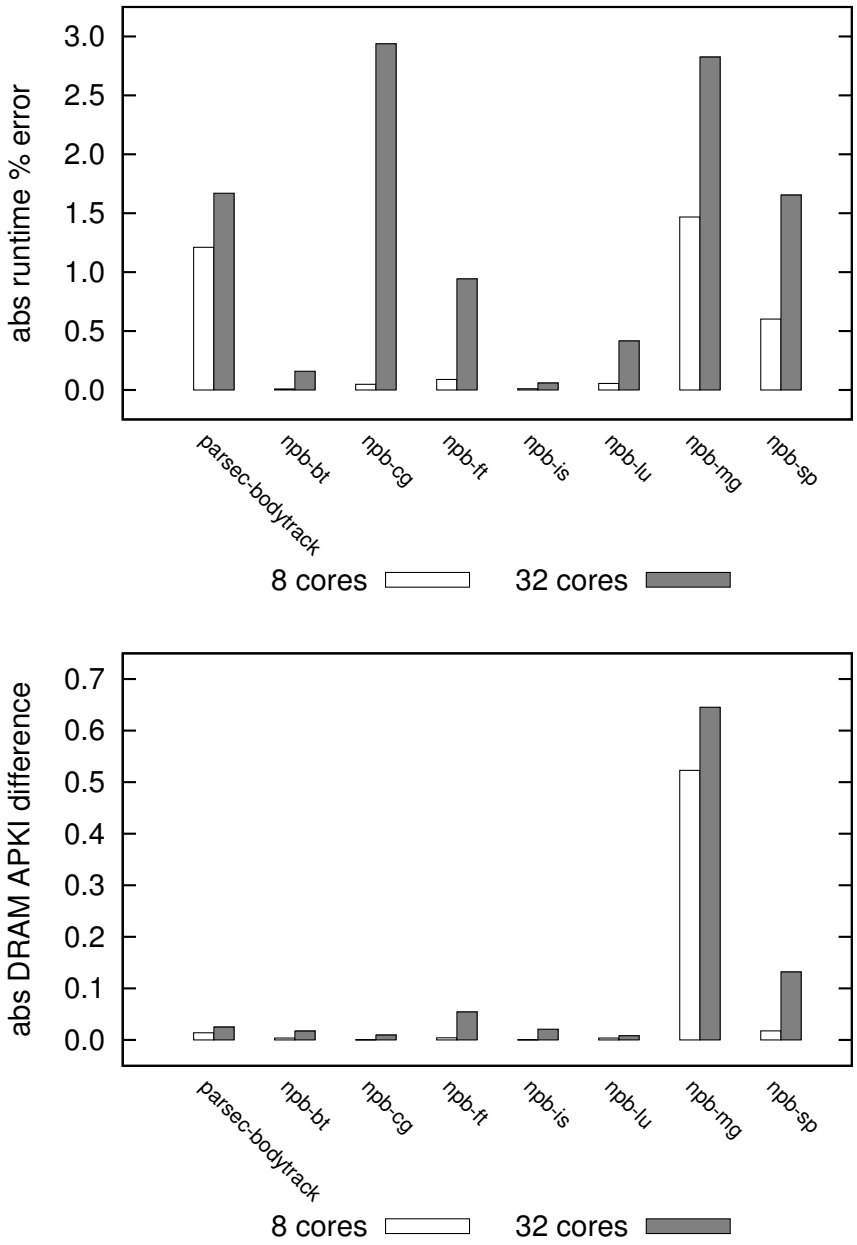


Figure 5.7: Percent absolute error for predicting application execution time (left) and absolute DRAM APKI difference (right) across all benchmarks using the BarrierPoint methodology, with unique warmup.

5.6 Results

5.6.1 Barrierpoint selection

We start by evaluating the barrierpoints selection first. We do this by assuming perfect warmup in order to isolate the error due to barrierpoint

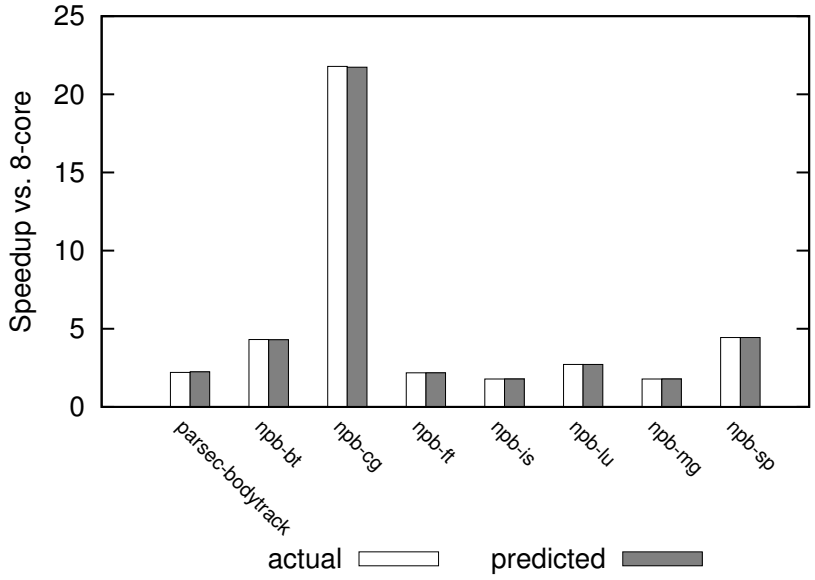


Figure 5.8: Relative scaling results: estimating 8-core versus 32-core speedup.

