UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Single Thread Performance in the Multi-core Era

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Leonard Emerson Porter

Committee in charge:

Professor Dean Tullsen, Chair
Professor Chung-Kuan Cheng
Professor Sadik C. Esener
Professor Steven Swanson
Professor Michael B. Taylor

2011

The dissertation of Leonard Emerson Porter is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2011

DEDICATION

*To Lori.*

# EPIGRAPH

*The liberally educated person is one who is able to resist the easy and preferred answers, not because he is obstinate but because he knows others worthy of consideration.*

—Allan Bloom

TABLE OF CONTENTS

LIST OF FIGURES

# LIST OF TABLES

## ACKNOWLEDGEMENTS

Before beginning, I want to stress that any attempt to acknowledge all the amazing people — family, mentors, friends, and colleagues — who have contributed to my education is foolish. Words can scarcely describe how much I owe to each of you and I lament the serial nature of writing in its inherent ordering. Everyone who has been part of my life and my journey in graduate school has been valuable to me and I can only apologize as my attempt to convey this sentiment undoubtedly falls short.

I firstly want to thank my advisor, Dean Tullsen, for his unparalleled wisdom, integrity, and professionalism. Learning from him has been nothing shy of an honor and privilege. I cannot thank him enough for his unwavering faith in me that persisted even in the worst of times — through failed experiments and unprofitable research directions. I am going to miss our many conversations about research, teaching, family, and sports.

I have been immensely fortunate to have a number of mentors, in addition to Dean, throughout my graduate career: specifically Allan Snavely, Beth Simon, and John Glick. Although I have only recently begun working with Allan, I thank him for his guidance and mentoring in HPC research. Beth has provided me with amazing advice throughout the years. She, too, has shown amazing faith in me, involving me in major teaching projects well before I thought myself capable. I cannot thank her enough for teaching me that we can apply the same standards of research to education that we do to computer science. Lori and I will miss the dinners with you and Chris.

John was the original inspiration for me to attend graduate school in computer science as my advisor at USD. As a testament to his patient guidance, he was there for me when I was trying to pick a major, when I was trying to decide on graduate schools, when I was struggling in graduate school, and when I was trying to decide which faculty position to accept. He's been amazing at providing me with support and opportunities, including the priceless opportunity to teach at USD. I will miss our many lunches at Peabody's.

I also want to thank the additional members of the thesis committee —

Sadik Esener, Chung-Kuan Cheng, Steven Swanson, and Michael Taylor — for reviewing my thesis, overseeing my dissertation, and for spending many of their valuable hours assisting with the quality of this thesis.

My wife, to whom this thesis is dedicated, has provided me with unfathomable patience, love, and support throughout graduate school. I could not have done this without her. For all that she has endured, I fervently believe universities should grant spousal degrees (Ph.D. Survivors, or Ph.D.S. for short). I cannot thank her enough for her reassurances that we could finish, for her uncanny ability to make me laugh, and for all we have learned about life together.

As I become "Dr. Porter," I cannot help but think of this title as still reserved for my father. We lost him far too early and I so wish he could have been here to see me graduate. There is no doubt in my mind that I would never have been in this place in life without him. He taught me to have an insatiable thirst for knowledge and to approach life both rationally and philosophically. I owe him so much for teaching me the value of hard work, the importance of family, and the value of humility. I still marvel at (and aspire to) his unique ability to earn the respect of fellow physicists, soldiers, and small-town carpenters alike. At the end of my life, I will consider myself a success if I am half the scholar, citizen, and father he was.

My mother complemented my father well, and it's easy to see why they had such a happy marriage (and family) together. She is the very definition of nurturing. I have her to thank for life itself, and there has never been a moment in my life where I doubted her faith, love, or willingness to do anything she can to support me. I'd also like to thank her for her support, both emotionally and financially during graduate school.

My big sister is an amazing friend, mentor, and confidant. She has always been there for me and I cannot thank her enough for her support. I still have much to learn from her, whether it be lessons in good leadership, lessons in grammar, or the artistic value of 80s music. I thank her husband, Rick, as well, for his support through the years; he is both an amazing leader and person.

My in-laws, Lori's family, are nothing shy of unique. They are far too

numerous to name, but they have accepted me into the family and have treated me exceptionally well. I am so thankful that they are all part of my life. I want to especially thank Bonnie, Lonnie, and Bob for their role in Lori being the amazing person that she is.

My friends have had a uniquely defining influence on my life. Stemming from high school, I have had the distinct pleasure of such amazing friends as Ivan, Erin, Kyle, Corin, Beth, Mahru, Jen, Eri, Tim, Tom, Anne, Nick, Art, Steph, Kaleb, Aaron, and Shae. Many of them endured my friendship during high school and have surprisingly seemed willing to stick around as my friends even today. I think friendships as strong as ours, that last as long as ours, are rare, and I am truly privileged to have all of them in my life.

From my Navy days, I have to thank Rich, Alex, Barry, Jon, Brian, Martin, Bill, Frank, Stew, and, especially, Bob. Although many of us have lost touch, I will value our bonds as soldiers all of my life.

Lastly, I have to thank my friends at UCSD who have made graduate school not just bearable, but enjoyable. Jeff, thank you for teaching me what it means to be a good software engineer, for setting the example for perseverance through adversity, and for being so patient with me as I familiarized myself with the simulator and as I crawled my bike up Torrey Pines. Jack, you are a great friend. You have and will make tremendous contributions to our field during your career. I wish I had your insight and I will sorely miss our lunches. I thank Brian for commiserating with me and for teaching me so much about his field. I'm going to miss our afternoon coffee trips. I'd also like to thank Hung-wei, Bumyong, and the rest of Dean's lab for our many chats. Lastly, I thank Cynthia and Dan for all the great computer science education research we've worked on together.

To the group: Mikey, Laura, Evan, and Sandie, thank you for all our fun adventures together — Settlers weekends, lunches, marathons, 50ks, triathlons, wine tastings, Rock Band! I hope we will continue to stay in touch even as we all go our separate ways.

In summary, the journey of my life that led to this dissertation is best described as a story of relationships — relationships with family, friends, colleagues,

and mentors. To all of you who have played a role in this journey, you have my eternal appreciation.

Chapter 2 contains material from “Mapping Out a Path from Hardware Transactional Memory to Speculative Multithreading”, by Leo Porter, Bumyong Choi, and Dean M. Tullsen, which appears in *Proceedings of the Eighteenth International Conference on Parallel Architectures and Compilation Techniques (PACT).* The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2009 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

Chapter 3 contains material from “Fast Thread Migration via Cache Working Set Migration”, by Jeffrey A. Brown, Leo Porter, and Dean M. Tullsen, which appears in *Proceedings of the Seventeenth International Symposium on High Performance Computer Architecture.* The dissertation author was the secondary investigator and author of this paper. The material in this chapter is copyright ©2011 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or email `permissions@acm.org`.

Chapter 4 contains material from “Creating Artificial Global History to Improve Branch Prediction Accuracy”, by Leo Porter and Dean M. Tullsen, which appears in *Proceedings of the Twenty-Third International Conference on Supercomputing.* The dissertation author was the primary investigator and author of

this paper. The material in this chapter is copyright ©2009 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or email `permissions@acm.org`.

Chapter 5 contains material from "Exploiting Thread Heterogeneity for Improved Throughput, Energy, and Fairness on Multicore Processors", by Leo Porter, Allan Snavely, Anthony Gamst, and Dean M. Tullsen, which has been submitted for possible publication by the Association for Computing Machinery in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 11)*. The dissertation author was the primary investigator and author of this paper.

## VITA AND PUBLICATIONS

| | |
|---|---|
| 2000 | Bachelor of Arts, Computer Science<br>University of San Diego |
| 2000-2004 | Officer<br>United States Navy |
| 2005-2010 | Teaching Assistant<br>University of California, San Diego |
| 2006-2011 | Research Assistant<br>University of California, San Diego |
| 2006 | Intern<br>Microsoft Research, Redmond |
| 2007 | Master of Science, Computer Science<br>University of California, San Diego |
| 2009 | Adjunct Professor, Computer Science Department<br>University of San Diego |
| 2010 | Summer Graduate Teaching Fellow<br>University of California, San Diego |
| 2011 | Master TA, CSE Department<br>University of California, San Diego |
| 2011 | Doctor of Philosophy, Computer Science<br>University of California, San Diego |

## PUBLICATIONS

Leo Porter, Cynthia Bailey Lee, Beth Simon, Daniel Zingaro, "Peer Instruction: Do Students Really Learn from Peer Discussion in Computing?", *In Seventh International Computing Education Research Workshop (ICER)*, August, 2011.

Leo Porter, Cynthia Bailey Lee, Beth Simon, Quintin Cutts, Daniel Zingaro, "Experience Report: A Multi-classroom Report on the Value of Peer Instruction", *Proceedings of the Sixteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, June 2011.

Jeffrey A. Brown, Leo Porter, Dean M. Tullsen, "Fast Thread Migration via Cache Working Set Prediction", *Proceedings of the Seventeenth International Symposium on High Performance Computer Architecture (HPCA)*, February 2010.

Beth Simon, Paivi Kinnunen, Leo Porter, Dov Zazkis, “Experience Report: CS1 for Majors with Media Computation”, *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, June 2010.

Leo Porter, Bumyong Choi, Dean M. Tullsen, “Mapping Out a Path from Hardware Transactional Memory to Speculative Multithreading”, *Proceedings of the Eighteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2009.

Leo Porter, Dean M. Tullsen, “Creating Artificial Global History to Improve Branch Prediction Accuracy”, *Proceedings of the Twenty-Third International Conference on Supercomputing (ICS)*, June 2009.

Bumyong Choi, Leo Porter, Dean M. Tullsen, “Accurate Branch Prediction for Short Threads”, *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2008.

ABSTRACT OF THE DISSERTATION

**Single Thread Performance in the Multi-core Era**

by

Leonard Emerson Porter

Doctor of Philosophy in Computer Science

University of California, San Diego, 2011

Professor Dean Tullsen, Chair

The era of multi-core processors has begun. These multi-core processors represent a significant shift in processor design. This shift is a change in the design focus from reducing individual program (thread) latency to improving overall workload throughput. For over three decades, programs automatically ran faster on each new generation of processor because of improvements to processor performance. However, in this last decade, many of the techniques for improving processor performance reached their end. As a result, individual core performance has become stagnant, causing diminished performance gains for programs which are single-threaded.

This dissertation focuses on improving single-thread performance on parallel hardware. To that end, I first introduce modifications to a new form of parallel memory hardware, Transactional Memory, which can improve the viability of Speculative Multithreading — a technique for using idle cores to improve single-threaded execution time. These modifications to Transactional Memory improve Speculative Multithreading effectiveness by a factor of three. I further improve the performance of Speculative Multithreading by addressing a primary source of performance loss — the loss of thread state due to frequent thread migrations between cores. By predicting the cache working-set at the point of migration, we can improve overall program performance by nine percent. Recognizing the demand for transistors to be dedicated to shared or parallel resources (more cores, better interconnect, larger shared caches), I next propose a method of improving branch prediction accuracy for smaller branch predictors. I demonstrate that there are regions of program execution where long histories hurt prediction accuracy. I provide effective heuristics for predicting these regions — in some cases enabling comparable accuracies from predictors of half the size. I then address the problem of contention among coscheduled threads for shared multi-core resources. To reduce resource contention, I propose a new technique for thread scheduling on multi-core processors with shared last level caches which improves the overall throughput, energy efficiency, and fairness of the coschedule.

# Chapter 1

# Introduction

The age of multi-core processors has begun. As such, modern processors can execute more threads simultaneously than ever before. Modern processors contain both many individual cores and simultaneous multithreaded cores, i.e., each core can execute multiple threads. For example, Intel's Nehalem processor has 4-8 cores, each of which is capable of executing two threads and Sun's Niagara 3 has 16 cores, each of which can execute 8 threads. With this, the nature of modern processor design now focuses on improving overall system throughput rather than improving a single thread's performance — the latter being the primary focus of processor design until around 2005.

In today's multi-core processors, many thread contexts lead to increased overall throughput when there are sufficient threads available for execution. However, these additional thread contexts do not aid the execution of any single thread. Single threads remain highly relevant as many applications exist as legacy binaries (often single-threaded), many applications are difficult and/or expensive to parallelize, and some applications simply lack fundamental parallelism. In addition, even for those applications which are parallelizable, Amdahl's Law teaches us that as we improve the non-serial portions of these highly parallelizable applications, the serial component will begin to dominate performance [HM08].

Opportunities exist to improve single-thread performance using parallel hardware. Parallel hardware provides a large number of thread contexts and many of these contexts will be idle due to a lack of available threads. We can leverage these idle resources to improve the performance of single-threaded programs.

Improving single-thread performance in the new multi-core landscape demands solutions to a number of central questions:

1. Can we use idle thread contexts to improve the performance (execution time) of a single thread?

2. Can we reallocate transistors from single-thread optimizations to parallel components (more cores, larger shared caches, better interconnect) without sacrificing single-threaded performance?

3. When threads share resources (common in multi-core designs), how can we schedule threads to avoid resource contention?

This dissertation focuses on answering, in part, these questions. It addresses methods for improving single thread performance using the abundant parallel hardware in multi-core processors. These methods include addressing the roadblocks and challenges related to speculatively parallelizing single threads, providing accurate branch predictions with less hardware, and scheduling threads together given limited shared hardware. Each of these points will be elaborated upon in the following sections.

## 1.1 Leveraging Parallel Memory Hardware to Support Speculative Multithreading

Speculative Multithreading (SpMT) is a promising new technique for improving single-thread performance by leveraging idle parallel hardware [SBV95, AD98, HHS$^+$00, KT98, MGQS$^+$08, MGT98, PO05, SR01, SM98, SAHL04]. SpMT aims at splitting a single thread into multiple threads which can be executed in parallel. By dividing the execution into threads that are executed in parallel, performance gains become possible.

Unlike traditional parallelism where dependencies between parallel threads are rare (if ever), speculative parallelism targets the domain where dependencies

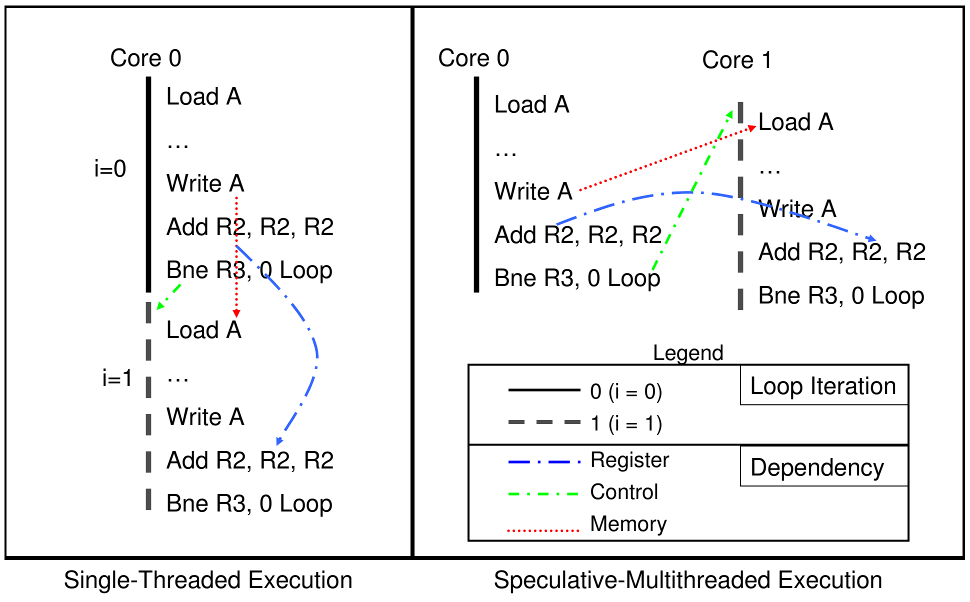


Figure 1.1: Example of speculative loop parallelization using Speculative Multithreading. Dependencies across loop-iterations are shown by the arrows.

are frequent. These frequent dependencies are often either manageable or predictable, and those dependencies which cannot be managed or predicted need to be uncommon.

Loops are a common target of SpMT and provide a useful example (see Figure 1.1). In single-threaded execution, each loop iteration is executed in serial and values are passed from one loop iteration to the next. There are three types of dependencies between loops: control dependencies (branches), memory dependencies (stores and loads), and register dependencies (register producing and register consuming instructions). In SpMT, loop iterations may be selected to be executed in parallel. Unlike serial execution where dependencies are commonly addressed via serial commit, SpMT requires dependencies be handled between threads. Control dependences and register dependencies are highly predictable, but memory dependencies create considerable challenges.

Memory dependencies are commonly addressed in SpMT proposals by adding special-purpose memory hardware which is designed to detect dependencies and support recovery from misspeculation. This hardware is a significant barrier for widespread SpMT adoption due to its cost.

At the same time, Transactional Memory (TM) [HM93, AAK$^+$05, BGH$^+$08, CTTC06, HF03, HWC$^+$04, MCC$^+$05, MHW05, RHL05, ST95, YBM$^+$07] has been recently proposed for multi-core processors to aid in parallel execution. TM provides two key features:

1. *TM provides an easier interface for parallel programming.* It replaces locks with easier to understand transaction semantics (begin transaction and end transaction).

2. *TM provides support for optimistic concurrency.* Unlike locks that serialize execution of critical sections by allowing only one thread to execute at a time to prevent a potential conflict between threads, TM allows all threads to execute a critical section and only aborts if a true conflict exists.

Transactional Memory has been proposed using either hardware (HTM), software (STM), or hybrids of both. In this work we focus on HTM implementations as STM is likely too slow for latency-sensitive speculative threads.