selection. (We will evaluate the error due to warmup in Section 5.6.2.) This is done by running the simulation of the entire benchmark once and by recording performance metrics on an inter-barrier region granularity. We determine the barrierpoints as described before using microarchitecture-independent signatures, and pick the performance metrics for the barrierpoints from the full benchmark simulations to reconstruct total application execution time. Because all of the inter-barrier regions have a perfectly warmed up microarchitecture state, this approach thus evaluates barrierpoints selection in isolation. Comparing the estimated application execution time against the measured execution time for the full benchmark execution yields a metric for the accuracy of the barrierpoints selection.

Figure 5.4 quantifies the error of the BarrierPoint methodology assuming perfect warmup for (top graph) estimating total application execution time, and (bottom graph) LLC miss rate, or the number of memory accesses per thousand instructions (APKI). Accuracy is high for both metrics with an average absolute error of 0.6% (max error of 2.8%) for predicting application execution time, and an average absolute error of 0.1% (max error of 0.6%) for predicting the number of memory accesses per kilo instructions. Without barrierpoint scaling (where inter-barrier code regions are similar, but instruction counts differ; see Section 5.3.4 for details), the average error increases significantly from 0.6% to 19.4%. Overall, these results demonstrate the high accuracy of the barrierpoints selection strategy as well as the reconstruction mechanism for estimating total application execution time

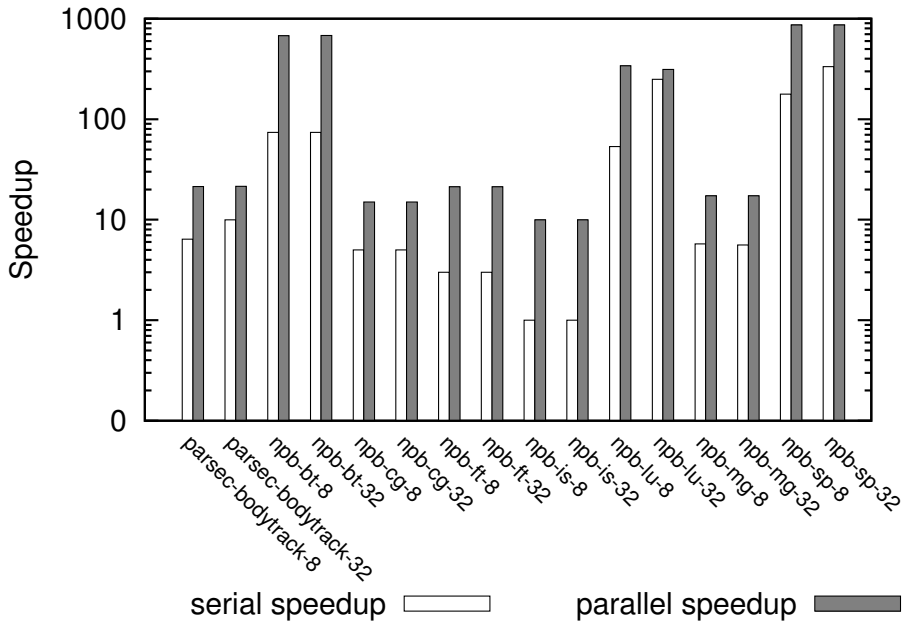


Figure 5.9: Achieved speedups for each benchmark with the BarrierPoint methodology. The serial speedup results from back-to-back execution of barrierpoints and represents the reduction of required simulation resources. The parallel speedup results from a parallel simulation of barrierpoints and shows the simulation latency reduction assuming sufficient server resources.

and performance metrics from a select number of barrierpoints.

Similarity and clustering metrics

We now explore the effect of a number of BarrierPoint parameters on accuracy. Figure 5.5 quantifies average error rates for predicting application execution time for different similarity methods and clustering parameters. We evaluate the impact of the maximum number of barrierpoints selected (maxK) on overall accuracy. We also consider signature vectors consisting of BBVs only, LDVs only, and combined BBV-LDVs. In addition, we also consider weighted LDVs, as previously described in Section 5.3.1, where $1/v$ is $1/1$, $1/2$ and $1/5$ as indicated in the figure. There are several interesting observations to be made here. First, a single barrierpoint yields poor accuracy but accuracy generally improves with an increasing number of barrierpoints. This makes intuitive sense as more regions are being simulated in detail and used to predict overall performance. It also illustrates the widely varying execution characteristics of inter-barriers regions in these workloads. Second, combined signature vectors that characterize both code and data memory access behavior yield the highest possible ac-

curacy, especially with larger numbers of barrierpoints. Weighted LDVs improve accuracy only slightly when used in combined signature vectors; hence, we consider unweighted LDVs ($1/v = 1$) in our default setting. The highest accuracy is achieved with combined signature vectors and a maxK of 20, which is the default setting used throughout the chapter unless mentioned otherwise.

Barrierpoints

A key feature of the BarrierPoint methodology is to provide an easy to use model for sampled simulation. The output of the methodology is a number of select, representative barrierpoints used for detailed simulation along with their multipliers which enables estimating total application execution time. Table 5.3 lists the significant barrierpoints for each of the benchmarks used in this study. The summary details for insignificant barrierpoints are defined as barrierpoints with a contribution of less than 0.1%. Across the benchmarks used, the number of selected barrierpoints is quite small, ranging between 2 and 16, and two to three orders of magnitude smaller than the total number of dynamically executed barriers. Note that because of instruction scaling as described in Section 5.3.4, inter-barrier regions can be larger or smaller than similar ones, meaning that the multipliers do not necessarily sum to the total number of regions.

Barrierpoints across architectures

In Figure 5.6 we present the core cross-validation results. Here we can see that results from an 8-core BarrierPoint run produce similar accuracy numbers compared to the results from the 32-core BarrierPoint similarity generation. This demonstrates that for the OpenMP barrier runtime, the unit of work remains the same across core counts.

Note that the BarrierPoint methodology requires taking a default thread (or core) count to collect the statistics that serve as input to the analysis. This is why Table 5.3 lists different barrierpoints for different core counts. However, barrierpoints can be reliably used across core counts (as long as the number of executed barriers does not depend on thread count). This is quantified in Figure 5.6 which shows accuracy results for using the barrierpoints determined with 8 threads on a 32-core system, and, vice versa, barrierpoints determined with 32 threads on an 8-core system. The key conclusion from this graph is that barrierpoints can be transferred across processor architectures, and hence form well-defined, fixed units of work that can be reliably used to compare processor architectures.

5.6.2 Warmup

In our evaluation up to this point, we have assumed perfect state warmup, which is an idealized situation. Typically when using BarrierPoint, one will need to warm up the microarchitectural state prior to detailed simulation of a barrierpoint. Figure 5.7 quantifies prediction error for application execution time (left graph) and the number of memory accesses per instruction (right graph). In spite of the simplicity of the proposed warmup technique, we find it to be quite accurate for determining application performance metrics. The execution time prediction error increases only slightly to 0.9% on average and 2.9% at most. The error due to incorrect warming is higher for 32 cores than for 8 cores (compare Figure 5.7 to Figure 5.4 which assumed perfect warmup): this is due to the larger total cache capacity (32MB versus 8MB) and performance prediction being more sensitive to accurate warmup for larger caches.

5.6.3 Relative accuracy

Up to this point, we have been concerned with estimating performance in a single processor architecture design point. However, architects are often more interested in predicting relative performance between two design points. Figure 5.8 quantifies the accuracy of BarrierPoint for predicting the relative performance difference between two design points, namely the 8-core versus the 32-core system. BarrierPoint's accuracy is very high for relative performance trends. Three of the benchmarks exhibit superlinear speedups, with `npb-cg` as the most notable example, the reason being cache effects (32MB LLC in the 32-core system versus 8MB in the 8-core system).

5.6.4 Simulation speedup

Figure 5.9 quantifies the simulation speedups for the BarrierPoint methodology across different core counts for the NPB and Parsec benchmark suites. Speedups are defined as the reduction in aggregate instruction count. The serial speedup is the reduction in required resources, and can be thought of as the speedup when running each barrierpoint back-to-back in serial. The parallel speedup results when all barrierpoints are run in parallel with sufficient machine resources. Using the BarrierPoint methodology, our results show a (harmonic) mean parallel speedup of 24.7× and a maximum parallel speedup of 866.6×. Along with application performance improvements, we see an average reduction of 78× in the number of machine resources required for simulation.

5.7 Related Work

5.7.1 Single-Threaded Sampling

The SimPoint [61] methodology clusters large intervals, on the order of 100M instructions, using BBVs and machine learning (cluster analysis) to identify representative chunks of an application in a microarchitecture-independent way. The SMARTS [71] methodology and Conte et al. [20] construct a sample consisting of a large number of sampling units of a relatively small number of instructions per sampling unit. These approaches are unable to accurately estimate run-time of synchronized multi-threaded applications.

5.7.2 Multi-Threaded Sampling

Ekman et al. [24] propose matched-pair comparison as a way to reduce sample size for multi-threaded workloads, but show that synchronized applications do not see a significant sample size reduction with their technique.

Wenisch et al. [69] propose Flex Points as a way to increase simulator performance for multi-threaded commercial workloads. Van Biesbrouck et al. [66] propose the Co-Phase Matrix as a reduction technique for multi-program workloads. Both of these techniques depend on the fact that each thread is independent. Explicit thread synchronization violates their assumptions.