HTM has already appeared in one real processor [TC08] because of its benefits for parallel execution. We recognize that we can leverage this investment in parallel memory hardware to provide SpMT at a fraction of the cost of adding SpMT support to traditional memory designs. In Chapter 2 we evaluate the potential for HTM to provide SpMT and demonstrate that its current support for dependency handling offers limited SpMT performance. We present a number of key additions to HTM which improve the performance of SpMT. In doing so, we propose a new HTM design which is capable of high performance in both transactionally parallel programs [RRW08] and speculatively parallelized single threads [PCT09]. Using two cores, basic register prediction, and averaged over the SPEC CPU2000 benchmarks, SpMT using prior eager-detect HTM designs offers only a 5% improvement in performance, whereas our new HTM design offers a 26% improvement.

## 1.2 Improving SpMT Performance by Reducing Thread Migration Cost

Although our new HTM design from the previous section improved SpMT performance greatly, it continued to suffer performance losses from the migration of execution from one core to another. This is a common problem for any technique which aims to improve performance by migrating threads and is especially problematic for SpMT because of its frequency of thread migrations.

Modern processors learn about the behavior of threads as they execute and, as a result, are capable of executing long threads efficiently. However, when threads are short and/or migrate frequently, this disrupts the ability of the processor to learn about the threads. There are two key hardware components designed to learn thread behavior. The first is branch prediction hardware, which learns the behavior of branches. The second is caches, which learn the behavior of memory.

The first of these challenges, branch prediction, was addressed in our prior work. The second of these challenges, cache behavior, remains a critical problem. In Chapter 3 we propose a number of techniques for improving the memory performance of speculative threads. We demonstrate that a recent proposal, working-set-migration [BPT11], can be used to reduce the average memory delay in SpMT by 50% for select SPEC CPU2000 benchmarks. This reduction in SpMT memory delay translates to a 9% improvement in whole program performance.

## 1.3 Branch Prediction Accuracy with Fewer Resources

The shift in focus to parallel performance over single-thread performance causes architects to reevaluate transistor usage throughout the processor. In reevaluating the transistor usage, the primary question is how much utility does each transistor provide in terms of overall processor performance. For example, by reducing the number of transistors dedicated to non-critical components that provide marginal performance gains, you may be able to better use those saved transistors (and the power associated with them) to build another core, a better core inter-connect, or larger caches.

Deep pipelines place heavy stress on branch prediction as they force hardware to speculate on branch outcomes many cycles before branches are resolved. Misspeculation has the severe consequence of flushing an entire pipeline — in effect losing many cycles worth of work. As a result, branch prediction hardware became a major hardware component in single-core design as pipeline depth increased.

We now have to revisit this design choice from the past, aiming to scale back the hardware cost of branch prediction. The critical question then becomes, can we still provide high branch prediction accuracy, and hence high single-thread performance, using fewer resources. To this end, we examine the value of history for branch prediction accuracy in Chapter 4. Many modern branch predictors use global history to index prediction tables [YP93, CEP96, Kes99, LCM97, McF93, SFKS02, SM99, JL02, EM98, Sez05]. Although history is valuable for many branches (hence the development of history-based predictors), we demonstrate that there are periods during program execution where history breaks down. We propose a simple technique based on heuristics that improves the performance of existing branch predictors and, in some cases, enables the same branch prediction accuracy with half the hardware. This frees transistors (and their associated power) for potential reallocation elsewhere in the processor.

## 1.4 Thread Scheduling on Multi-core Processors

In the previous section, we discussed the challenge of achieving high single thread performance as resources are diverted from components which only benefit a single thread to components which benefit processor-wide performance. Many of these components which benefit processor-wide performance are shared. In this section, we discuss the complementary problem of how to use scheduling to ensure these shared resources are used effectively.

Modern multi-core processors have resources that are dedicated to one core (private) or are shared among multiple cores (shared). Figure 1.2 is an example of

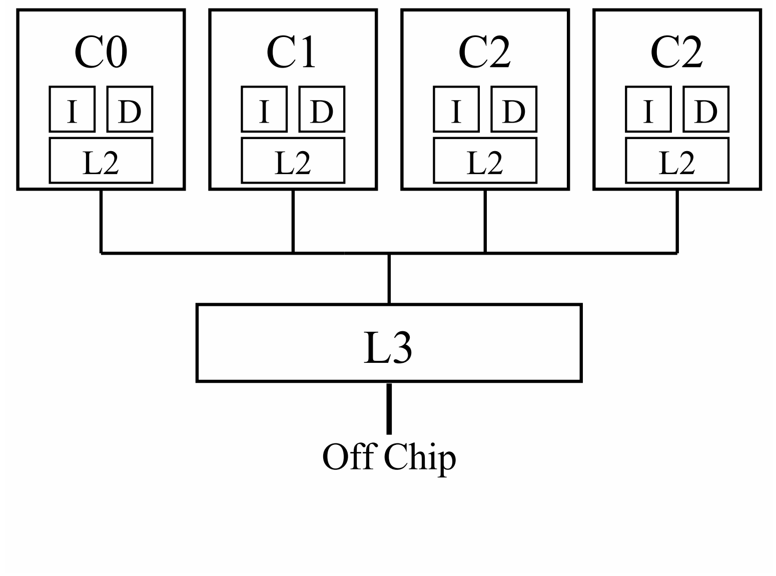


Figure 1.2: Cache layout on a modern (Nehalem) processor.

the cache layout on a recent multi-core processor. In this example, the L1 (I and D) and L2 levels of cache are private to each core, whereas all cores share the L3 and off chip bandwidth.

Shared resources can be a significant source of contention between threads scheduled on different cores of the same processor. This contention can impact overall processor throughput, individual thread performance, overall fairness (do all contending threads suffer equally?), and overall energy efficiency. As such, deciding which threads should be scheduled together (coscheduled) becomes a critical decision for anyone concerned with the single-threaded performance within that coschedule.

In Chapter 5, we recognize that coscheduling threads with similar memory characteristics is almost always the wrong choice as they require the same shared resources and will therefore experience significant contention. We demonstrate that scheduling heterogeneous threads can alleviate this contention and provide superior performance. We then propose a metric which can be used to predict which heterogeneous coschedules offer high throughput (99% of best possible), fairness (48% better than average), and energy efficiency (99% of best possible). These results are consistent on both Nehalem [KDMK08] and Westmere [KBM$^+$10] processors under multiple scheduling assumptions. An additional advantage of our metric is that it can be determined through a single profiling pass and yet provides useful predictions for scheduling on multi-core processors with diverse processor characteristics.

## 1.5 Contributions

This dissertation provides the following contributions:

**Chapter 2**

- We evaluate the potential for Hardware Transactional Memory to support Speculative Multithreading (SpMT) and demonstrate that Hardware Transactional Memory, as currently proposed, offers poor SpMT performance (5% whole program improvement using two cores).
- We propose and examine a number of modifications to Hardware Transactional Memory to determine which are critical for SpMT and which are not. Those we identify to be critical are both inexpensive and offer significantly improved SpMT performance (26% whole program improvement using two cores).
- The value to SpMT of these modifications is shown to be consistent across different numbers of cores, different core architectures, and different register predictor assumptions.

**Chapter 3**

- SpMT introduces additional memory slowdowns due to frequent thread migrations and line invalidations. To address this problem, we evaluate a recent proposal for generic thread migration, Working Set Migration, in the context of SpMT. We examine a number of heuristics for Working Set Migration and, although many of these heuristics are shown to be ineffective for SpMT, we identify one that significantly improves SpMT memory performance. Adding this feature to our memory design from the previous chapter reduces average memory access time by 50% and improves whole program performance by 9%.

**Chapter 4**

- The majority of branch predictors today use global history to inform their predictions. Although many branches are correlated with previous branches, we identify regions of execution where branch correlation is limited, causing history to become essentially noise. For many branch predictors, this noise interferes with both prediction and training, resulting in worse prediction accuracy.

- We propose simple and nearly free techniques for identifying these regions which, when combined with smaller branch predictors, can improve their accuracy. In some cases, a branch predictor modified to use our techniques can offer prediction accuracy comparable with traditional branch predictors of twice the size.

**Chapter 5**

- We demonstrate that the coscheduling of threads which are homogeneous is almost always the wrong choice on multi-core processors with shared cache resources. Coscheduling homogeneous threads can cause resource contention which results in poor overall processor throughput, poor energy efficiency, and poor fairness. Heterogeneous coscheduling can reduce such contention and improve all three of these performance metrics.

- For use in heterogeneous coscheduling decisions, we propose the Cache Hit Rate Vector (CHRV) to classify the memory behavior of a program. The CHRV has the advantage that it can be determined with a single profile pass and yet is still useful for scheduling decisions on multiple processor architectures.

- We propose and examine a number of metrics for use in coscheduling decisions. A novel metric which leverages the CHRV, WSO-Combo, provides the best coscheduling decisions of the metrics examined. By using this metric to inform our scheduler, coschedules are selected which offer 99% of the best possible throughput, 99% of the best possible energy efficiency, and 48% better than average thread fairness.

In whole, these chapters address a number of the key challenges to single-threaded performance in the multi-core era.

# Chapter 2

# Mapping Out a Path from Hardware Transactional Memory to Speculative Multithreading

In the previous chapter, we offered Speculative Multithreading as a potential solution to the question: "Can we use idle thread contexts to improve the performance of a single thread?" In this chapter, we focus on the memory system required for Speculative Multithreading. Specifically, we examine the potential for leveraging Hardware Transactional Memory to provide the memory hardware required by Speculative Multithreading.

Speculative Multithreading (SpMT) is a promising technique for improving single thread performance by leveraging idle parallel resources. However, the cost of handling memory dependences is a significant barrier for wide-spread adoption. In this Chapter, we investigate the potential for using Transactional Memory (TM) for SpMT.

TM has been proposed as a powerful programming primitive for shared memory multiprocessors to replace traditional lock-based synchronization [HM93, AAK+05, BGH+08, CTTC06, HWC+04, MCC+05, MHW05, RHL05, YBM+07, HF03, HLMS03, ST95]. Prior research has examined software transactional memory [ST95, HF03, HLMS03] and hardware transactional memory [HM93, AAK+05, BGH+08, CTTC06, HWC+04, MCC+05, MHW05, RHL05, YBM+07]. TM has gained significant interest in both academia and industry because of the potential to make programming the coming multi-core and many-core processors easier [HCW+04] and more effective [HM93]. These factors led SUN to propose inclusion of hardware support for transactional memory in their Rock processor [TC08]. We expect other implementations to follow.

Speculative Multithreading (SpMT) [SBV95, HHS+00, MGQS+08, PO05, SM98, KT98, SR01, MGT98, AD98, SAHL04] is another architectural alternative that becomes increasingly attractive as we enter the multi-core era more fully. It provides the hope of exploiting available parallel cores to increase the performance of a single thread of execution. Serial code will increasingly be a limiting factor both in terms of power and performance [HM08].

Given the imminent arrival of hardware support for TM, we would like to leverage that hardware to add support for true speculative multithreading, at a lower incremental cost than supporting speculative multithreading from scratch. The focus of this work is to map out a path from HTM to SpMT. What are the elements that we would need to add to a memory design that supported HTM in order to allow SpMT? How should they work in an HTM context? Which of those elements are crucial and which are not?

It has been stated and demonstrated that transactional memory provides support for speculative multithreading, or thread level speculation via its support for opportunistic concurrency [CTTC06, CCM+06, HWC+04, MCC+05, PCC07]. That is, if two iterations of a loop both access shared data within a transaction, but the accesses conflict in only a small number of cases, the iterations will execute in parallel when there is no conflict, and cause transactions to abort and execute in series when the conflict exists. Thus, the traditional domain of TM is code that is truly parallel (dynamically). It does not support the parallel execution of code where frequent and numerous dependences exist. It also requires significant compiler or programmer support to identify and properly exploit opportunities for thread-level speculation.

To perform this study, we create an architecture for speculative multithreading that executes unmodified single-threaded binaries, relies on prediction for register dependence handling, and relies on the memory subsystem to handle memory dependences. We use this SpMT framework in concert with a number of HTM designs. In addition to the designs previously proposed, we examine a number of intermediate steps along the path from HTM to full SpMT support and show the performance potential at each step. We ultimately define a unique architecture which efficiently supports both TM and dependency-rich SpMT.

Assuming this speculative multithreading approach, we evaluate HTMs which feature eager conflict detection (also assuming enough architectural support to be able to execute in this environment) using SPEC CPU2000 binaries and found that a limited speedup of 10% can be obtained on two cores. The primary weaknesses of traditional HTMs in a SpMT environment are their inability to differentiate more speculative threads from less speculative threads, false sharing, cold cache effects whenever a new thread is initiated on a core, the delayed notification of potential dependencies, and the inability to forward available values when loaded by a more speculative thread. By introducing improvements to HTM with eager conflict detection, we are able to achieve more than a 3x increase in the effectiveness of SpMT (resulting in a speedup of 36% on two cores).

The improvements considered along the path from TM to SpMT include word-granularity tracking of memory coherence, support for ordered transactions, forwarding of values to higher ordered transactions, and a write-update cache coherence protocol.

This chapter is organized as follows. Section 2.1 discusses related work. The assumed SpMT architecture is outlined in Section 2.2. The methodology is provided in Section 2.3. In Section 2.4, the baseline transactional memory is discussed. Improvements to hardware TM are detailed in Section 2.5 with associated performance results. Section 2.6 addresses the generality of these results.

## 2.1 Related Work

Our work relates to previous proposals for Speculative Multithreading and Transactional Memory. The work in each area is described in the following sections,

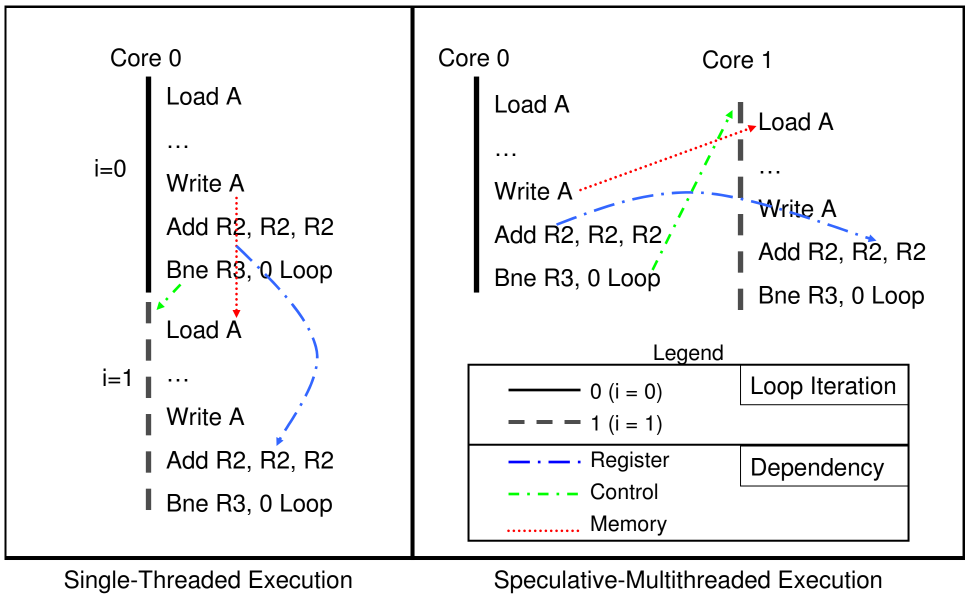


Figure 2.1: Example of speculative loop parallelization using Speculative Multithreading. Dependences across loop-iterations are shown by the arrows.

following a brief background on Speculative Multithreading.

### 2.1.1 Speculative Multithreading Background

Speculative Multithreading was first proposed by Sohi et al. [SBV95] as an architecture tailored for splitting sequential execution into speculatively parallel threads. Revisiting our example from the introduction, Figure 2.1 is an example of the speculative parallelization of a loop. In this example, the first iteration of the loop ($i = 0$) is executed non-speculatively on Core 0. A speculative thread is created to execute the second iteration of the loop ($i = 1$) on Core 1. From this example, we can see that there are a number of requirements for a Speculative Multithreading architecture. These requirements include:

- The identification of an instruction which will trigger the spawn and execution of the speculative thread. This instruction is commonly called the Spawn Point (SP) [MG02]. Which segments of execution are selected for speculative threads depends on the SpMT hardware. In this example, a loop iteration has been selected and the SP would be the first instruction in the loop. Another common target for speculative threads is a function call. For function calls, the non-speculative thread executes the function and the speculative thread begins execution after the function returns. Any two points can result in a speculative thread in the Mitosis architecture [MGQS$^+$08].

- The identification of the point in execution to merge/validate the speculative thread. This is commonly referred to as the Control Quasi-Independent Point (CQIP) and is the first instruction of the speculative thread. For loop iterations, as in this example, the CQIP is the same as the SP and is the first instruction in the loop.

- Register Live-Ins need to be known to identify which register values must be passed from parent thread to child thread or predicted at spawn. The register live-in in our example is register R2 which is consumed by the "Add" instruction in the speculative thread and produced by the "Add" instruction in the non-speculative thread. There are a number of ways to manage register dependences which are discussed in the following section.

- Memory dependences need to be detected and managed. In our example, there is a dependence on address "A" because the parent thread produces the value (Write A) which is subsequently consumed (Load A) by the child thread. Various memory hardware designs (discussed in the following section) manage conflict detection and management differently. The primary issue is that if the speculative thread reads "A" and uses it before the correct value is produced by the non-speculative thread, corrective action is required. This action may require the squashing of the entire speculative thread, or just the flushing of a few instructions [SAHL04].

Each SpMT proposal in the literature manages each of these requirements differently. Given this basic background, we can discuss these proposals in more detail.

### 2.1.2 Speculative Multithreading

As mentioned in the previous section, Speculative Multithreading [SBV95, HHS$^+$00, MGQS$^+$08, PO05, SM98, KT98, SR01, MGT98, AD98, SAHL04] provides the ability to parallelize otherwise serial code. Previous studies in the speculative multithreading literature produce speculative threads either using a compiler or hand tuning [SBV95, HHS$^+$00, MGQS$^+$08, PO05, SM98], statically using binary instrumentation [KT98], or else the speculative threads are identified entirely in hardware [AD98, MGT98, SAHL04]. When most SpMT proposals encounter an inter-thread dependence conflict, the speculative thread is squashed; however, Dynamic Multithreading only squashes, then re-executes, those instructions affected by the dependence [AD98, SAHL04].