Perelman et al. [55] extend the SimPoint methodology to parallel workloads via instruction-based sampling. Recent work [3; 12] has shown however that instruction-based sampling is inaccurate for multi-threaded workloads that employ active or idle waiting due to synchronization. Furthermore, application IPC is an inappropriate metric for multi-threaded workloads [2]. BarrierPoint, instead, predicts total application execution time, and uses both code and data memory signatures to identify representative barrierpoints.

Time-based sampling of multi-threaded applications, proposed by Ardestani et al. [3], Carlson et al. [12] and described in Chapter 4, allows for accurate sampled simulation of synchronizing multi-threaded applications. By extrapolating time during fast-forwarding phases, and taking care to keep thread interactions through synchronization and shared memory intact, execution time can be accurately predicted. These approaches suffer from two major limitations, which we overcome with BarrierPoint. First, it requires functional simulation of the entire program execution and in addition requires the memory hierarchy to be warmed in between sampling units. In contrast, BarrierPoint can leverage checkpointing to allow

barrierpoints to be simulated independently and in parallel, and does not require functional simulation and cache warming of the complete benchmark execution. Second, time-based sampling can lead to the creation of different samples across processor architectures which may complicate performance analysis. BarrierPoint overcomes this limitation by proposing well-defined, fixed units of work in a microarchitecture-independent way.

5.7.3 Simulation parallelism

Simulation latency can be improved by exploiting parallelism in the units of work that need to be simulated. For example, both SimPoint and SMARTS have proposed checkpointing techniques to simulate each sampling unit independently and in parallel [65; 68]. Bryan et al. [9] demonstrates the potential speedup when executing multiple inter-barrier regions in parallel, provided that massive simulation resources are available. Our work takes this work one step further by reducing the number of inter-barrier regions to be simulated in detail, which dramatically reduces the number of simulation machines needed. This improves overall time-to-discovery while being able to accurately predict total application execution time from the selected of barrierpoints.

5.7.4 Warmup

The cold-start problem is a well-known problem in sampled simulation. No-State-Loss (NSL) [21] and Live-points [68] propose the playback of unique addresses in the memory hierarchy to warm up cache state prior to each sampling unit, and apply it to single-threaded workloads running on single-core systems. MRRL [34] uses the distribution of reuse distances to determine how far to go back prior to the sampling unit to warm up caches. We extend these methods to support multi-threaded workloads and multi-core cache hierarchies by replaying an amount of data equal to the largest last-level cache visible to each core.

Van Biesbrouck et al. [65] propose a warmup method, memory hierarchy state (MHS), that uses a snapshot of the largest cache to be simulated, and reduces the state for the target simulated cache. This technique requires explicit cache state reconstruction in the simulator, is limited to single-core systems, and does not provide a way to reconstruct coherency state.

Barr et al. [6] propose a method for reconstructing cache and coherency state in multi-processor systems. They therefore propose a data structure, called the Memory Timestamp Record (MTR), that records the timestamp of the last access to each cache block before a sampling unit. The MTR allows the simulator to reconstruct coherency state as well as a cache hier-

archy of arbitrary size and associativity, assuming a lower bound on cache line size.

Our warmup methodology offers an alternative to MTR where coherency state and cache hierarchy is reconstructed without detailed knowledge of the cache hierarchy's coherence and multi-level state. The only information needed is the largest total shared LLC capacity that will be simulated in any system configuration.

5.7.5 Similarity analysis

Perelman et al.'s [55] similarity analysis extends Sherwood et al.'s work [61] to multi-threaded workloads. Both of these works evaluate agglomerated/combined thread views into fixed-length intervals but disregard its use as they compare threads to one another during an execution run. On the other hand we compare the entire multi-threaded application's behavior between barriers, not each thread to one another. We are the first to:

- Show that combined thread views can be used to compare inter-barrier regions to perform workload reduction.
- Use LRU stack distances (along with BBVs) to more accurately compare regions.
- Enable the use of variable-length multi-threaded execution profiles which can occur between barriers. Perelman et al. [55] assume fixed-length intervals that are not compatible with variable-length barrierpoints.

Sherwood et al. [61] use BBVs to identify regions of similar execution behavior, called phases, in long-running single-threaded applications; Perelman et al. [55] extend this method to multi-threaded workloads. Shen et al. [59] propose LRU stack distances [49] to automatically determine phases of a single-threaded application. In contrast, our methodology uses program semantics, barriers, to delineate phases, and we find the combined usage of BBVs and LRU stack distances to outperform either alone.

Alameldeen et al. [2] suggest that a common unit of work is required, but does not extend this to multi-threaded workloads, nor do they address the reduction of simulation requirements. In contrast, we are the first to provide an automatic workload reduction and performance estimation methodology for barrier-based multi-threaded workloads.

5.8 Conclusion

Sampling is a well-known technique to speed up architectural simulation. Only recently have researchers extended sampled simulation towards multi-threaded workloads. Some prior work assumed non-synchronizing multi-threaded workloads for which random sampling allows for an accurate representation of the overall application. Time-based sampling proposes a solution for synchronizing multi-threaded workloads but the main limiting factor for achieving significant simulation speedups is the requirement for using functional warming to maintain an accurate micro-architectural state in-between sampling units. In addition, time-based sampling leads to different sampling units across different processor architectures, complicating performance analysis. Prior work in speeding up the simulation of barrier-synchronized applications requires massive simulation resources to simulate all inter-barrier regions in parallel.

To address these limitations, we propose BarrierPoint, a methodology for the reconstruction of application execution time using the similarity of multi-threaded benchmarks between barriers. With BarrierPoint, it is now possible to evaluate the performance of each selected inter-barrier region independently, leading to a higher potential for simulation speedup. Our proposed methodology automatically identifies a select number of most representative regions, called barrierpoints, from which it is possible to estimate total application execution time. Using a set of barrier-synchronized parallel benchmarks from the NPB and Parsec benchmark suites, we demonstrate that high simulation speedups that can be achieved (with a harmonic mean of 24.7$\times$ and up to 866.6$\times$) while using a limited number of simulation machine resources, and while being within 0.9% on average and at most 2.9% compared to detailed simulation. Overall, we reduce the amount of machine resources needed by an average of 78$\times$.

Chapter 6

Conclusion

In this chapter we summarize the conclusions that can be drawn from this dissertation.

6.1 Overview

Processor complexity, as described by Moore's Law, allows processor designers to use an increasing amount of transistors to provide faster, more efficient designs for next-generation microprocessors. Nevertheless, as designs become more complex, the amount of time it takes to verify these designs also increases. But because of current processor power and technology limits, these performance gains are being implemented in the form of multi-core processors. As simulators today tend to be single-threaded, they are facing a losing battle where the performance of the processor itself becomes more and more difficult to simulate over time, widening the simulation gap.

There are two primary solutions to handle this growing simulation gap. One way is to improve the performance of a simulator. By speeding up the simulator, we can now reduce the time it takes to simulate an entire application. In addition, an orthogonal approach deals not with the simulator, but the application itself. Using workload sampling, it is possible to simulate a small amount of an application in detail that provides a representative view into the performance of an entire application. By combining both of these techniques, one can accurately estimate application performance in a shorter amount of time.

The main goal of this research is to allow architects and researchers to use these tools to help in the exploration of new microarchitectural enhancements. We hope that through the use of these methodologies, experiments that occur over long timescales with large input sets can now

become tractable to explore and optimize, allowing for optimizations that could not previously be evaluated in a reasonable time frame.

In the following sections, we detail a number of the key findings from this work that dead with both of these solutions to simulation.

6.1.1 Simulator Speedup with High-Level Core Models

We developed Sniper Multi-Core Simulator, as a way to speed up microarchitectural investigation and analysis. Two key components of the Sniper Multi-Core Simulator are the interval simulation model, and parallel simulation. Interval simulation is a high-level simulation techniques that allow for faster simulation speeds compared to traditional simulation methodologies, while still providing the accuracy required to perform microarchitectural experiments. Interval simulation, therefore, allows for a higher-level of abstraction than traditional detailed cycle-level core models, while providing significantly more runtime accuracy than the one-IPC core model which does not take into account application ILP or MLP. Through the use of parallel simulation and the interval simulation technique, we show an average absolute error within 25% of real hardware with simulation speeds of up to 2.0 MIPS.

In addition, to interval simulation, we detail a new core model, the instruction-window centric core model, which improves accuracy significantly with respect to hardware, to an average error of 11.1%. The instruction-window centric core model, while more detailed, still provides faster simulation than a typical detailed core model while resulting in significantly better accuracy than the simpler one-IPC core model. The strength of the instruction-window centric model is that is allows one to model more detailed timing interactions that occur in an out-of-order core, while still taking advantages of the insights provided by interval simulation to enable fast simulation.

6.1.2 Workload Reduction through Multi-Threaded Sampling

Workload sampling reduces the amount of work the simulator needs to perform in order to estimate key metrics of an application, such as runtime. Single-threaded sampling has been mature, and only recently have we seen methodologies emerge to handle sampling of multi-threaded applications with very specific qualities. Multi-threaded application sampling has the ability to provide significant reductions in the amount of detailed simulation that needs to be performed for microarchitectural investigations. To close the gap for generic multi-threaded applications, we propose time-based multi-threaded sampling to reduce the amount of an application needs to be run in detail to as little as 10% while achieving an average abso-

lute error of 3.5%. By taking into account application-specific behavior, BarrierPoint uses multi-threaded application inter-barrier regions to classify and reduce the amount of an application needs to be simulated. Through the use of this methodology, we have seen performance improvements by a harmonic mean of 24.7× and up to 866.6×, with an average absolute error of 0.9% and 2.9% at most.