Register dependences can be identified statically which enables a number of potential solutions. Since register dependencies can be determined by the compiler, the values can be explicitly forwarded between threads so long as the speculative thread is stalled before the consumption of the value [SBV95]. The stalling of the thread to wait for a register value can be a performance bottleneck which motivated the use of register prediction [MG00]. Another solution to the register dependency problem comes from a recent SpMT proposal, the Mitosis architecture [MGQS$^+$08]. Mitosis uses compiler support to produce a short thread, the precomputation slice, which executes before the start of the speculative thread. The precomputation slice computes register live-ins to avoid explicit forwarding or prediction at the cost of executing additional instructions.

Memory dependences pose a significant challenge for SpMT as they generally cannot be identified until runtime and thus require additional support. A number of memory designs for SpMT have been proposed, starting with the Address Resolution Buffer (ARB) [FS96, SBV95]. The ARB benefits from simple memory address disambiguation, but a single centralized data cache for multiple cores may represent a bottleneck. To address this issue, multiple memories which buffer speculative state in local data caches were proposed to provide improved performance [CMT00, HHS$^+$00, VGSS01]. The Stanford Hydra CMP [HHS$^+$00] is a hybrid which allows private L1 caches to contain speculative contents but maintains all speculative data in dedicated buffers adjacent to the L2. These buffers are maintained consistent with the L1 contents via a write-through L1 store policy. The Speculative Versioning Cache (SVC) [VGSS01] uses cache coherency modifications combined with per-line or per-word pointers to ensure speculative data is buffered, conflicts are detected, and ordering information remains.

Our work shares many of the same goals as memories designed uniquely for SpMT. However, none of these models support TM code, and, of course, do not leverage a potential transactional hardware base.

### 2.1.3 Transactional Memory

Optimistic Concurrency — the execution of multiple threads in parallel with the aim that conflicts are infrequent — is an aim of both SpMT and Transactional Memory. Hardware Transactional Memory (HTM) was proposed by Herlihy and Moss as a simpler means of synchronization capable of performing optimistic concurrency [HM93]. This hardware model has been adopted by more recent TM proposals including LTM [AAK$^+$05], UTM [AAK$^+$05], PTM [CNV$^+$06], LogTM [MBM$^+$06], and VTM [RHL05]. While these proposals vary significantly in their interaction with software transactional memory, ability to handle context-switches and transactional overflow, and support for nested transactions, their conflict detection semantics are similar and can all be represented for this work by a single hardware model (the ULI model, described in Section 2.4.1). These designs use eager conflict detection; they detect conflicts at the time they occur. Other designs, discussed soon, use lazy conflict detection; they buffer conflict information and, as a result, detect conflicts after they occur. LogTM-SE [YBM$^+$07] proposes eager conflict detection based on hashed signatures at block granularity. This reduces the storage requirements on caches to track read-set and write-set information. However, a conflict in LogTM-SE is the same as in our ULI model.

TokenTM [BGH$^+$08] avoids modifying coherence protocols by using tokens to track conflicts. Again, the notion of a conflict in TokenTM is the same as our ULI model. Hybrid Transactional Memory [DFL$^+$06], the likely model of hardware support for SUN's proposed Rock Processor [TC08], also has conflict detection semantics similar to our ULI model.

FlexTM [SDS08] separates conflict detection and conflict management. In some benchmarks, our LAZY design achieves stronger performance than our eager design. For these cases, switching between eager and lazy conflict management could be useful. However, the software overheads for FlexTM may hinder performance.

Dependence Aware TM (DATM) [RRW08] offers some of the features recommended in this work. However, there are key differences. DATM adheres to transactional semantics which offer weaker guarantees than those required by our SpMT architecture. Most notably, stale data read by a transaction can persist in a cache and be read by a non-transaction. In addition, DATM requires bus requests for most transactional requests even in the case of a cache hit whereas our recommended design (OFWI, discussed in Section 2.5.4) does so only in the event of conflicts on that line. Lastly, our recommended design (OFWI) allows previous transactions to overwrite words written by later transactions whereas DATM does not. The impact of this last limitation is discussed in Section 2.5. Most importantly, however, DATM and our conclusions are complementary. The features added by DATM are shown to aid transactional parallelism and the features added by our designs are shown to aid speculative parallelism. The overlap of many of these features emphasizes their importance.

Transactional Coherence and Consistency (TCC) has been proposed as an alternative to traditional per-memory operation coherence [HWC$^+$04, MCC$^+$05]. In TCC, all memory operations are grouped into transactions. The transactional data is buffered at each core during a transaction. When the transaction commits, all transactional data is broadcast on a shared bus ensuring an atomic commit. Other cores observing this broadcast update their cache values similar to a write-update coherence protocol. Conflict detection in TCC is performed when the

Table 2.1: Correspondence of previously proposed HTM designs to the designs proposed later in this chapter.

| HTM Proposal | Conflict Detection | Our Corresponding HTM Design |
|---|---|---|
| LTM, UTM, PTM, LogTM, LogTM-SE, TokenTM, VTM, and Hybrid TM | Eager | ULI |
| TCC and Bulk | Lazy | LAZY |

broadcast occurs (commonly referred to as ”lazy” conflict detection). The conflict detection is performed at word granularity by broadcasting word addresses and the new data for that word address. Similar to TCC, Bulk [CTTC06] also performs lazy conflict detection at word granularity with write update but does so using hashed signatures rather than cache line bits to reduce the amount of data broadcast at commit time. These proposals are represented by our LAZY model (Section 2.5.8).

Our HTM designs represent a number of existing proposals. Again, many of these proposals differ in details largely orthogonal to their potential to support SpMT. A summary of the existing designs, their manner of conflict-detection, and our corresponding design appears in Table 2.1.

The ability of Transactional Memory to support Speculative Multithreading is mentioned in TCC [HWC$^+$04, MCC$^+$05] and Bulk [CTTC06]. Transaction sizes in the context of speculative parallelism are addressed in [CCM$^+$06]. Recent work extends programming language loop constructs to provide support for speculative multithreading [PCC07]. Bulk also uses HTM for compiler-based SpMT [CTTC06]. However, none of these mechanisms provide full support for the parallel execution of dependent threads.

## 2.2 SpMT Execution Model

This section describes our SpMT execution model and the minimum architectural support needed by any of the memory designs to be able to execute in our SpMT architecture. The focus of the rest of this chapter is the design of the memory architecture itself; however, this section primarily details other aspects of the architecture necessary to fully support speculative multithreading, and what assumptions were made about that architectural support.

### 2.2.1 Speculative Thread Selection

Prior proposals have assumed varying degrees of compiler support [SBV95, AD98, HHS$^+$00, KT98, MGQS$^+$08, MGT98, PO05, SR01, SAHL04, SM98]. We fully believe that compiler support will maximize the potential performance of speculative multithreading. However, for this study we assume no compiler or software support to identify speculative threads. Assuming no support provides a general SpMT model that allows us to examine our different memory designs in a fashion unbiased by the compiler, as we are not able to structure the code (even unintentionally) to favor one model. Additionally, this approach significantly expands the domain of SpMT — enabling parallel execution of legacy uniprocessor code, allowing the use of a single binary for a family of architectures that support different levels of SpMT, etc.

Our strict insistence on no compiler support causes our results to be potentially pessimistic compared to a system that allows compiler involvement. Even simple compiler support on single-thread binaries that moves or removes problem memory operations could greatly increase available SpMT parallelism. The bottom line, though, is that achieving the highest possible SpMT speedups is not the goal of this research. Rather, we are trying to understand the support in the transactional memory subsystem necessary to achieve the full potential of SpMT. The absolute magnitude of the speedups achieved is less significant.

Following the terminology of [MG02], a speculative thread is characterized by two PC addresses: the point in execution where a new thread will be created, called the Spawning Point (SP), and the point in the program where the new thread will begin executing, called the control quasi-independent point (CQIP). The new thread ends when it reaches another thread's CQIP and validates that dependences between the two threads were handled correctly. Compared to transactional execution, the entire thread essentially becomes a single transaction, and either completes execution atomically or is squashed/aborted in its entirety.

Speculative threads have been initiated at loop iterations, loop continuations, and function calls in the past [MG02]. The Mitosis architecture [MGQS$^+$08] can construct a SP-CQIP pair from any two arbitrary points in the program. However, without compiler support, our options are more limited. We target loop iterations and function call continuations, as they can be easily identified at run time and have been shown to be good candidates [MG02]. Out-of-order spawns are possible when a less speculative thread encounters a spawnpoint.

When an executing thread fetches an instruction whose address is a spawn point, we determine if an idle core is available. If one is available, we transmit any program state needed by the new thread (e.g., register live-ins) and start the speculative thread on that core after a communication and fetch delay.

When the parent thread encounters the CQIP of its child it stops fetching and waits for the CQIP to be committed. When the CQIP is committed, the child thread is validated. Validation includes the checking of any register live-ins and ensuring that the speculative thread was not squashed due to a memory dependence conflict. Speculative threads validated by non-speculative threads become non-speculative. Speculative threads wait to commit until they become non-speculative

A confidence counter is maintained to indicate the frequency at which a spawn point results in a committed or squashed speculative thread. Spawn points which frequently squash are ignored after reaching a number of consecutive failures but are still given a small chance of re-execution. We found that allowing nine consecutive failures and giving a five percent chance for retry of failed spawnpoints was successful at reducing interference from frequently squashing threads.

To maintain a desirable speculative thread length, threads greater than ten instructions and less than ten thousand instructions are targeted. This can be accomplished by recording the length of previously executed threads.

Threads may be squashed due to dependence issues (discussed below), a system call by the non-speculative thread, or when the non-speculative thread has transactional state overflow. A squash of any thread causes all more speculative threads to be squashed. Speculative threads may be paused (caused to stop execution until made non-speculative) if they encounter a system call or cause a transaction overflow.

A store miss request can be issued at execute, preemptively loading the cache line, or at commit. By requesting the line during execute, the request latency is partially hidden rather than delaying the store entirely to commit. All memory designs assume this realistic optimization. This introduces the possibility of squashes due to wrong-path write miss requests for all of our memory designs except those which handle conflict detection at thread commit[CTTC06, HWC$^+$04, MCC$^+$05].

Our simulator assumes a centralized structure for speculative thread coordination which we refer to as the Global Speculative Thread Supervisor (GSTS). Important thread information (thread ordering, CQIP of children, etc.) is distributed to each core to avoid unnecessary communication with the GSTS. In a general TM system, parallel transactions can typically commit in any order that does not produce conflicts. Because we must maintain sequential semantics, we must commit threads in execution order. Support for this already exists in some TM proposals [CTTC06, HWC$^+$04, MCC$^+$05]. An overall notion of thread ordering allows for committing of threads in order, a necessary feature for SpMT. This is different than having the memory coherence know and account for that order which is addressed in a later section. For example, if coherence were to be made aware of thread ordering, more speculative threads may be able to safely write to addresses read by less speculative threads. This is a useful distinction in thread ordering, both because the overall thread ordering is necessary and coherence-aware thread ordering is an optimization, and also because the former is easily supported in the GSTS, and the latter requires coherence modification.

### 2.2.2 Handling Register Dependences

Because we execute an unmodified single-threaded binary, the code, although executed in parallel, assumes a single unified register file. As a result, there will be register dependences that cross thread boundaries. Prior work has managed these dependences in various ways including explicit forwarding [SBV95, Vij98, ZCSM02], hardware prediction [MG02], and software precomputation [MGQS$^+$08].

Table 2.2: Architectual Specification

| | | | |
|---|---|---|---|
| Cores | 2 | I cache miss penalty | 20 cyc |
| Fetch width/core | 4 | D cache | 32k, 4 way |
| INT instruction queue | 64 entries | D cache miss penalty | 20 cyc |
| FP instruction queue | 64 entries | shared L2 cache | 2 MB, 8 way |
| Reorder Buffer entries | 128 | L2 miss penalty | 75 cyc |
| FP registers per core | 132 | L3 | 4 MB, 8 way |
| Fork penalty | 10 cyc | L3 miss penalty | 315 cyc |
| Cache line size | 64 bytes | Victim cache entries | 8 |
| I cache | 32k, 4 way | | |

In this research we initially want to decouple the register dependence problem (which is orthogonal to the memory design) and thus adopt a very simple model for our initial results — register dependences are handled by prediction and all predictions are correct. We consider the case of a realistic, but not particularly aggressive, predictor in Section 2.6 and show that all the key conclusions of the chapter still hold. We should note that the assumption of perfect or near-perfect prediction is not an absurd one — the Mitosis register prediction model [MGQS+08], because it leverages code from the spawning thread, can be made to be arbitrarily accurate. However, this causes a larger delay in thread spawn.

## 2.3 Methodology

To evaluate the performance of our various models of speculative multithreading, we added support for SpMT, including the full suite of memory designs, to the SMTSIM simulator [Tul96]. The SMTSIM simulator has extensive support for both multithreaded and multi-core execution, and for this study it is configured for multi-core. Speculative threads are executed on available cores on a CMP with a varying number of in-order execution cores. Table 2.2 gives the configuration details of the default architecture we simulate.

The simulator models spawning at instruction fetch, exposes wrong-path memory operations to coherence, and squashes threads in reaction to various memory coherence operations. It also models the effect of spawn points fetched on the wrong path (a result of the design decision to spawn threads when the spawn point is fetched rather than at commit) by ensuring that the target core is occupied and unavailable for execution until the branch mispredict is discovered.

We use the SPEC CPU2000 benchmarks, specifically the Alpha binaries for SimpleScalar [ALE02]. Using reference inputs, we simulate the execution of one hundred million instructions starting at the standard SimPoint [SPHC02].

## 2.4 Leveraging TM for SpMT

The two main requirements of any SpMT memory design are:

1. *Speculative State Buffering*–speculative state must be invisible to less speculative threads as well as protected from reaching lower levels of the memory hierarchy.

2. *Conflict Detection*–the hardware must be able to detect if a speculative thread has used an incorrect memory value or a hazard is present.

Hardware transactional memory supports each of these requirements. Speculative state is tagged as transactional in a transactional buffer or the cache, depending on the implementation. It is possible, in some implementations, for the same value to exist in multiple transactional caches, with different values. Memory data conflict detection is also supported and relies on modifications to the cache coherence protocol.

For SpMT, these features of TM can be leveraged to detect memory dependencies (and hazards) between threads. By starting a transaction when spawning a speculative thread, the TM hardware will protect all memory operations. Should a memory dependence be caught, the TM hardware can squash the offending speculative thread and discard its speculative state. Which memory dependences cause a speculative thread to be squashed depends on the TM implementation.

### 2.4.1 Baseline Transactional Memory Design

Our baseline HTM design is based on the original proposal by Herlihy and Moss [HM93] which has been adopted by more recent TM proposals including LTM [AAK+05], UTM [AAK+05], PTM [CNV+06], LogTM [MBM+06], and VTM [RHL05]. This design has the same conflict detection semantics as those in LogTM-SE [YBM+07], TokenTM [BGH+08], and HybridTM [DFL+06]. However, like most recent studies, we buffer speculative state in cache, whereas the original work used a special Transactional Buffer. In our implementation, a transactional bit is added to each cache line to differentiate those cache lines affected by the current transaction. A transactional read bit is added to identify lines read by the current transaction. A memory conflict occurs whenever the write set of one thread and the read or write set of another thread overlap. When a conflict occurs, the TM arbiter (which handles the restarting of threads) consults the thread ordering and squashes the more speculative thread. This TM design uses a write-invalidation protocol and any transactional, dirty cache lines are invalidated when squashed. We refer to this design as the Unordered Line Invalidate (ULI) design, to distinguish it from other designs we will introduce.

The baseline TM design buffers speculative state in the L1 data cache, with an eight-entry dedicated victim cache for transactional state overflow. In the rare event that the victim cache becomes full with transactional data for a speculative thread, we cause the thread to wait rather than invoking software. Likewise, if the non-speculative thread overflows its victim cache, we squash all speculative threads. Although squashing is not required, this is not a common case in our studies, so the baseline HTM (without ordering) approach of squashing is used in that event.

Conflict detection is handled eagerly per memory operation by modifying the coherence protocol. Non-transactional coherence follows a traditional MESI protocol [PP84]. TCC [HWC+04] and Bulk [CTTC06] perform lazy conflict detection on a per thread basis, incorporating burst transfers of write set information. Lazy conflict detection will be addressed in Section 2.5.8.

In this chapter, we assume a system that has the ability to execute either

SpMT code or standard parallel TM code. Thus, any optimization on the path from TM to SpMT that adds significant overhead but only provides benefit for SpMT execution would be questionable. Similarly, an optimization that improves SpMT performance at the expense of normal parallel performance would also be undesirable.

It is likely that a standard TM architecture would have support for transactions larger than supported by the base hardware TM mechanisms [AAK$^+$05, BGH$^+$08, CNV$^+$06, MBM$^+$06, RHL05]. This is critical because standard TM code will fail to make forward progress if a transaction is too large. However, in our SpMT architecture, we can simply abort a transaction/thread that overflows the buffer space (the victim cache in this case). This is the solution we simulate, because fall-back support for larger transactions would likely be too slow for a SpMT solution. Thus, which of the proposed large transaction solutions might be supported by the actual system is unimportant for this work.

### 2.4.2 Baseline TM SpMT Performance

Although the conventional TM architecture described in this section ensures correctness by catching inter-thread dependences, it does not provide encouraging performance results. Using two cores and averaged over the SPEC CPU2000 benchmarks, we achieve a 1.10 speedup over baseline execution. (Results per benchmark are provided in Figure 2.2.)

## 2.5 Accelerating SpMT

The previous section revealed that the SpMT performance of an existing HTM design with eager conflict detection is limited. The focus of the remainder of this chapter is identifying the shortcomings of that design and evaluating a set of hardware additions that eventually enable full support for speculative multithreaded execution. The key issues we address are support for ordered transactions, forwarding between transactions, addressing false sharing, and cold-cache effects.