6.2 Future Work

While multi-threaded sampling provides a good approach that allows for general-purpose application reduction for simulation, the overall speedups are limited (to around 10× in our examples) in this methodology because of the requirement to keep the memory subsystem active throughout the entire execution of the application. BarrierPoint has the potential to provide even larger speedups because simulation time is now related to the number of representative regions instead of total instruction count, which can reduce simulation time significantly. The best example of this can be seen when comparing the resulting speedups from the two sampling methodologies. The difference in performance between the NPB `sp` benchmark which sees a simulation speedup of about 2.5× with general purpose multi-threaded sampling to 866.6× with BarrierPoint, a sampling methodology targeted to barrier-based applications is significant, around a 300× reduction. This improvement in simulation performance results from being able to take into account application-specific behavior. Through the use of other application-specific sampling methodologies, it might be possible to optimize other classes of multi-threaded applications to expose large reductions in the amount of an application that is needed to be simulated in detail.

A broader direction for future work with respect to multi-threaded sampling could be to expand into the direction where full-system simulation details are taken into account. For example, the effects of external IO (network traffic, or HDD/SDD accesses) might introduce some added complexity into the sampling methodology and require historic analysis for periodicity in the case of general-purpose time-based sampling. Additionally, taking into account true operating system effects might also require additional analysis. Synchronization with relaxed synchronization that takes place during multi-threaded workload execution, even without sampling, might pose less of a problem for accurate IO simulation because of the much larger time scales that off-chip communications tend to deal with. That said, for IO-heavy benchmarks or phases, it could be possible to build an interval-style simulation methodology that uses on-and-off phases of an IO workload to approximate runtime in a fast way.

One additional direction for future work with multi-threaded sampling could be to allow an application's periodicity to be determined at runtime. This would allow one to use a generic set of sampling parameters to start, but as the periodicity of the application is determined, to adjust the sampling periods of the application under evaluation. Some challenges with this approach are the ability to both detect the application's periodicities reliability as well as doing so with a low overhead. Additionally, one would need to determine, with confidence, whether an application's periodicity requires that a new run be started with the newly selected parameters. This could also lead to a problem where an application would need to be simulated multiple times. Finally, the additional overhead in determining periodicity might erase much of the performance benefits seen by using this technique.

Appendix A

Additional Research

This appendix provides additional references to notable research work completed during my Ph.D. studies

A.1 Hardware/Software Co-Design

Stringent performance targets and power constraints push designers towards building specialized workload-optimized systems across a broad spectrum of the computing arena, including supercomputing applications as exemplified by the IBM BlueGene and Intel MIC architectures. In this work, we make the case for hardware/software co-design during early design stages of processors for scientific computing applications. Considering an important scientific kernel, namely stencil computation, we demonstrate that performance and energy-efficiency can be improved by a factor of 1.66× and 1.25×, respectively, by co-optimizing hardware and software.

To enable hardware/software co-design in early stages of the design cycle, we propose a novel simulation infrastructure by combining high-abstraction performance simulation using Sniper with power modeling using McPAT and custom DRAM power models. Sniper/McPAT is fast — simulation speed is around 2 MIPS on an 8-core host machine — because it uses analytical modeling to abstract away core performance during multi-core simulation. We demonstrate Sniper/McPAT's accuracy through validation against real hardware; we report average performance and power prediction errors of 22.1% and 8.3%, respectively, for a set of SPEComp benchmarks.

> W. Heirman, S. Sarkar, T. E. Carlson, I. Hur, and L. Eeckhout. Power-aware multi-core simulation for early design stage hardware/software co-optimization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 3–12, September 2012

A.2 Workload Analysis

We propose a methodology for analyzing parallel performance by building cycle stacks. A cycle stack quantifies where the cycles have gone, and provides hints towards optimization opportunities. We make the case that this is particularly interesting for analyzing parallel performance: understanding how cycle components scale with increasing core counts and/or input data set sizes leads to insight with respect to scaling bottlenecks due to synchronization, load imbalance, poor memory performance, etc.

We present several case studies illustrating the use of cycle stacks. As a subsequent step, we further extend the methodology to analyze sets of parallel workloads using statistical data analysis, and perform a workload characterization to understand behavioral differences across benchmark suites. We analyze the SPLASH-2, PARSEC and Rodinia benchmark suites and conclude that the three benchmark suites cover similar areas in the workload space. However, scaling behavior of these benchmarks towards larger input sets and/or higher core counts is highly dependent on the benchmark, the way in which the inputs have been scaled, and on the machine configuration.

> W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 38–49, November 2011

The performance of data-intensive applications, when running on modern multi- and many-core processors, is largely determined by their memory access behavior. Its most important contributors are the frequency and latency of off-chip accesses and the extent to which long-latency memory accesses can be overlapped with useful computation or with each other.

In this work we present two methods to better understand application and microarchitectural interactions. An *epoch profile* is an intuitive way to understand the relationships between three important characteristics: the on-chip cache size, the size of the reorder window of an out-of-order processor, and the frequency of processor stalls caused by long-latency, off-chip requests (epochs). By relating these three quantities one can more easily understand an application's memory reference behavior and thus significantly reduce the design space. While epoch profiles help to provide insight into the behavior of a single application, developing an understanding of a number of applications in the presence of area and core count constraints presents additional challenges. *Epoch-based microarchitectural analysis* is presented as a better way to understand the trade-offs for memory-bound applications in the presence of these physical constraints.

Through epoch profiling and optimization, one can significantly reduce the multidimensional design space for hardware/software optimization through the use of high-level model-driven techniques.

> T. E. Carlson, S. Nilakantan, M. Hempstead, and W. Heirman. Epoch profiles: Microarchitecture application analysis and optimization. *Computer Architecture Letters*, 2014

A.3 Undersubscription

High-performance computing workloads are compute and memory intensive workloads that are designed to scale very well onto large machines. While this software is able to continue to scale on many-core architectures similar to the Xeon Phi, we investigate how the application's working-set directly impacts performance. As an application scales to a number of core counts, the working set can change in different ways, where some application's working sets can grow, others shrink, and still others have more complex behavior. By treating the application's working set as the most important to performance, one can determine how the application will fit into the cache, and also how much performance can be gained. We find that because of the complex relationships between thread count and working set, it is possible to achieve application performance improvements through the undersubscription of an application on a shared-memory node. In addition, we propose a clustered-cache microarchitecture that allows cores, while undersubscribing, to achieve higher performance because they are now able to take advantage of the larger cache sizes. Finally, we propose a run-time system to automatically determine the optimal undersubscription levels in the OpenMP runtime.

> W. Heirman, T. E. Carlson, K. Van Craeynest, I. Hur, A. Jaleel, and L. Eeckhout. Undersubscribed threading on clustered cache architectures. In *International Symposium on High Performance Computer Architecture (HPCA)*, February 2014

Bibliography

[1] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*, pages 7–18, February 2003.

[2] A. R. Alameldeen and D. A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26:8–17, July/August 2006.

[3] E. K. Ardestani and J. Renau. ESESC: A fast multicore simulator using time-based sampling. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 448–459, February 2013.

[4] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: infrastructure for full system simulation. *ACM SIGOPS Operating Systems Review*, 43(1):52–61, January 2009.

[5] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In R. Eigenmann and M. Voss, editors, *OpenMP Shared Memory Parallel Programming*, volume 2104, pages 1–10. July 2001.

[6] K. C. Barr, H. Pan, M. Zhang, and K. Asanovic. Accelerating multiprocessor simulation with a memory timestamp record. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 66–77, March 2005.

[7] 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 (PACT)*, pages 72–81, October 2008.

[8] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell,

M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[9] P. Bryan, J. Poovey, J. Beu, and T. Conte. Accelerating multi-threaded application simulation through barrier-interval time-parallelism. In *2012 IEEE 20th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 117–126, August 2012.

[10] T. E. Carlson, W. Heirman, K. V. Craeynest, and L. Eeckhout. BarrierPoint: Sampled simulation of multi-threaded applications. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, March 2014.

[11] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 52:1–52:12, November 2011.

[12] T. E. Carlson, W. Heirman, and L. Eeckhout. Sampled simulation of multi-threaded applications. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, April 2013.

[13] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[14] T. E. Carlson, S. Nilakantan, M. Hempstead, and W. Heirman. Epoch profiles: Microarchitecture application analysis and optimization. *Computer Architecture Letters*, 2014.

[15] M. Casas, H. Servat, R. M. Badia, and J. Labarta. Extracting the optimal sampling frequency of applications using spectral analysis. *Concurrency and Computation: Practice and Experience*, 24(3):237–259, March 2011.

[16] J. Chen, L. K. Dabbiru, D. Wong, M. Annavaram, and M. Dubois. Adaptive and speculative slack simulations of CMPs on CMPs. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 523–534, December 2010.

[17] X. E. Chen and T. M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. *ACM Trans. Archit. Code Optim.*, 8(3):10:1–10:28, October 2011.

[18] 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 *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 249–261, December 2007.

[19] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 76–87, June 2004.

[20] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 468–477, October 1996.

[21] T. Conte, M. Hirsch, and W.-M. Hwu. Combining trace sampling with single pass methods for efficient cache simulation. *IEEE Transactions on Computers*, 47(6):714–720, June 1998.

[22] Y. Cui, W. Wu, Y. Wang, X. Guo, Y. Chen, and Y. Shi. A discrete event simulation model for understanding kernel lock thrashing on multicore architectures. In *Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 1–8, December 2010.

[23] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 220–231, September/August 2003.

[24] M. Ekman and P. Stenström. Enhancing multiprocessor architecture simulation speed using matched-pair comparison. In *Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 89–99, March 2005.

[25] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Transactions on Computer Systems (TOCS)*, 27(2):42–53, May 2009.

[26] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–184, October 2006.

[27] A. Fog. Instruction tables. `http://www.agner.org/optimize/instruction_tables.pdf`, April 2011.

[28] A. Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, 2013.