This approach allows us to map out a path from HTM to SpMT that will allow us to fully exploit the hardware support for HTM we expect to see on future processors, while allowing the more aggressive speculative parallelization enabled by speculative multithreading.

### 2.5.1 Ordered Transactions

Of the various HTM designs, TCC [HWC$^+$04] and Bulk [CTTC06] support ordering between transactions [HWC$^+$04]. In these designs, conflict detection is handled at transaction commit and commits can be forced to occur in an explicit order. The performance of these designs is in Section 2.5.8. However, allowing for ordering between transactions in an eager conflict detection HTM, where detection occurs per memory operation, requires a very different implementation than TCC or Bulk.

We extend the baseline (ULI) to create an Ordered Line Invalidate (OLI) design. In this design, the coherence protocol is aware of the thread ordering and uses that knowledge to avoid unnecessary squashes. Then, for OLI, a memory conflict consists of any write-set overlap between threads as well as overlap between the read set of a more speculative thread and the write set of a less speculative thread. Overlap between the read set of a less speculative thread and a write set of a more speculative thread does not cause a squash. But this requires special handling of that line in the less speculative thread, because once the thread commits, the line is no longer valid within the execution context of a future thread that might occupy this core. The solution we employ is the same as that proposed by SVC [VGSS01]. We mark the less speculative line as *stale*. By stale we mean that the data is valid during this transaction but must be discarded when the transaction is completed. A stale line can be thought of as a delayed invalidate. Thus, a *stale* bit is added to all cache lines. A stale cache line is written back (if necessary) and invalidated when a thread commits. As with the ULI design, if a conflict occurs the more speculative thread is squashed and its transactional, dirty cache lines are invalidated.

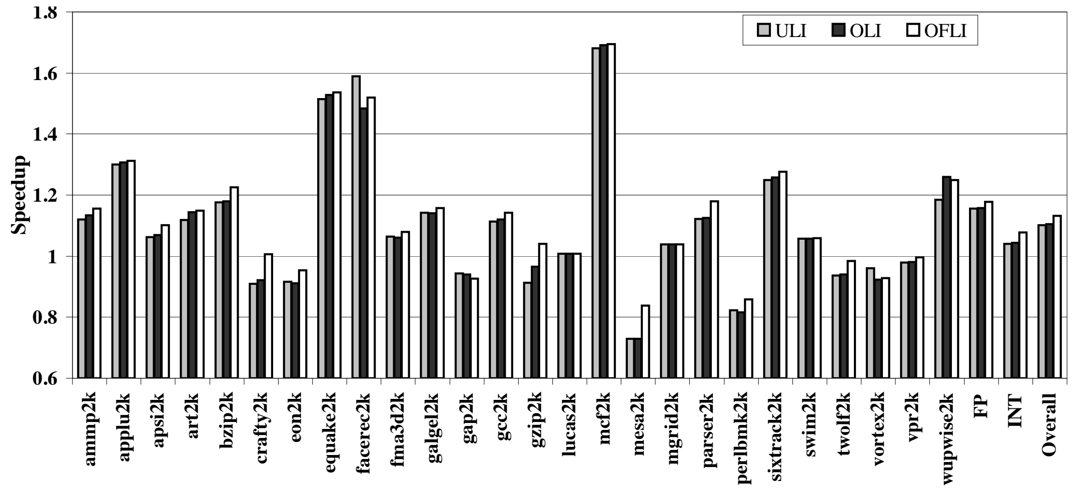


Figure 2.2: Speedup per benchmark for ULI (unordered line invalidate), OLI (ordered line invalidate), and OFLI (ordered forwarding line invalidate) designs, relative to sequential execution.

### 2.5.2 Data Forwarding for Ordered Transactions

We can further exploit the thread ordering information now that we have introduced it into our coherence mechanisms. A logical extension to the OLI design recognizes that if a more speculative thread requests a cache line held dirty by a less speculative thread, this should not result in a squash. Instead, the cache line can simply be forwarded between cores. Since the more speculative thread can only be committed if all less speculative threads are committed, this operation does not impact correctness. To address this scenario, we propose the Ordered Forwarding Line Invalidate (OFLI) design. By allowing for forwarding, we no longer need to squash when a more speculative thread reads a line held dirty by a less speculative thread. In addition to the information tracked in the OLI design, this new design requires a *forwarded* bit. The *forwarded* bit recognizes that a cache line is clean but has been forwarded from another core. Lines marked as forwarded need to be invalidated if this thread is squashed.

A summary of coherence actions that result in a squash can be found in Table 2.3 for the coherence designs discussed so far — ULI, OLI, and OFLI.

Table 2.3: Squashes caused by coherence actions for three memory designs. The more speculative thread is squashed in each case. The table indicates what happens when a single cache line is accessed by one core while the line is held by another core. LS = Less speculative. MS = More Speculative. R = Read. W = Write.

| Action | Holder | ULI | OLI | OFLI |
|---|---|---|---|---|
| R | R | | | |
| W | W | X | X | X |
| LSW | MSR | X | X | X |
| MSR | LSW | X | X | |
| MSW | LSR | X | | |
| LSR | MSW | X | | |

### 2.5.3 Effectiveness of Line Granularity TM for SpMT

The speedup over the baseline (single thread execution) per benchmark for our first three memory designs is shown in Figure 2.2. In the majority of our benchmarks, the OFLI design outperforms the other designs.

Our intuition that adding ordering and forwarding between threads will result in more speculative threads committed is validated in that we see an average of 144k, 154k, and 156k threads committed in ULI, OLI, and OFLI respectively. Unfortunately, these improvements did not result in a significant performance gain (about 3% over ULI). In addition, our results are being offset by an overall loss in memory performance caused by cold cache effects. Using these memory designs for SpMT, the average memory access time has increased by 20% over single-threaded execution. As we successfully commit threads, the stream of committed execution flows from core to core, and we become vulnerable to cold caches. Section 2.5.8 and Chapter 3 examine this phenomenon.

Also of note is that a number of the benchmarks suffer a slowdown. The majority of this slowdown can be attributed to SpMT overheads and memory slowdowns. For these benchmarks, a less aggressive spawn policy would be beneficial.

Averaged across all benchmarks, these results show that neither ordered transactions nor data forwarding have significantly improved the viability of speculative multithreading.

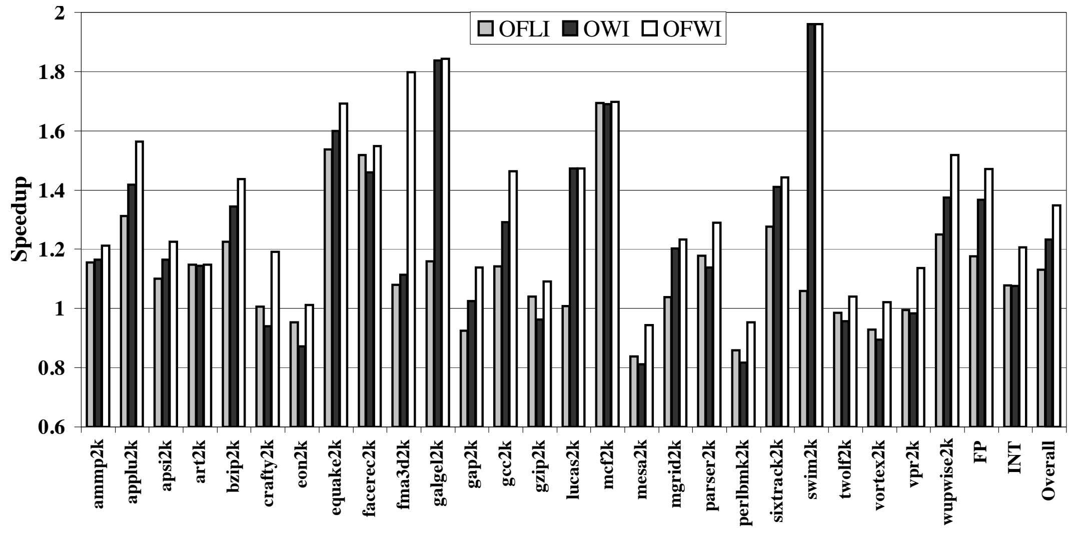


Figure 2.3: Speedup per benchmark for OFLI (ordered forwarding line invalidate), OWI (ordered word invalidate), and OFWI (ordered forwarding word invalidate) designs, relative to sequential execution.

### 2.5.4 Eliminating False Sharing

Prior SpMT work has shown that word-granularity speculative state information is necessary for strong performance [CMT00, VGSS01]. Our previously described HTM designs maintain state information at a cache-line granularity. This keeps coherence cost low, but results in a number of unnecessary squashes. In fact, for our OLI design, 72% of all squashes were due to false sharing. This number would be even higher, except that after threads repeatedly fail to commit due to false dependences, they stop being spawned.

There are two reasons tracking coherence at line granularity can cause excessive squashes. The first is that a read by a more speculative thread followed by a write of a different word in the same cache line by a less speculative thread need not result in a squash. However, false sharing would force a squash in our line-granularity designs. The second reason is the inability to reconstruct lines correctly in the presence of write sharing. When two threads write to the same cache line, a squash is required in earlier designs simply because it is impossible, without word-granularity tracking, to determine which word(s) from each dirty line should be preserved in the correct version. Each of these problems can be partially addressed by keeping a read and write bit per word per cache line. Writes of less than a word continue to be problematic, but they are infrequent in most programs — we deal with byte-granularity writes in Section 2.5.6.

Previous SpMT solutions, if they address this issue, require significant additional coherence traffic to resolve word-granularity versioning issues. Instead, we introduce a new solution by adding a *safe* bit to each line to eliminate unnecessary false squashes. When a more speculative thread has read a word, and a less speculative thread later writes a different word in the same line, our previous designs signal a squash. Instead we mark the line as *unsafe* in the more speculative thread's cache. That thread can still read data it previously read or wrote, but accesses to any other word in an *unsafe* line require a coherence operation to get the latest version of the line. This is somewhat conservative, because we keep only a single *safe* bit, rather than one per word. Words written in unsafe lines are written back and invalidated when the thread is committed. On a squash an unsafe line is invalidated.

Without a safe (unsafe) bit to mark that only a portion of a line is valid, the coherence mechanism would either have to force the thread to squash (when we mark it unsafe) or immediately generate coherence traffic to collect all previously untouched words (which could be dispersed in various caches). The latter solution would likely cause prohibitive bus congestion. We evaluated the former solution and found that 55% of committed threads would have been squashed if it were not for the safe bit.

We add two new memory designs, then, which both support coherence tracking at the word level. They are the Ordered Word Invalidate (OWI) design and the Ordered Forwarding Word Invalidate (OFWI) design. They are analogous to the OLI and OFLI designs, respectively, but with word-level invalidation and a *safe* bit. In the OFWI design, we still require only a single *forwarded* bit per line.

### 2.5.5 Word Granularity Results

The results for the word-granularity coherence tracking designs are shown in Figure 2.3. These results demonstrate that the ability to track coherence status at a word granularity has a significant impact on SpMT performance. For the OFWI design, for example, we now experience gains of 47% for floating point, 35% overall, and a maximum speedup of 96%. Also of interest is that word-level tracking suddenly makes data forwarding much more important. We see this because the difference between OFWI and OWI is much larger than the difference between OFLI and OLI seen previously. In fact, for SPECint, word-level tracking alone provides no performance gain, but in combination with forwarding the gains are significant.

The hardware overhead of word granularity is not insignificant, but it is clearly an important enabler of SpMT. But the advantage will not be limited to SpMT. Other parallel code using standard transactional memory will also benefit, in some cases likely significantly, from the elimination of transactional aborts due to false sharing [RRW08]. Not only does eliminating false sharing in a TM system improve performance, but also insulates the programmer from yet another complexity of traditional parallel programming (false sharing).

### 2.5.6 Byte Granularity Results

All proposed word-granularity designs address byte granularity writes by maintaining a single bit per cache line to signify that a sub-word granularity write occurred to the line. When these writes make it impossible to reconstruct the word, the more speculative thread is squashed. The impact of byte hazard squashes is minor. We see 1% and 5% of thread squashes on average to be caused by byte hazards for OWI and OFWI, respectively. But the performance loss is virtually none. This would indicate that those threads that are lost due to byte hazards were typically either going to be squashed anyway, or were not going to be high quality speculative threads.

### 2.5.7 Word Granularity for Unordered Transactions

One goal of this research is to identify a unified architecture for optimistic concurrency which effectively supports both TM and SpMT. Standard TM code will not always make use of all of the features we have proposed. The differences will be seen only when the code makes use of unordered transactions (which may be common in TM code).

In order to execute unordered transactions (transactions with equal ordering) written for traditional TM systems, minor modifications to the protocol above are required. For transactions of equal order, any cache line transactionally modified by another thread is made unsafe. Reads to previously untouched words in unsafe lines are filled by less speculative threads or by the L2 rather than by forwarding between equally ordered threads. Any overlap of the word read-set and the word write-set between equally ordered transactions results in an abort.

### 2.5.8 Write Update

Despite the performance improvements resulting from adding word granularity information to cache lines, memory performance remains a limiting factor. As the memory designs improve their support for completion of speculative threads, coherence misses and memory delay increases. The coherence misses increase from 39% of data cache misses for OFLI to 47% for OFWI. The slowdown in average memory delay over baseline execution also increases from 1.18 for OFLI to 1.40 for OFWI. There is a cold cache effect, and these numbers show that a significant factor is coherence misses.

With a write-invalidate cache coherence protocol, writes cause line invalidations which cause coherence misses. For example, on a four-core SpMT processor, a frequently-read variable will be in every cache. Each write to that variable will potentially result in three eventual coherence misses.

Our next design seeks to address the latter of these concerns by implementing a write-update protocol. Few cache-coherent multiprocessors have incorporated the write-update protocol [McC85], despite significant early research. This is because few data items are write-shared by many threads of a typical parallel application at once. However, with SpMT, especially in the absence of compiler support, this characteristic is not necessarily preserved. Thus, we investigate the utility of the write-update protocol in this architecture.

Before outlining the addition of the write-update protocol to our OFWI design, we will discuss our LAZY design which represents TCC [HWC+04] and Bulk [CTTC06]. The discussion regarding this design has been delayed because TCC and Bulk both have word-granularity conflict detection and a modified write-update protocol and are hence better compared against an eager-detection HTM with similar features.

## LAZY

Two HTM proposals support lazy conflict detection — no information about modified lines are visible outside the core until a transaction commits. Both of these HTM proposals support word granularity. These proposals are represented by the design *LAZY* in our results. The first, TCC [HWC+04, MCC+05], supports word-granularity coherence and the write-update protocol. TCC proposes replacing per-memory operation coherence with the bursting of write sets at commit. All execution remains completely isolated during a transaction. At commit, we impose an eight cycle arbitration delay and then broadcast all words in the transaction set (at two words per cycle). The second, Bulk [CTTC06] also supports word-granularity and performs write-update on lines with false-sharing word conflicts. It should be noted that lazy conflict detection can be highly advantageous in an environment where memory conflicts are infrequent. We do not expect that to be the case with SpMT, but we still model this approach to better understand the tradeoffs in a combined SpMT/HTM environment.

For the primary LAZY results, we will specifically model TCC; however, we also examined an idealized Bulk design and found that the difference between the two in this context is small. For SpMT on two cores, the only significant differences are that TCC keeps additional state per cache line whereas Bulk maintains signatures — Bulk may transmit less information when performing conflict detection, and Bulk is susceptible to false conflicts. For more than two cores, in addition to these differences, Bulk supports "Partial Overlap" in which data written by a speculative thread prior to spawning a new thread is made available to that new thread. It is important to note that the Partial Overlap optimization is not the same as forwarding. Forwarding allows for values to be transmitted between simultaneously active threads whereas Partial Overlap can only transfer values to new threads at spawn.

In the event of a transactional state overflow (the victim cache fills) by the non-speculative thread, we commit the transaction rather than squash.

In the context of our work, LAZY has a number of advantages as well as disadvantages. By committing the entire write set of a thread when the transaction commits, many coherence operations can be merged and wrong-path memory operations remain invisible to other threads. Additionally, since all information is maintained at a word level, words rather than cache lines are transmitted at transaction commit. However, since writes are only visible at cache commit, LAZY designs are unable to forward data from one cache to another. Finally, for TCC, the burst of a large number of writes consumes the buses and all data caches for a number of cycles.

In TCC, words written in a transaction are buffered. To solve sub-word granularity hazards, a single bit per word address in the store address FIFO is maintained to denote if a write at sub-word granularity occurred. Squashes are required if sub-word writes to the same word in different threads occur.

All data presented here assumes that transactions are committed either when the non-speculative thread overflows the victim cache (a rare event in our results) or when it commits (reaches and commits the CQIP). We explored the possibility of more frequently committing the non-speculative thread to expose the write set to the other threads earlier; however, this provided no real gain due to additional commit overhead and the still present, although reduced, delay in notifying threads of conflict violations and inability to forward data when accessed by another core.

### Ordered Forwarding Word Update (OFWU)

In the previous section, we saw that the updating of cache lines is fairly straight forward for LAZY designs. By ordering thread commit, LAZY designs only broadcast committed data at commit and those lines held by other cores receive the new values. In contrast, adding the write-update feature to eager-conflict detection HTM designs is less straight forward because speculative data is broadcast when modified (and still speculative), not when committed.

For example, adding write update to the protocol introduces a new hazard when we have more than two cores. Consider the case of three speculative threads, thread 0 being least speculative and thread 2 most. All share a cache line. Thread 0 writes a word of that line. Because of the write-update protocol, both threads 1 and 2 get the update but must mark the line as forwarded. Now thread 1 writes a different word in the line, and thread 2 again gets the update and is consistent. But if thread 0 now writes the same word as thread 1 wrote, thread 2 does not know what to do, because it does not store enough information (and it would be prohibitively expensive at a word granularity) to track from where it received each value. To address this situation at minimal cost, we add a *forward distance* field to each line. The observation is that as long as updates to the line happen in monotonically increasing order (from less speculative to more speculative) then there is no ambiguity. The *forward distance* allows us to ensure that the distance (in threads) from the last writer to this thread is decreasing. Once the order departs from that, the line is marked as unsafe by clearing the *safe* bit. When the line is unsafe, the local thread may read words that it has previously read or written, but reads of other words result in a coherence action to restore the full line to a correct state.

*Forward distance* allows for the preservation of updated cache lines and helps by reducing subsequent coherence misses due to invalidations of safe lines. With four cores, our OFWU design reduces memory slowdown by 20% compared with OFWI. Nearly half of this improvement is lost without forward distance bits. Since the goal of the update protocol is to improve memory performance, the *forward distance* is an inexpensive and effective tool for an implementation with more than two cores. Our performance with four cores is further examined in Section 2.6.

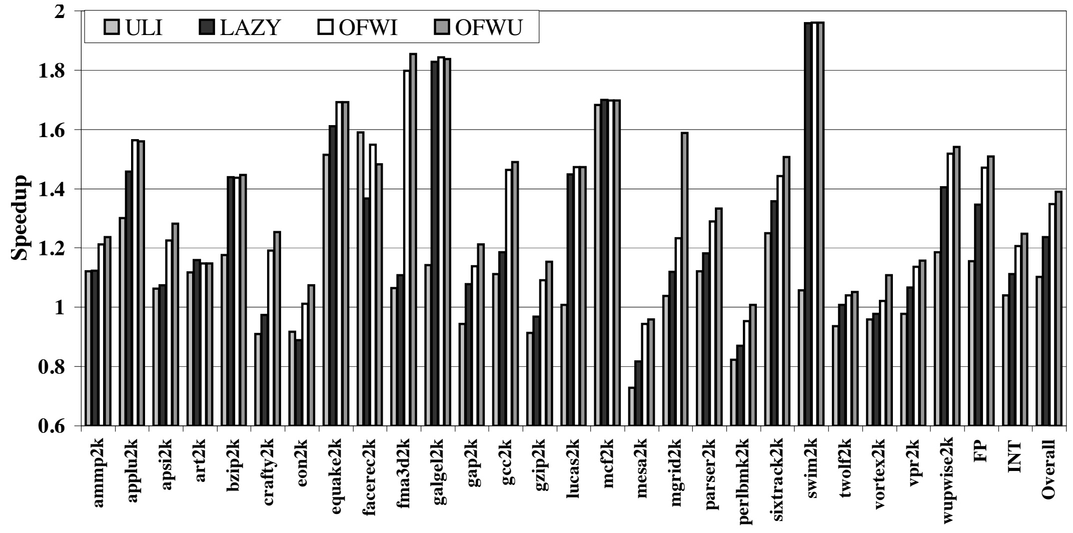


Figure 2.4: Speedup per benchmark for ULI (unordered line invalidate), LAZY (lazy conflict detection), OFWI (ordered forwarding word invalidate, and OFWU (ordered forwarding word update) designs, relative to sequential execution.

### Update Design Results

The results for the ULI, OFWI, LAZY, and OFWU designs are shown in Figure 2.4. LAZY HTMs perform better than the line-granularity eager conflict detection HTMs. However, LAZY HTMs do not perform as well as word-granularity designs with forwarding for this SpMT architecture. This is not surprising, as this is not the execution model for which the lazy conflict detection systems were designed.

Our LAZY design is based on TCC. One key advantage of Bulk over TCC is the amount of data transmitted at commit. To determine if this transfer time was a significant limiter for LAZY, we ran TCC with near-instantaneous transfer times. The results were only slightly better with an average speedup of 1.26 rather than 1.23. This shows that the delay of transferring words (rather than signatures) is not the primary limitation of LAZY for SpMT.

The biggest disadvantage of the LAZY design in this context is the delayed notification of conflicts. A thread that reads data that has been written by a less speculative thread will not only be unable to acquire that data, but will not even find out it needs to squash until the earlier thread becomes non-speculative and commits. In many cases, that could be a long delay, especially if the writing transaction must wait to become non-speculative. In theory, the ability to forward could be added to a lazy conflict detection HTM, but it would represent a dramatic change to both the coherence protocol and the philosophy of lazy conflict detection.

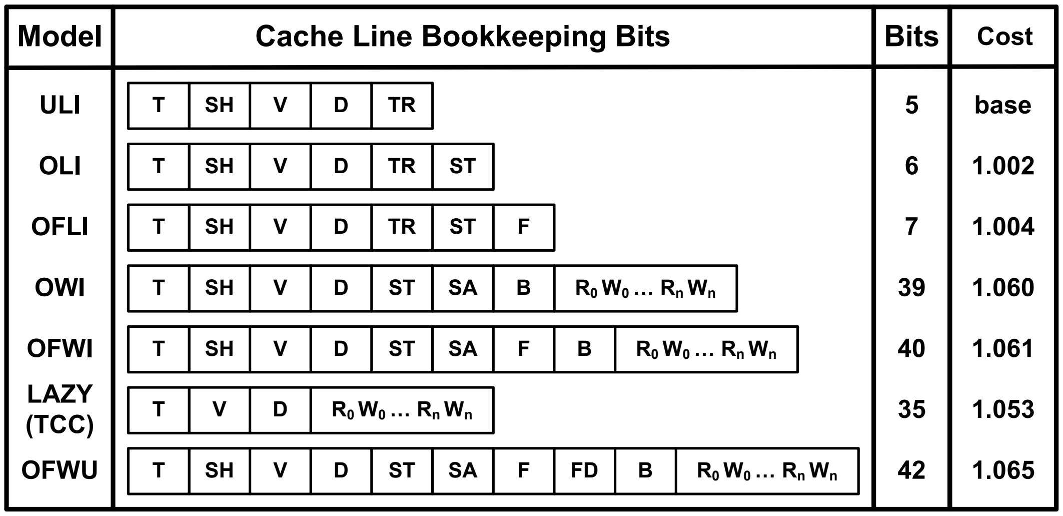
Dependence Aware TM (DATM) [RRW08] also offers word granularity with write-update. As mentioned before, the transactional guarantees by DATM are too weak for our SpMT framework. However, the inability for previous transactions to write to words written by later transactions has a significant effect. Just adding that restriction to our OFWU model caused a slowdown of 9%, resulting in performance similar to OWI and LAZY designs.

Shifting from invalidate to update (OFWU) provided slightly improved performance, but the gains were small. In addition to the overall increase in IPC, coherence misses are reduced from 47% to 37% and the memory slowdown decreased from 1.40 to 1.22 from OFWI to OFWU. We see that average memory access time was improved, partially as a result of decreased coherence misses. But the surprising result is the high rate of coherence misses that remain.

In a conventional write-update machine, all coherence misses would be eliminated. But there are many more types of coherence misses in this system (we use the simplest definition of a coherence miss — a tag hit to an invalid line). There are the delayed invalidates that are not seen until a thread commits, as well as all the invalidates that happen when a thread is squashed. These results show that nearly 80% of the coherence misses were caused by actions that the write-update protocol did not address. Further optimizations to address cold cache problems are the focus of Chapter 3. In Chapter 3, we evaluate the potential of preemptively migrating data from core to core at the point of thread spawn via a recently proposed technique, Working Set Migration [BPT11].

### 2.5.9 Summary of HTM Designs

A summary of the bookkeeping bits required per cache line for all designs is shown in Figure 2.5. We see that the overhead of word-level coherence is clearly the largest storage cost.



| Model | Cache Line Bookkeeping Bits | Bits | Cost |
|---|---|---|---|
| ULI | T, SH, V, D, TR | 5 | base |
| OLI | T, SH, V, D, TR, ST | 6 | 1.002 |
| OFLI | T, SH, V, D, TR, ST, F | 7 | 1.004 |
| OWI | T, SH, V, D, ST, SA, B, $R_0 W_0 \ldots R_n W_n$ | 39 | 1.060 |
| OFWI | T, SH, V, D, ST, SA, F, B, $R_0 W_0 \ldots R_n W_n$ | 40 | 1.061 |
| LAZY (TCC) | T, V, D, $R_0 W_0 \ldots R_n W_n$ | 35 | 1.053 |
| OFWU | T, SH, V, D, ST, SA, F, FD, B, $R_0 W_0 \ldots R_n W_n$ | 42 | 1.065 |

Figure 2.5: Bookkeeping bits per cache line. T - Transactional. TR - Transactionally Read. SH - Shared. V - Valid. D - Dirty. ST - Stale. SA - Safe. F - Forwarded. FD - Forward Distance. B - Byte Written. (R)ead and (W)rite bits for n words. The third column gives the total number of bits, assuming four cores and 64-byte cache lines (n=16). The last column is the associated hardware cost in terms of increased data cache size.

As stated previously, the magnitude of the gains achieved is not the point of this study, and that magnitude would vary with specific assumptions in our architecture. Of greater interest is what optimizations are important to reaching the potential of speculative multithreading without adding significant complexity to hardware TM.

We find word granularity coherence tracking and forwarding are both critical for SpMT performance. Word granularity adds some complexity and a modest amount of hardware overhead, but it should accelerate both SpMT and TM execution.

Despite the high rate of coherence misses, the write update protocol provides modest gains. Given the complexity of supporting that protocol (including difficult corner cases that do not exist in conventional systems), and the fact that based on past research we do not expect write update to be particularly useful for non-SpMT parallel code, this optimization does not appear to be warranted.

Finally, lazy conflict detection proposals provide better performance than existing eager conflict detection proposals but have lower performance than our more advanced designs because of the delay in exposing the result of earlier writes to later computation.

## 2.6 Generality of Results

All of our results so far have examined a specific SpMT architecture. This section varies the details of that architecture to verify that our preceding conclusions still hold. Three dimensions of our architecture likely to change are the number of cores utilized, the register dependence handling mechanisms, and the complexity of the cores. Previous results were restricted to a small 2-core implementation and this section examines a larger (4-core) configuration. We have been simulating a very accurate (actually, perfect) register value predictor to handle the prediction of spawned thread live-ins — again, near-perfect prediction is not necessarily unattainable [MGQS$^+$08] (with a performance cost). However, we also examine a much more conservative register value predictor. Lastly, we examine results for more complex (out-of-order) cores.

Our TM designs were all run with two and four cores. The results for execution on four cores averaged across all benchmarks is shown on the left side of Figure 2.6. We see that we get higher overall speedup with four cores (as much as 45%), but the overall gains have not scaled particularly well primarily because of the increase in cold cache effects, which are the result of two phenomena. The first is the expected increase in cold cache misses as we migrate between cores more frequently. The second phenomenon is that as we move to four cores, we successfully spawn more threads (fewer threads fail to spawn due to unavailable cores). The number of useful threads increases, but the number of squashed threads increases as well. Squashed threads typically result in invalidated lines. These effects are surprisingly large, resulting in an increase in average memory slowdown in the OFWU design from 1.22 with 2 cores to 1.44 with 4 cores. These effects should decrease with higher quality spawned threads (via compiler support, for

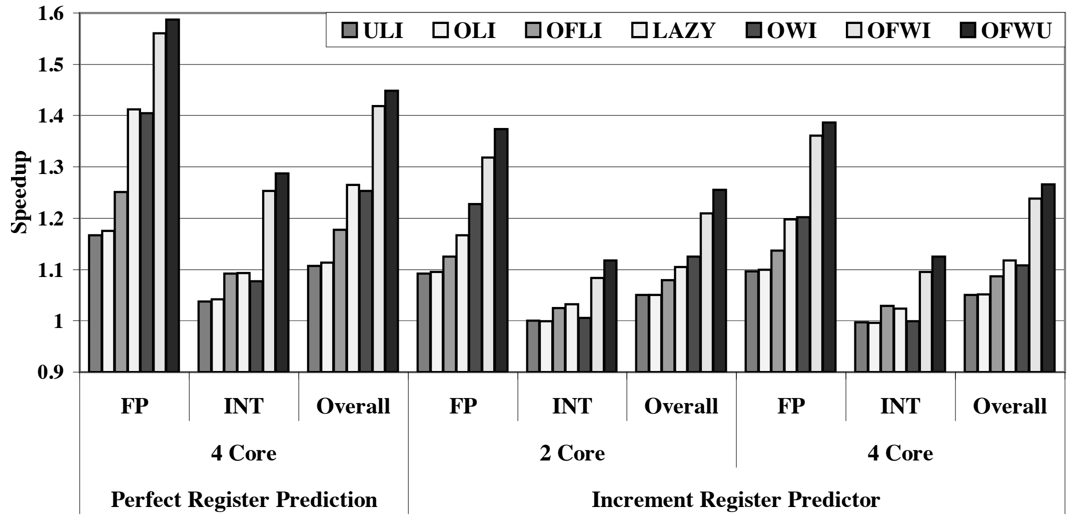


Figure 2.6: Speedup for four cores with perfect register prediction and for two and four cores with an increment register predictor.

example) and better solutions for the cold cache problem. The former is the focus of future research; the latter the focus of Chapter 3.

The more important result, however, is that despite significant changes in the execution details when running with four cores (more parallelism, more threads spawned, more coherence misses, etc.), all of the conclusions of our previous results remain. The word-granularity plus forwarding results provide over 40% performance gains, compared to the baseline TM design which still only gets about 10%. LAZY achieves a speedup of 26% which is less than the OFWI or OFWU designs. We continue to see that forwarding and word-granularity tracking must appear in concert to be effective. We also see the new result that while our aggressive designs provide some scaling with increasing number of cores, the default TM designs do not.

To evaluate the other end of the register prediction spectrum (from our perfect predictor), we have also modeled a basic 4K entry increment predictor [MG02]. Live-ins are tracked for spawnpoints using 2-bit saturating counters per register. Live-in registers are predicted at spawn by adding the increment indexed in the predictor to the current value held by the spawning thread. These predicted values are compared against the actual live-ins at thread commit and if the predictions are incorrect, the validating thread is squashed. This predictor uses simple mechanisms for prediction, indexing, and update while also minimizing storage cost by only storing enough bits to capture most increment sizes. The only change to our previous methodology is that threads which perform illegal operations due to mispredictions are squashed.

The average results across all benchmarks are shown in Figure 2.6 for both 2 and 4 cores. With two cores, the baseline HTM design, ULI, continues to provide limited performance with a speedup of 1.05. LAZY performs better than ULI with a 1.10 speedup. Our OFWI and OFWU designs demonstrate significant improvements, 1.21 and 1.25 speedups respectively. The accuracy of the small increment predictor was quite high, between 63% and 76% depending on the memory design. The average number of live-in registers was between 2.5 and 3.0 depending on design. Undoubtedly, one factor in the high register prediction accuracy was that the confidence counters eliminated (for spawning) those threads with poor register prediction.

We also evaluate the trends for out-of-order cores with 2 and 4 cores using either a perfect or conservative register predictor, and determine that the same trends and results hold with the more powerful cores. For example with the perfect predictor and four cores, we found speedups of 1.09 for ULI, 1.12 for OFLI, 1.17 for LAZY, 1.33 for OFWI and 1.35 for OFWU.

We see in all three cases (more cores, realistic prediction, and more complex cores) that all the important trends continue. We see similar gains over the baseline HTM design for the same best designs, LAZY falling between the baseline design and the better designs, word granularity being critical, and word-level granularity and forwarding providing highly synergistic gains.

## 2.7 Chapter Summary

Speculative multithreading has the potential to significantly increase our ability to leverage highly parallel multi-core, multithreaded processors for code that lacks the characteristics of traditional parallelism. Transactional Memory provides support for optimistic concurrency as well as an easier parallel programming interface and has such gained hardware support in processors proposed by industry. Prior work has recognized the ability for TM to support a domain of speculative multithreading limited to threads which are independent (dynamically).

This chapter evaluates the path from TM to full SpMT. It evaluates a number of intermediate design points between TM and SpMT. It identifies a unified architecture capable of exploiting transactional parallelism, compiler-identified speculative parallelism, and hardware detected speculative parallelism. Three features are identified at being critical to SpMT performance — ordering of transactions, the ability to forward between transactions, and coherence tracking at word granularity. These trends persist across processors with in-order or out-of-order execution, two or four cores, and perfect or conservative register prediction.

## Acknowledgments

The work in this chapter was supported in part by the Reserve Officers Association Henry J. Reilly Memorial Scholarship, NSF Grant CCF-0702349, and Semiconductor Research Corporation Grant 2005-HJ-1313.

This chapter contains material from “Mapping Out a Path from Hardware Transactional Memory to Speculative Multithreading”, by Leo Porter, Bumyong Choi, and Dean M. Tullsen, which appears in *Proceedings of the Eighteenth International Conference on Parallel Architectures and Compilation Techniques (PACT).* The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2009 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

# Chapter 3

# Improving Speculative Multithreading Memory Behavior

As shown in Chapter 2, Speculative Multithreading (SpMT) is a promising technique for improving single-thread performance by leveraging idle parallel resources [SBV95, HHS$^+$00, MGQS$^+$08, SM98]. However, poor cache performance was shown to be an obstacle for realizing the full potential of SpMT. In this chapter, we focus on addressing this problem.

Working Set Migration (WSM) is a recently proposed technique for mitigating performance losses at the time of thread migration [BPT11]. WSM has been shown to be effective for an artificial workload which forces migrations at arbitrary points of execution. This work seeks to answer the following questions:

- Can WSM aid cache performance for a system with non-arbitrary and frequent thread migrations–namely a Speculative Multithreaded workload?
- Are the heuristics recommended for the artificial workload also the best choice for an SpMT workload?

We demonstrate that WSM can be effective for SpMT using the correct WSM heuristics. For the SPECint benchmarks, applying WSM to SpMT reduces average memory access time by nearly 50%; resulting in an average improvement to whole program performance of 9%.

The rest of the chapter is organized as follows. Experimental methodology is described in Section 3.1. Section 3.2 motivates this study. Section 3.3 discusses prior related work. Section 3.4 provides background on WSM techniques. In Section 3.5, we evaluate WSM techniques for improving SpMT performance. We conclude with Section 3.6.