[29] H. Franke, R. Russell, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proceedings of the 2002 Ottawa Linux Summit*, pages 479–495, June 2002.

[30] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[31] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 307–318, February 2010.

[32] A. Glew. MLP yes! ILP no. *ASPLOS Wild and Crazy Idea Session*, October 1998.

[33] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. D. an B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 102–113, June 2004.

[34] J. W. Haskins, Jr. and K. Skadron. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 195–203, March 2003.

[35] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using cycle stacks to understand scaling bottlenecks in multi-threaded workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 38–49, November 2011.

[36] W. Heirman, T. E. Carlson, K. Van Craeynest, I. Hur, A. Jaleel, and L. Eeckhout. Undersubscribed threading on clustered cache architectures. In *International Symposium on High Performance Computer Architecture (HPCA)*, February 2014.

[37] W. Heirman, S. Sarkar, T. E. Carlson, I. Hur, and L. Eeckhout. Power-aware multi-core simulation for early design stage hardware/software co-optimization. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 3–12, September 2012.

[38] T. Huffmire and T. Sherwood. Wavelet-based phase classification. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 95–104, September 2006.

[39] A. Jaleel, R. S. Cohn, C.-K. Luk, and B. Jacob. CMP$im: A Pin-based on-the-fly multi-core cache simulator. In *Proceedings of the Fourth Annual Workshop on Modeling, Benchmarking and Simulation (MoBS), co-located with ISCA 2008*, pages 28–36, June 2008.

[40] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PACT)*, pages 208–219. October 2008.

[41] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS Parallel Benchmarks and its performance. Technical report, NASA Ames Research Center, October 1999.

[42] T. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, pages 338–349, June 2004.

[43] J. Lau, E. Perelman, G. Hamerly, T. Sherwood, and B. Calder. Motivation for variable length intervals and hierarchical phase behavior. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 135–146, March 2005.

[44] J. Lau, J. Sampson, E. Perelman, G. Hamerly, and B. Calder. The strong correlation between code signatures and performance. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 236–247, March 2005.

[45] B. Lee, J. Collins, H. Wang, and D. Brooks. CPR: Composable performance regression for scalable multiprocessor models. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 270–281, November 2008.

[46] T. Li, A. Lebeck, and D. Sorin. Spin detection hardware for improved management of multithreaded systems. *IEEE Transactions on Parallel and Distributed Systems*, 17(6):508–521, June 2006.

[47] J. D. Little. A proof for the queuing formula: L=λw. *Operations research*, 9(3):383–387, May/June 1961.

[48] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[49] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[50] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 145–156, October 1994.

[51] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1–12, January 2010.

[52] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*, pages 254–265, February 2006.

[53] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie. PinPlay: a framework for deterministic replay and reproducible analysis of parallel programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–11, April 2010.

[54] 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 International Symposium on High Performance Computer Architecture (HPCA)*, pages 406–417, February 2011.

[55] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong. Detecting phases in parallel applications on shared memory architectures. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[56] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[57] F. Ryckbosch, S. Polfliet, and L. Eeckhout. Fast, accurate, and validated full-system software simulation of x86 hardware. *IEEE Micro*, 30(6):46–56, November/December 2010.

[58] D. Sanchez and C. Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 475–486, 2013.

[59] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Sperating Systems (ASPLOS)*, pages 165–176, October 2004.

[60] X. Shen, Y. Zhong, and C. Ding. Predicting locality phases for dynamic memory optimization. *Journal of Parallel and Distributed Computing*, 67(7):783 – 796, July 2007.

[61] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 45–57, October 2002.

[62] D. J. Sorin, V. S. Pai, S. V. Adve, M. K. Vernon, and D. A. Wood. Analytic evaluation of shared-memory systems with ILP processors. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (ISCA)*, pages 380–391, June 1998.

[63] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972.

[64] V. Uzelac and A. Milenkovic. Experiment flows and microbenchmarks for reverse engineering of branch predictor structures. In *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 207–217, April 2009.

[65] M. Van Biesbrouck, B. Calder, and L. Eeckhout. Efficient sampling startup for SimPoint. *IEEE Micro*, 26(4):32–42, July 2006.

[66] M. Van Biesbrouck, T. Sherwood, and B. Calder. A Co-Phase Matrix to guide simultaneous multithreading simulation. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 45–56, September 2004.

[67] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic. RAMP: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57, March 2007.

[68] T. Wenisch, R. Wunderlich, B. Falsafi, and J. Hoe. Simulation sampling with live-points. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 2–12, March 2006.

[69] T. Wenisch, R. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. Hoe. SimFlex: Statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, July/August 2006.

[70] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[71] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *Proceedings of the Annual International Symposium on Computer Architecture (ISCA)*, pages 84–95, June 2003.

[72] M. Yourst. PTLsim: A cycle accurate full system x86-64 microarchitectural simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 23–34. April 2007.

[73] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz. Performance analysis using the mips r10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing (SC)*, pages 16–16. IEEE, November 1996.