## 3.1 Methodology

We use the Speculative Multithreading framework described in Chapter 2.3 and [PCT09]. We modify it to evaluate the working set migration techniques described in the following sections. To review, in this framework, loops and function calls are identified for speculative parallelization entirely in hardware. Register dependencies between threads are predicted by a live-in predictor and the values are predicted using an increment predictor [MG02]. Memory dependencies are addressed via a modified form of Hardware Transactional Memory, specifically the OFWI design recommended by [PCT09]. This memory design is aware of thread ordering, forwards values between threads, and detects conflicts at word granularity. Since the protocol creates invalidations, write-sharing can cause additional cache misses. We perform working-set-prediction prefetches non-transactionally, so they do not impact the read/write sets of transactions.

We evaluate the full SPEC CPU2000 benchmark suite with reference inputs. Each benchmark is executed using 100M-instruction simulations based on SimPoint [SPHC02]. We model dual-core execution using architectural parameters similar to [PCT09]; these parameters include a shared L2 cache, which decreases the penalty for transferring data between cores.

## 3.2 Motivation

The benefits of caches for single-threaded performance are well established [HP02]. Caches enable high performance by providing storage close to computation resources on chip. Accesses to local (on-chip) caches can be orders of magnitude faster than accesses to off-chip resources, thus significant performance gains can be achieved by having local caches satisfying most requests. Caches attempt to satisfy these requests by learning about thread memory behavior. To this end, caches retain blocks of recently accessed data and those threads which have the tendency to reuse those blocks benefit. Because caches learn over time, threads need to run on a given core for some time before the benefits of caching can be realized [BPT11].

SpMT disrupts the performance of caches by migrating computation between cores through the spawning and committing of speculative threads on idle cores. A number of factors contribute to poor cache behavior for SpMT:

- Speculative threads are spawned on idle cores, the caches of which may be cold, i.e. the cache is either empty or its contents are irrelevant to current computation. The caches on these cores require time to learn, and performance suffers during this warm-up time.

- When speculative threads successfully commit, they become non-speculative and the main thread of computation migrates cores. This moves computation from a core with a typically warmer cache to a core with a colder cache.

- When speculative threads fail to commit, any data written by that thread is invalidated (evicted from the cache). Failed speculative threads hence contribute to the loss of cache state among cores.

- As speculative threads execute, data is shared between cores. If cache coherence is managed via an invalidation protocol, threads invalidate the cache contents of threads running on other cores, worsening their cache performance.

Potentially offsetting the many factors which contribute to poor SpMT cache performance, SpMT has one behavior which can improve cache performance. Speculative threads are a form of runahead execution in that they perform future computation which can prefetch future memory accesses into shared caches [MSWP03].

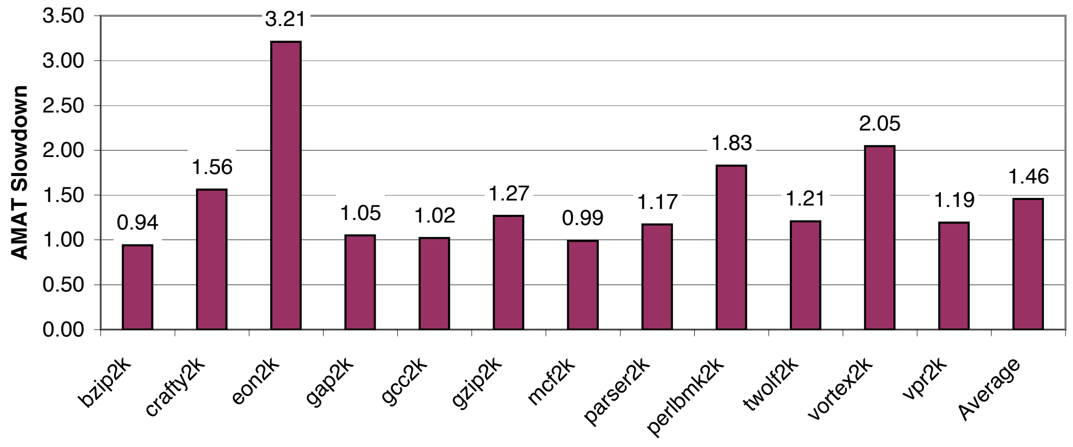


Figure 3.1: Average Memory Access Time Slowdown for SpMT relative to single-threaded execution for the SPECint benchmarks.

To evaluate the impact of cache behavior on SpMT performance, we compare Average Memory Access Time (AMAT), the average time for each memory request, between single-threaded and SpMT execution. For SpMT, AMAT is just the average memory access time for memory references in non-speculative and committed threads. Memory accesses for squashed threads are not included in this calculation.

Figure 3.1 provides the AMAT slowdown from SpMT for the SPECint benchmarks. For these benchmarks, some suffer significant AMAT slowdowns (*crafty2k*, *eon2k*, *perlbmk2k*, and *vortex*). Interestingly, two benchmarks have their memory performance slightly improved by SpMT, *bzip2k* and *mcf2k*. These benchmarks are likely benefiting from the prefetching effect of the speculative threads. Averaged across SPECint, AMAT increases by 47% due to the negative memory impact of SpMT.

The AMAT slowdown for the SPECfp benchmarks appears in Figure 3.2. Although the negative impact of SpMT memory behavior is less pronounced on SPECfp than for SPECint, many SPECfp benchmarks still suffer increases in AMAT. Other FP benchmarks experience improvements to AMAT from prefetching speculative threads, most notably *equake* which achieves an 18% reduction in AMAT. Averaged across SPECfp, the negative memory impact from SpMT slightly outweighs the advantage, resulting in an increase in AMAT by 6%.

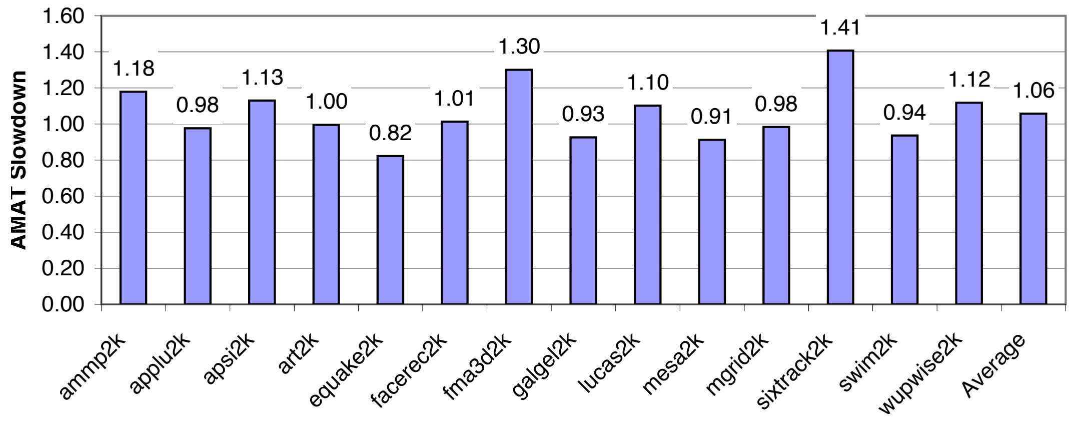


Figure 3.2: Average Memory Access Time Slowdown for SpMT relative to single-threaded execution for the SPECfp benchmarks.

Averaged across all of the SPEC CPU2000 benchmarks, SpMT causes a 25% increase to AMAT. This slowdown to AMAT will be shown to hamper whole program SpMT speedups. Thus, by improving SpMT memory performance, we can improve the effectiveness of SpMT and its ability to improve single-threaded performance.

## 3.3 Related Work

Prior work has examined the problems of poor performance due to (1) general migration costs and (2) SpMT's impact on cache behavior.

### 3.3.1 Reducing Migration Costs

Previous work [BT08, TKTC04] describes support mechanisms for migrating register state in order to decrease the latency of thread activation and deactivation; however, performance subsequent to migration still suffers due to cold-cache effects. Our work is complimentary; we specifically address the post-migration cache misses which limit the gains of those techniques. Choi, et al., explore the complementary problem of branch prediction for short-lived threads, specifically branch prediction for SpMT [CPT08].

Data Marshaling [SMJ$^+$10] mitigates inter-core data misses in Staged Execution models. In contrast to the techniques evaluated here (Working Set Migration), DM targets scheduled stage transitions using compile-time flagging of producer instructions, hardware to track writes by flagged instructions, and a new instruction which triggers data transfers.

Working Set Migration (WSM) techniques have been proposed in the context of frequent thread migrations by Brown et al. [BPT11]. In their work, they demonstrate WSM techniques to improve the performance of short threads which result from frequent thread migrations in artificial workloads. This work evaluates their technique on a more realistic workload — SpMT. The following section discusses this work in more detail and Section 3.5 evaluates using this technique for SpMT.

### 3.3.2 Cache Performance for SpMT

In Speculative Multithreading (SpMT), loss of cache state impedes performance as execution migrates across cores [FS06, VGSS01]. This is a well-documented and long-standing problem. Execution that would ordinarily reside on a single core is now spread across several, creating misses and invalidate traffic where the original code experienced hits in a single cache. Additionally, coherence-based speculative multithreading requires certain data to be invalidated in caches for both squashed and committed threads, exacerbating the problem.

Speculative Versioning Cache [VGSS01] and the work of Fung and Steffan [FS06] use bus snarfing to pull requested lines into adjacent cores. We evaluated the effectiveness of snarfing in our framework and found limited gains (5% AMAT reduction, 2% speedup across SPEC CPU2000). Fung and Steffan additionally evaluate the potential of keeping a small buffer of recently stored addresses [FS06] which were requested for read sharing by other threads. At thread commit, those values in the buffer are written back and pushed to the next thread context. Unlike the WSM techniques evaluated here, this technique does not aid the execution of the speculative thread at the critical point when the speculative thread begins execution.

## 3.4 Background - Hardware Support for WSM

In [BPT11], Brown et al. demonstrate that miss-stream characterization is insufficient to cover many migration-related misses. As such, they developed the following design for WSM to characterize the entire *access* stream. The hardware consists of a working set predictor which works in three stages. First, it observes the access stream of a thread and *captures* patterns and behaviors. Second, it *summarizes* this behavior and transfers the summary data to the new core. Third, it *applies* the summary via a prefetch engine on the target core to rapidly fill the caches in advance of the migrated thread.

WSM has been evaluated with a set of possible capture engines. We evaluate these same capture engines to determine if the design recommended by [BPT11] for artificial workloads is also the correct choice for SpMT workloads. The remainder of this section describes the WSM hardware and various capture engines described in [BPT11] and tailored to our SpMT framework for evaluation.

### 3.4.1 WSM Hardware

The WSM hardware consists of three main components shown in Figure 3.3:

- **Memory Logger** — During execution, the Memory Logger monitors and records sections of the access stream. Depending on which policy is applied, different sections of the access stream are recorded. These policies are described in full detail in the following section. Since the memory logger need not be correct, it can be removed from the critical path and should not impact performance.

- **Summary Generator** — At the point of thread spawn, the Summary Generator reads the logs from the memory logger, compacts the address information into cache-line size summaries, and transmits those summaries to the core which is the target of the spawn. This transmission occurs in parallel with the transfer/prediction of register state.

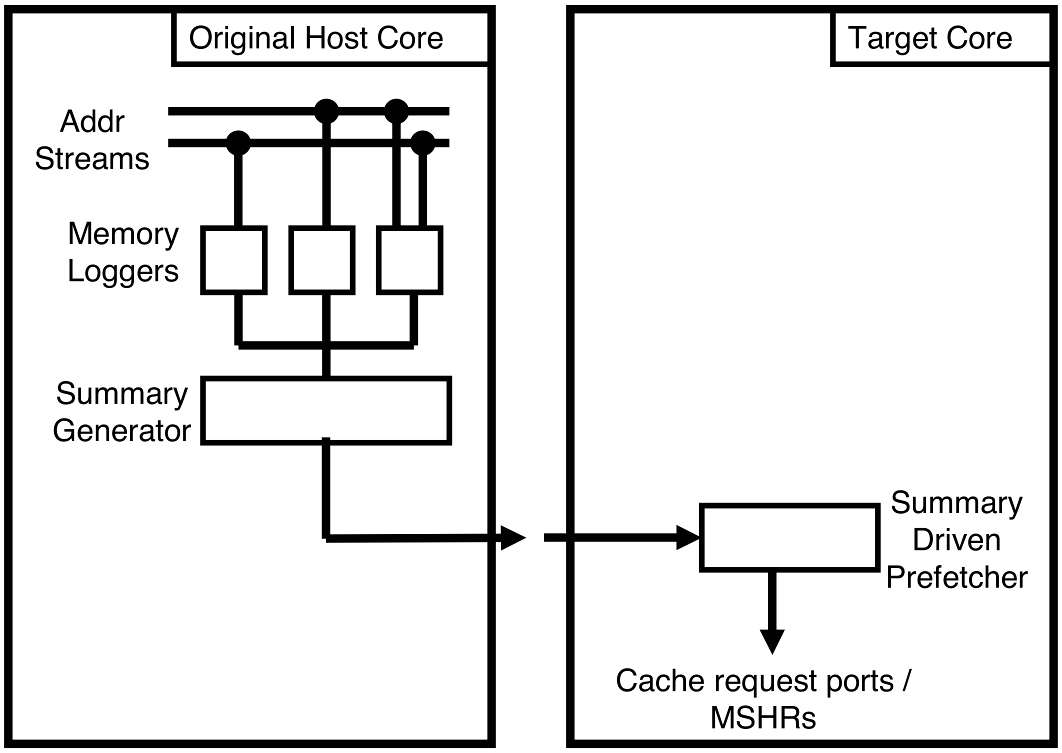


Figure 3.3: Hardware for Working Set Migration. Although each core has all three components, the only components shown are those which which contribute to the host core spawning a speculative thread on a target core.

- **Summary-driven Prefecher** — As the speculative thread begins activation on the target core, the summary records (which are transmitted from the parent cores Summary Generator) are read and prefetches are generated by the Summary-driven Prefetcher. These prefetches begin in parallel with register state transfer and continue while the speculative thread begins execution. All prefetches search the entire memory hierarchy and contend for the same resources as demand requests.

All three of these components have been added to each core in our SpMT architecture.

## 3.4.2 Memory Logger Policies

Much of the effectiveness of the WSM techniques is dependant on which of the Memory Logger policies discussed in [BPT11] are used. For each of these policies, small tables are used to log the appropriate information as small tables were shown to be effective in [BPT11]. The following are the policies for evaluation:

*Next-block-*{*Inst,Data*}*:* These monitor the I- and D-stream, respectively, for sequential block accesses. On a hit, entries are incremented by one cache block. We maintain a separate table for each of the I- and D-streams.

*StridePC:* This is designed to track individual instructions walking through memory with a consistent stride. This table only tracks D-stream accesses, using the PC values and access addresses.

*Pointer, Pointer-chase:* These capture active pointers and pointer traversals by detecting when the value from load subsequently matches the address of a later memory request, similar to pointer-cache [CSCT02] and dependence-based [RMS98] prefetching. *Pointer* tracks previously loaded values used as addresses without following them. *Pointer-chase* follows the pointer by replacing each entry with the target value when a load match is observed.

*Same-object:* This attempts to capture accesses to the same structure or object, by monitoring memory ranges accessed from from a common base address. Using the "base+offset" addressing mode, it tracks the minimum and maximum offsets observed for any given base address. (Accesses to the global or stack pointer are ignored.)

*SPWindow, PCWindow:* These do not require tables as only the current value of the stack pointer and PC are required. We use the stack pointer and PC, respectively, to prefetch a region of data blocks near the top of the stack or the region near the program counter of the first instruction after migration.

{*Inst,Data*}*-MRU:* These record the Most Recently Used (MRU) blocks accessed from the I- or D-stream. These addresses are managed at a four-cache-block granularity enabling them to cheaply cover a larger number of blocks.

*BTB, BlockBTB:* These capture taken branches and their targets by recording the most recent in-bound branch for each target. The latter is a block-aligned variant of the former. Both branch and target PCs are block-aligned, allowing a larger section of the instruction working set to be characterized with a smaller table size.

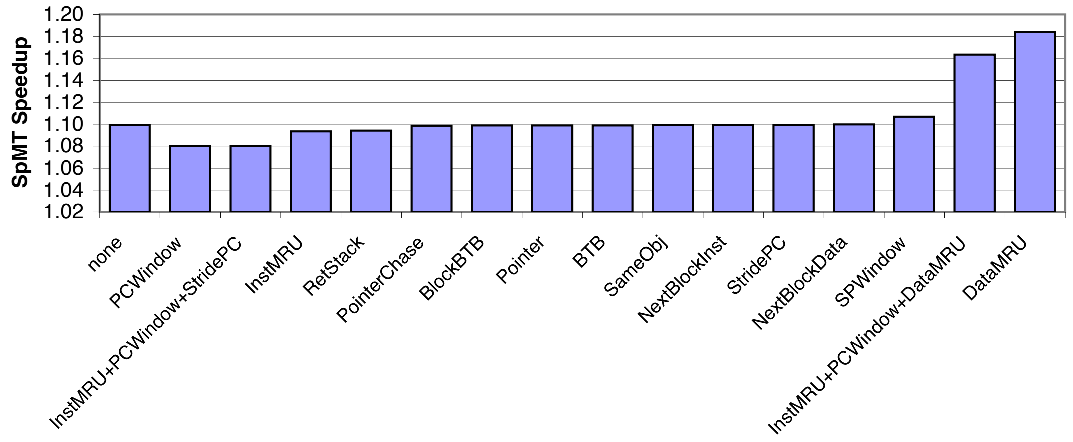


Figure 3.4: Impact of WSM policies on SpMT Speedup for SPECint.

*RetStack:* This maintains a shadow copy of the return stack in order to prefetch instruction blocks in the region of the top of a few control frames.

The potential for each of these policies to capture the working set of speculative threads is examined in the following section.

## 3.5 WSM for Speculative Multithreading

To evaluate the effectiveness of the WSM techniques for SpMT, we first evaluate each of the memory logging policies to determine which policies are most successful. We then return to the issue of the impact of SpMT on AMAT in the context of WSM techniques. Lastly, we evaluate the whole program speedup possible from using WSM techniques with SpMT.

### 3.5.1 WSM Policies

In [BPT11], the authors recommend the *InstMRU+PCWindow+DataMRU* policy. They also note that the largest impact on thread performance post migration is the lack of working set in the instruction cache.

We reevaluate the WSM policies in the context of SpMT in Figure 3.4. Figure 3.4 contains the average SPECint SpMT speedup for each of the different WSM policies. For SpMT workloads, the instruction cache migration techniques

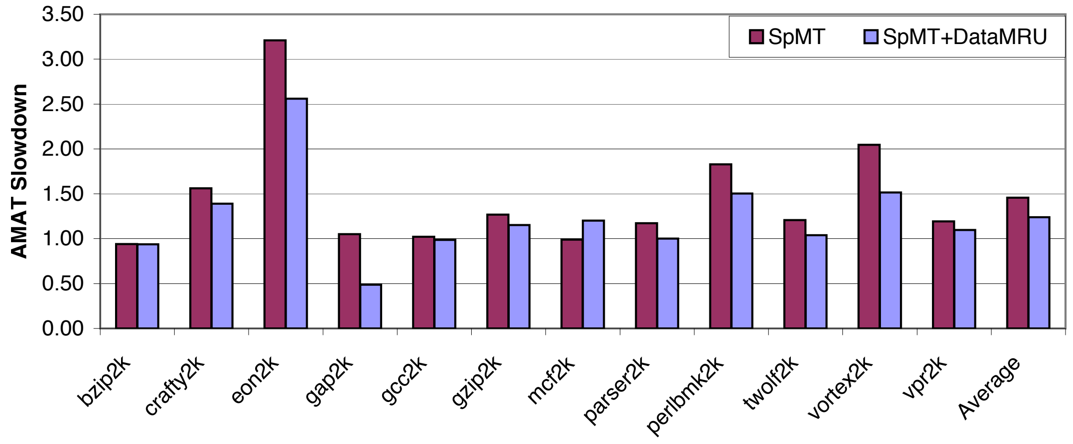


Figure 3.5: SPECint per benchmark AMAT slowdown from SpMT and from SpMT with DataMRU.

are not effective alone, and in the cases of *InstMRU+PCWindow+InstMRU* and *PCWindow*, can be disruptive. This is likely due to the instruction cache warming quickly from frequent thread migrations. One data cache policy which was particular effective for the artificial workload in [BPT11] was *StridePC*. For SpMT, *StridePC* is less effective. This is likely caused by *StridePC* needing time to learn the strides between memory addresses which is difficult with thread migrations so frequent. However, the other data cache policy recommended in [BPT11] is *DataMRU*. *DataMRU* is the most effective policy for SpMT. The combination of *InstMRU+PCWindow+DataMRU* is slightly less effective. In subsequent sections, we'll continue to evaluate WSM potential in the context of these effective policies (*InstMRU+PCWindow+DataMRU* and *DataMRU*).

### 3.5.2 WSM Impact on AMAT

As shown in the motivation (Section 3.2), SpMT speedup is hampered significantly by the increase in average memory access time (AMAT) that occurs when we spread the computation among multiple cores. However, the data working-set prediction provided by (*DataMRU*) significantly mitigates the AMAT inflation in SPECint, as shown in Figure 3.5. AMAT is also improved across SPECfp as shown in Figure 3.6. In some cases, it reduces AMAT to near, or even below, the single-thread AMAT (single-thread AMAT is 1.0). In nearly all other cases, prefetching

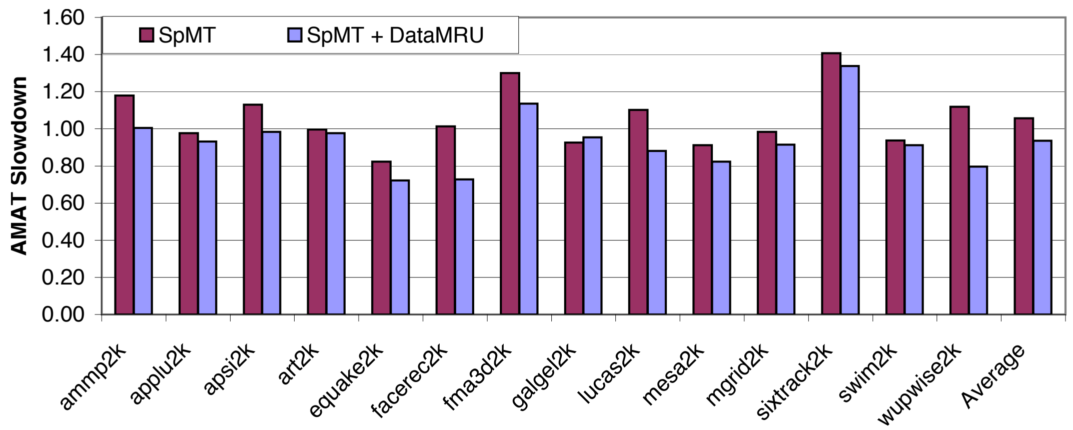


Figure 3.6: SPECfp per benchmark AMAT slowdown from SpMT and from SpMT with DataMRU.

significantly reduces AMAT.

One anomalous result is *mcf*. The performance of *mcf* is completely dominated by a small number of hard-to-predict "delinquent" loads [CWT+01]. SpMT benefits *mcf* less from the successful completion of spawned threads, as from the prefetching provided by those threads, which bring in hard-to-predict data. Hence, for *mcf*, speculative threads already succeed at performing critical prefetches; those critical prefetches are delayed by the extra traffic generated by DataMRU prefetching.

### 3.5.3 WSM Impact on SpMT Speedup

Figure 3.7 explores the effectiveness of WSM techniques for improving whole program performance for the SPECint benchmarks. Focusing on the average, we see that baseline SpMT execution achieves an average speedup of 1.10, and D-stream WSM nearly doubles the overall effectiveness of speculative threading for the integer benchmarks, increasing the gain from 10% to 18%.

Figure 3.8 explores the effectiveness of WSM on the floating point subset (SPECfp). In general, the regularity of the code in SPECfp enables higher baseline SpMT performance (36% improvement over single threaded performance). For SPECfp, D-stream WSM is again effective at boosting the average SpMT performance by 8% (to 44%). As such, when averaged across the entire SPEC

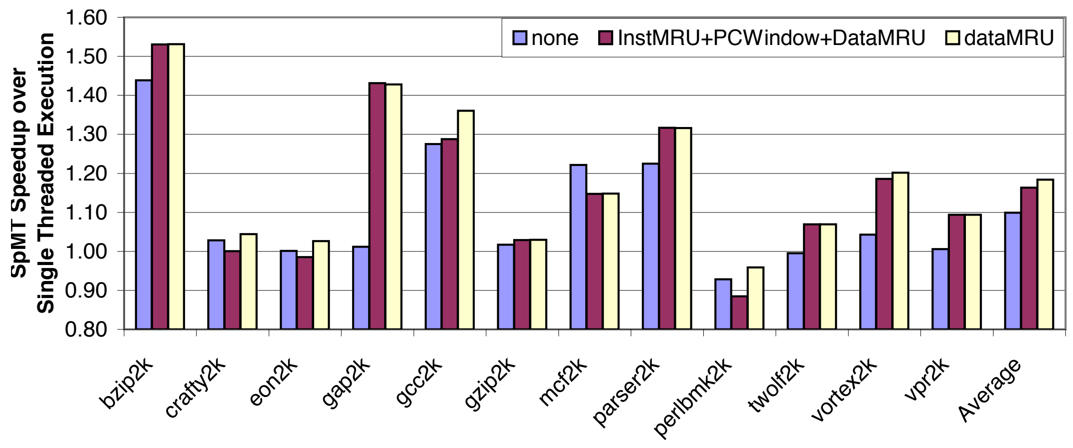


Figure 3.7: SPECint SpMT speedup per benchmark for different WSM techniques.

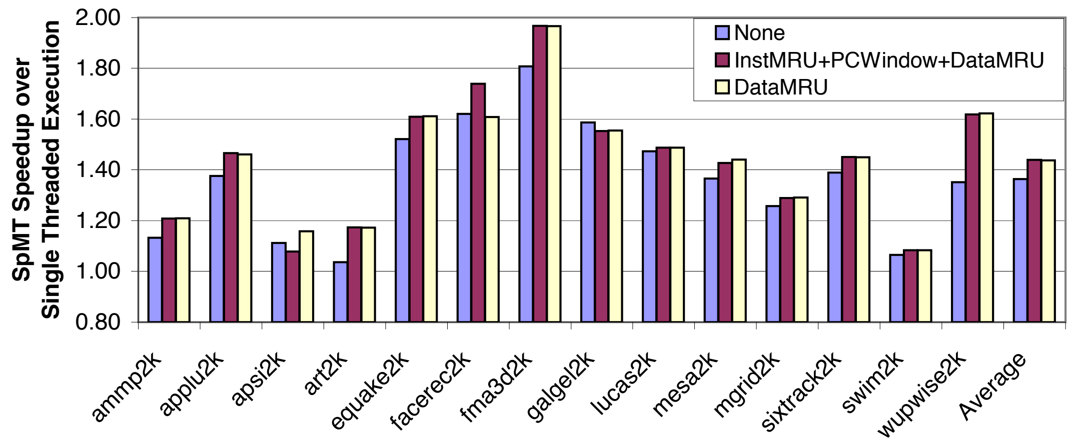


Figure 3.8: SPECfp SpMT speedup per benchmark for different WSM techniques.

CPU2000 benchmark suite, WSM increases overall gains from 24% to 32%. When comparing the combination of prefetching both instruction and data streams (*InstMRU+PCWindow+DataMRU*) against that of data alone (*DataMRU*), we see that I-stream prefetching is harmful for SPECint and ineffective for SPECfp. I-stream prefetching is slightly less effective on average primarily because the I-caches tend to be already warm. Although instruction prefetches are likely to be hits, they can impede the thread because they occupy cache request ports.

In addition to the SPECint average speedups previously shown in Figure 3.7, Figure 3.9 provides results for an oracle D-prefetcher, which, at each spawn point, prefetches the memory blocks used by the next 100 instructions (the

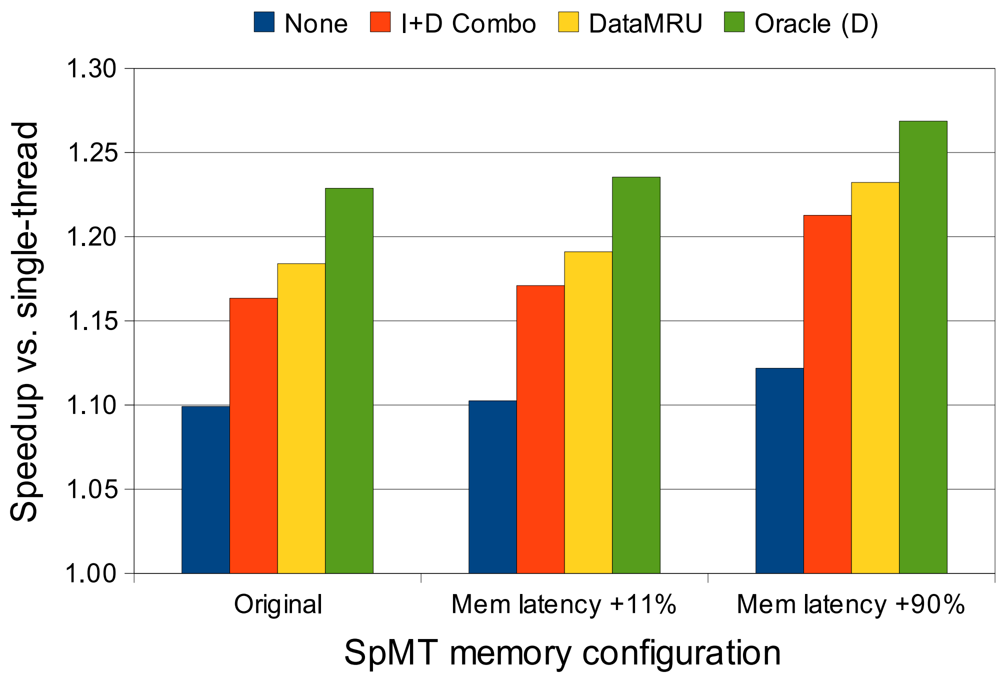


Figure 3.9: SPECint SpMT mean speedup across migration techniques and memory configurations. “I+D Combo” uses InstMRU, PCWindow, and DataMRU.

average speculative thread length is 59). Focusing on the left-most columns, this oracle achieves a 1.23 average speedup for the SPECint benchmarks. Since this oracle prefetcher should almost entirely solve the cache locality problem for SpMT, we find that our realistic working-set prefetcher achieves most of the gains (1.18 speedup) available.

### 3.5.4 Generality of WSM Results

As in the general case from [BPT11], post-migrate performance is dominated by cache-to-cache transfers. Prior work [SZG$^+$09] shows that memory simulation models such as ours can underestimate latency by up to 25%. Since main memory access patterns are largely unaffected by WSM, the details of that simulation should not impact our relative speedups. To demonstrate that our results are consistent even if our model underestimates memory latency, we add additional latencies to memory and reevaluate the results. In Figure 3.9, the performance of SpMT with different memory latencies is shown. The left group has our original memory latency, and the center and right groups show the performance with DRAM access latencies increased by 11% and 90%, respectively. These results demonstrate that the overall trends in this section are insensitive to large increases in memory latency.

To model a more aggressive SpMT system, we also evaluate the SpMT framework using a perfect register value predictor to determine if our results remain consistent without thread squashes caused by register mispredictions. Again, we see significant gains from the DataMRU working set predictor, which improves SPECint SpMT speedup from 1.24 to 1.34 and all SPEC CPU2000 SpMT speedup from 1.38 to 1.49.

## 3.6 Chapter Summary

SpMT performance is hampered by poor cache performance in the presence of thread migrations, thread spawns, and coherence invalidations. WSM offers a potential solution in that it has been designed to be a low cost method of migrating working set at the point of thread migration. However, WSM has only been demonstrated to be effective for workloads with artificial migrations.

In this chapter, we apply WSM to SpMT and demonstrate that the WSM policy of *DataMRU* is most effective for SpMT workloads. Unlike the artificial workload used to originally demonstrate the value of WSM, instruction caches warm quickly in SpMT workloads and become less critical. Similar to the artificial workload, *DataMRU* is shown to be an effective means of working-set migration for SpMT. *DataMRU* provides an 8% improvement to whole program SpMT speedups for the SPEC CPU2000 benchmarks and nearly doubles the effectiveness of SpMT for the SPECint benchmarks (improving the speedup from 10% to 18%).

## Acknowledgments

This research was supported in part by NSF grant CCF-1018356 and Semiconductor Research Corporation Grant 2005-HJ-1313.

This chapter contains material from "Fast Thread Migration via Cache Working Set Migration", by Jeffrey A. Brown, Leo Porter, and Dean M. Tullsen, which appears in *Proceedings of the Seventeenth International Symposium on High Performance Computer Architecture*. The dissertation author was the secondary investigator and author of this paper.

# Chapter 4

# Creating Artificial Global History to Improve Branch Prediction Accuracy

In the previous chapters, we examined techniques for leveraging idle parallel resources to improve single-threaded performance. In this chapter, we visit the complementary problem of freeing resources from components previously dedicated solely to single thread performance. These freed resources can be used elsewhere (more cores, better interconnect) to improve processor-wide performance. This reallocation of resources is done with the critical caveat that we do so only with a minor impact on single-thread performance. In this chapter, we focus on branch prediction hardware because branch prediction accuracy is tied solely to performance, not correctness.

The importance of accurate branch prediction has been well documented in the literature. As such, modern processors rely on highly accurate branch prediction for good performance. In this work, we propose a simple technique based on heuristics that improves branch prediction for a number of branch predictors. Smaller, less complex branch predictors benefit the most from our technique.

In the uniprocessor era, it made sense to pursue increasingly larger, more complex predictors to squeeze out even small margins of performance. In the multi-core era, every transistor we do not use on one core can be put to other use–allowing more cores, faster cores, more cache, better interconnects, etc. This is even more true of the power envelope, as future processors will be designed under very tight power constraints. Therefore, the highest performance processor is composed not from the highest performance building blocks, but rather from the most area-efficient and power-efficient building blocks. Thus, even the branch predictor must carefully justify the use of its transistor budget, and smaller predictors may provide higher processor-wide performance by better utilizing those resources elsewhere. This research demonstrates techniques that improve the branch prediction accuracy of most modern predictors. By requiring no additional storage and minimal logic, it improves overall processor performance with no power or real estate cost. Moreover, as it is most effective on small predictors, allowing those to become more competitive with larger, more complex predictors, it potentially enables a reduction in predictor size with no cost in per-core performance.

To produce highly accurate predictions, modern branch predictors use global history to index prediction tables [YP93, CEP96, Kes99, LCM97, McF93, SFKS02, SM99], as input to a neural network [JL02], or as tags in table lookup [EM98, Sez05]. Global history is successfully used to produce accurate branch predictions because branches often correlate with previously executed branches (other nearby branches and themselves). Longer branch histories enable predictors to view a larger window of previously executed branches and learn based on correlations with those branches.

Evers, et al. [EPP98] demonstrate that the amount of correlation with prior branches varies per branch. For branches highly correlated with recent history, global history can provide key prediction information. However, we show that for a branch that is not highly correlated, that history is mostly noise and does more harm than good. It increases the time to train the predictor and it significantly expands the level of aliasing in the prediction tables, reducing the accuracy of prediction on this and other branches.

The technique we propose directly modifies global history based on simple code heuristics. These heuristics identify sections of code where branch correlation is likely to be low, and modifies the GHR to reduce or eliminate unnecessary noise.

Specifically, these heuristics target backward branches, function calls, and returns.

We evaluate these techniques using select SPEC CPU2000 benchmarks and the Championship Branch Prediction (CBP) traces from The Journal of Instruction Level Parallelism. These techniques provide gains for a number of branch predictors. For a set of 32Kb predictors, these techniques improve each of the A21264 [Kes99], gshare [McF93], and alloyed perceptron predictors [JL02]. For the perceptron predictor, our technique reduces mispredicts per thousand instructions (MISP/KI) by 12% overall. A 416Kb implementation of 2Bc-gskew [SM99] also benefits from our techniques, achieving a 9% reduction in MISP/KI for the CBP traces. Smaller predictors benefit more from these techniques–an 8Kb implementation of gshare achieves a 18% overall reduction in MISP/KI for our select SPEC CPU2000 benchmarks.

This work provides the following contributions:

1. We provide a technique of GHR modification that has exceptionally low hardware cost and demonstrate that branch prediction can benefit from its use.

2. The benefit of this technique is shown for a number of branch predictors of low to medium complexity and for a number of branch predictor sizes. These gains are shown both on a selection of SPECint benchmarks and the Championship Branch Prediction Competition traces.

This chapter is organized as follows: Section 4.1 provides the motivation and basic architectural approach. Section 4.2 discusses related work. Section 4.3 discusses our hardware techniques based on code heuristics, and also discusses some potential hardware/software techniques. Section 4.4 provides the methodology. Section 4.5 provides results and Section 4.6 concludes.

## 4.1 Motivation

The global history register (GHR) allows a predictor to exploit correlations with recently executed branches. A longer GHR enables correlation with more

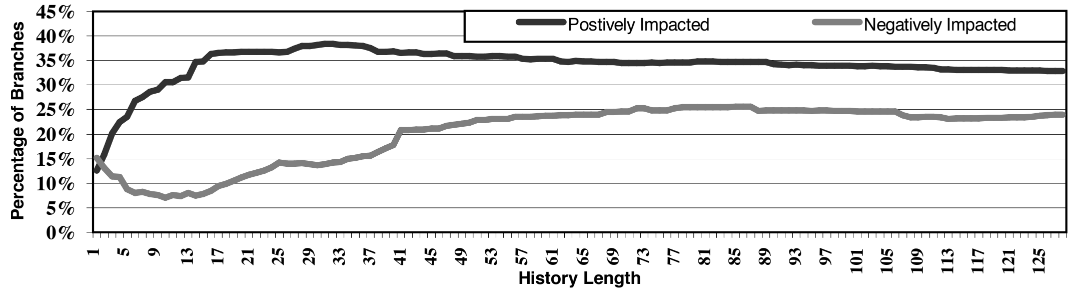


Figure 4.1: Percentage of dynamic branches positively or negatively impacted by history.

distant branches, but also increases the number of uncorrelated branches that are included in the history. Those uncorrelated branches can create significant noise. Consider a 15-bit GHR. A branch that is highly correlated with 3 prior branches will make good use of a correlating predictor; but even in this positive scenario, the history contains 12 bits of useless noise. This means that (worst case), we could have to use $2^{12}$ times more entries to predict this branch than we need, greatly increasing the training period and the aliasing with other branches. For a branch uncorrelated with prior branches, the entire 15 bits are noise, only serving to confuse the predictor and pollute the tables.

In our simulations of select SPEC benchmarks, we found that the average branch, using a 16-bit GHR, observes over 100 different branch histories; with 20 bits, it more than doubles and we see over 200 histories per branch. Some of those histories will be useful and indicate useful correlations, but most will not [EPP98]. Thus, it is reasonable to surmise that on average we are using dozens of times more table entries than desired.

Figure 4.1 shows the percentage of branches that achieve either loss or benefit from correlation history. It compares the accuracy of a correlating predictor (a unique 2-bit predictor for every possible history) with the actual bias of the branch. For example, if the correlating predictor achieved 78% accuracy, but the branch was 90% biased (toward taken or not-taken), we would say this branch was negatively impacted by history. The remaining branches not indicated on this graph had less than a 1% difference between the two.

In this graph, we see (1) that in all cases there are a number of branches that degrade because of branch history, and (2) that for larger histories, the number of degraded branches can be quite large. The effect we are observing is that with large histories, the noise begins to dominate. This leads to the somewhat counter-intuitive result that the longer the history, the fewer branches actually make effective use of that history.

Clearly, some branches do gain from large history, as prior work has shown. What we want, then, is an architecture that allows some branches to benefit from large histories, but eliminates or reduces the history noise in those regions where the noise is not useful. While prior research examined a number of techniques that target the optimal history for individual branches, this paper focuses on a surprisingly effective (yet simple) heuristic that only requires that we identify a place in the execution stream where we transition from one region (with potentially high correlation) to another, but where there is not correlation between the regions. It turns out that this captures the most significant correlation gaps.

Not surprisingly, the control flow instructions themselves are the most useful clues to these transition regions. The transitions we found to be most interesting are the transitions between procedures (indicated by call and return instructions), and loop exits (often indicated by not taken backwards branches). Upon identifying one of these transitions, we do better if we ignore all prior history (thus collapsing all possible paths into one). By setting the GHR to a single known value, we begin training the predictor for the subsequent branches quickly, and greatly reduce pollution and aliasing.

### 4.1.1 Source Code Example

*Gcc*, a SPEC CPU2000 integer benchmark, benefits from our techniques. An example from that benchmark which serves to illustrate the advantage of our techniques appears in Figure 4.2. Branch A is the first branch instruction in a function call which performs a null pointer check. The branch executes 2455 times during the execution of the 100M instruction simpoint and is never taken. The branch encounters 208 unique 16-bit histories. As a result, it is only predicted correctly 91% of the time using non-aliasing 2-bit predictors per history. If you

```
static void sched_analyze_2 (x, insn)
     rtx x;
     rtx insn;
{
  register int i;
  register int j;
  register enum rtx_code code;
  register char *fmt;

  if (x == 0)              // Branch A
    return;

  ...
}
```

Figure 4.2: Example function in sched.c from gcc.

were to set the GHR to a fixed value when the function call is made, the branch would be predicted correctly 99% of the time, again assuming no aliasing. In this particular case the Filter predictor [CEP96] would also solve this problem, but in the more general case (e.g., if Branch A were not completely biased, but did follow a pattern), it would not.

## 4.2 Background and Related Work

The Global History Register, first proposed by Yeh and Patt [YP93], is a special case of their two-level adaptive branch predictor, with per-branch pattern history being collapsed into one global history. The benefits of the GHR were further demonstrated by McFarling [McF93] with his *gselect* and *gshare* predictors. Branch prediction research has continued to use global history for branch prediction accuracy.

Figure 4.3 provides a diagram of the basic layout of a *gshare* predictor. The current branch address (program counter) is xor'd with the contents of the GHR to produce an index into the Pattern History Table (PHT). Each entry in that table contains a 2-bit prediction for that branch where the 2-bit values of $00_2$ and $01_2$ represent predict *Not Taken* and the values $10_2$ and $11_2$ represent predict *Taken*. When a new branch instruction is encountered, the branch address and GHR are xor'd to produce an index. The entry at that index is obtained and a prediction

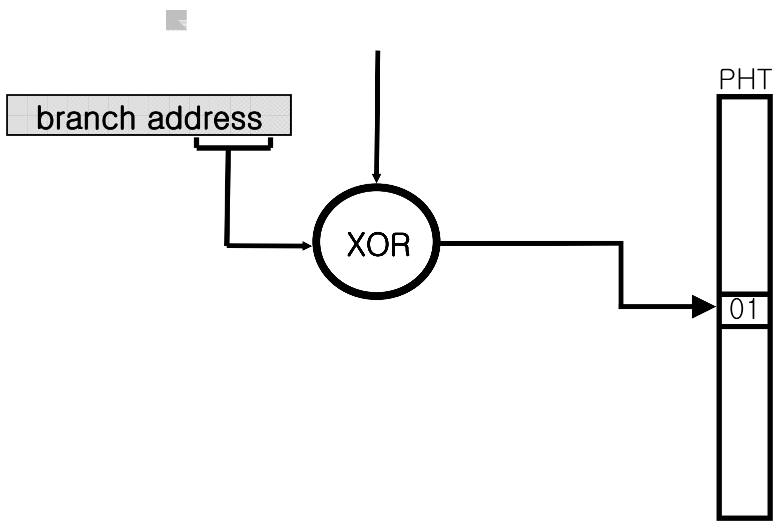


Figure 4.3: Diagram of the *gshare* branch predictor.

based on that entry is produced. When the branch outcome is later resolved, the entry at that same index is incremented for *Taken* and decremented for *Not Taken*. As a result, a prediction is based on the previous behavior of this combination of branch and history. Unfortunately, because the PHT is finite, only a subset of the branch address is used. This can result in aliasing (multiple addresses and histories accessing the same table entry). Many predictors described in the remainder of this section attempt to reduce or eliminate this aliasing.

The success of the *gshare* predictor and demonstrated value of global history has led to the exploitation of increasingly long global histories. Longer histories aid in establishing correlation with more distant branches as well as (in some cases) reducing aliasing in indexed predictors. The Alpha 21264 predictor [Kes99] uses 12 bits of global history to index its global history and choice tables. The 2Bc-gskew predictor [SM99] uses 21 bits of global history to index its three prediction tables. The perceptron predictor [JL02] uses 34 bits of global history to train a simple neural network. More recent branch predictors use even longer histories to improve performance. The O-GEHL predictor [Sez05] exploits history lengths ranging from 100-200 bits. The PPM predictor [CCM96] detects closest matching patterns in very long history lengths. The L-Tage [Sez07] predictor uses history lengths varying from 0 to 640 bits. However, the complexity of some of these predictors have deterred their adoption in modern processors [Loh05]. L-Tage is of particular interest to our work in that its ability to dynamically use different history lengths likely benefits from the phenomenon we identify.

Another trend leverages the property of branch bias. The bimodal [Smi98], bi-mode [LCM97], YAGS [EM98], and Filter [CEP96] predictors dynamically exploit the bias of a large percentage of branches. The bi-mode predictor separates branches, predicting those with a taken bias using a different table than those with a not-taken bias. The YAGS predictor maintains a similar table of biases indexed by the program counter (PC) and two gshare tagged caches which store the branches whose behavior is contrary to the bias. The Filter predictor [CEP96] uses the BTB to identify highly biased branches. Those branches are then excluded from the dynamic gshare prediction thus reducing the amount of aliasing. One benefit of our technique is that, like these predictors, it allows the prediction of some biased branches with minimal resource utilization; however, this is only part of the benefit, as evidenced by the fact that we improve even the performance of the Filter predictor, which has already eliminated the biased branches from the tables. A potential advantage of our technique when compared against these predictors which learn branch bias dynamically, is that our technique requires no such learning time.

The seminal work of Pan et al. [PSR92] investigates the benefits of global branch correlation. Evers et al. [EPP98] continue the investigation into branch correlation and recognize that while many branches are highly correlated with a small number of prior branches, some are not. This phenomenon - exploited by others in the work discussed in this section - is critical to this work. Additionally, the work of Thomas et al. [TFWS03] similarly identifies branches which are correlated and those which are not. Their technique removes non-correlated branches on a branch-by-branch basis. Recent work by Sazeides et al. [SMCK08] demonstrates that selecting the subset of history that is most highly correlated with a given branch can improve predictor accuracy.

Gao and Sair [GS05] target function entry and return as a point where correlation may diminish. A similar approach is taken in “Path-Based Next Trace

Prediction" by Jacobson et al. [JRS97] of discarding some of the irrelevant history from within a subroutine and after a subroutine return by using a Return History Stack. Our work similarly addresses entry and return but does so with a different mechanism. Their work attempts to save the GHR at entry and restore at return whereas our work does not require saving any copies of the GHR. The Frankenpredictor [Loh04] also targets call and return by shifting in masks depending on the instruction opcode. Their work likely benefits from the phenomenon we identify in this work.

The notion of removing useless bits from history is not entirely novel. The perceptron predictor, by the nature of its neural network, attempts to do exactly that. "Dynamic history-length fitting" [JSN98] directly tries to cut history to the desired length based on trial and error rather than using heuristics as we recommend. The Elastic History Buffer [TTG97] and Variable Length Path Branch Prediction [SEP98] both propose allowing branches to specify how much history will be used. Our proposal differs from this technique in that it works entirely using simple heuristics and because the modifications made to the GHR in our technique are not unique to a single branch but rather affect all subsequent branches.

Choi et al. [CPT08] propose modifying the GHR during thread migration between cores on a CMP featuring speculative multithreading (to provide a useful GHR to begin thread execution). For some benchmarks, they find that inserting the PC of the thread spawning instruction during thread creation can provide slightly better accuracy than providing an oracle-based correct GHR. This effect is likely related to the regions of limited branch correlation targeted by our work. Our work differs in that it improves branch prediction even for single-threaded execution.

## 4.3 Resetting the GHR

In this section, we discuss the particular hardware techniques we examine in this chapter. In addition, we will discuss some potential software/hardware techniques based on ISA modification and profile analysis. However, the latter is primarily only interesting in that it motivates the much simpler hardware-only techniques.

The goal of this section and the next is to identify points in the program control flow with little correlation to prior branches. These Regions of Limited Branch Correlation (RLBCs) may benefit from artificial modifications to the GHR.

### 4.3.1 Hardware RLBC Identification

Existing control flow constructs provide hints for finding these RLBCs. Loops and function calls often represent breaks in control flow. The not-taken path following a backward branch often indicates a loop exit, and we typically expect branches following the loop to be less correlated with branches inside the loop. Similarly, branches in a function call may not be correlated with the branches preceding the call. Finally, branches following a return from a procedure may lack correlation with the branches in the procedure. When these regions are detected–by the execution of the applicable instruction–we can perform modifications to the GHR to improve accuracy.

As mentioned before, when entering an RLBC, the GHR contains noise. To eliminate this noise, we could zero the GHR but this may cause potential problems for index-based predictors. However, if we were to zero the GHR, index-based predictors like gshare would bias training toward one particular region of the predictor. Therefore, in resetting the GHR we use the same technique as [CPT08]. They generate a GHR from the program counter, in their case to manufacture a GHR when forking a speculative multithreading thread, for which the correct GHR is unknowable. Using the PC provides a unique history for each point at which we reset the GHR (eliminating aliasing between the different resetting points), and ensures that when we return to this code, we will reset to the same value.

For not-taken backward branches, the value inserted into the GHR is the PC of the backward branch. For function calls, the PC of the calling instruction is inserted into the GHR. Finally, for function returns, the PC of the return instruction is inserted into the GHR.

For the hardware-only techniques, we assume a very simple decision–we either always reset the GHR, or not. In which cases we want to reset the GHR (e.g., on backwards branches and calls, but not returns) depends on the specific branch prediction hardware (specifically, which branch predictor and what size) we are modifying. For example, the higher the incidence of branch aliasing, the more aggressive we will want to be. For a wide variety of predictors, we will identify the best combination of these three resetting points.

Modifications to the GHR will typically take place in the fetch pipeline stage, just like any other modification. For example, if a backwards branch is predicted not taken, we will modify the GHR as described, but checkpoint the old GHR as on any other speculative branch. If the branch is mispredicted, we restore the GHR. If the branch was originally predicted taken, we update the GHR as normal, and only apply our technique if the branch is resolved mispredicted. In either case, the GHR is modified in the same places as other branches.

Results for these approaches are discussed in Section 4.5.

### 4.3.2 Static Analysis of RLBC Points

We also examine profile-based identification of the best points to modify the GHR (and even allow more flexible modification, such as only clearing regions of the GHR). However, the static techniques are somewhat problematic for several reasons, and ultimately provide little gain over our hardware-only techniques.

These issues include:

1. it requires ISA modification to indicate the modification point and possibly how the GHR should be modified

2. it requires expensive profiling

3. it requires binary compatibility for different architectures

4. it requires software know the exact details of the branch predictor, and some manufacturers have been extremely protective of those details

However, these results are still interesting as a comparison with the hardware-only techniques. But we omit many details of the profile-driven analysis, since the

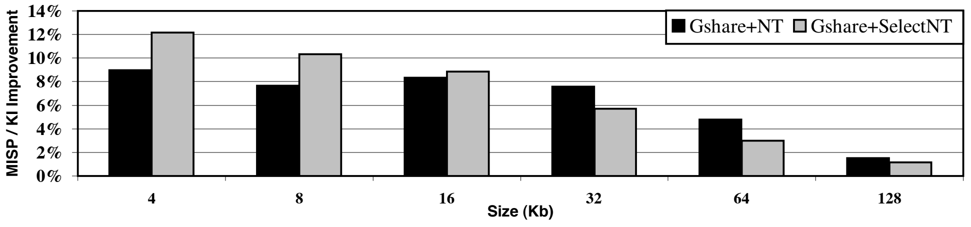


Figure 4.4: MISP/KI improvement from targeting all backward branches or those selected by profiling.

results are primarily useful as a point of comparison.

We created a branch trace of each benchmark, and for that trace recorded the expected result (branch predicted correctly or incorrectly) assuming a history length of any given value below a certain maximum. Prediction accuracy assumed a correlating predictor for each branch with no aliasing. Accounting accurately for aliasing at this stage in the profiler would have made the subsequent steps prohibitively expensive.

We could identify a "good" place to dynamically reset the GHR, by identifying a place in the trace where the next branch predicted well with zero bits of history, the subsequent branch with 1 bit of history, the next with 2 bits, etc. By calculating all such places (a single possible location would be following a possible static branch, in either the taken or not-taken case–we could make different decisions for each), we select the best. Because the different resetting points will interact, we need to start the analysis anew after one is chosen to select the next. We continue the process until we reach a minimum threshold of marginal improvement. Interestingly, the optimal minimum threshold was actually a negative improvement. This is because the effect of aliasing makes the gains higher than the trace predicts–so we need to make it more aggressive than it would otherwise be.

Of the four types of conditional branches (forward taken and not-taken, backwards taken and not-taken), we quickly learned that the first three are almost always better left alone, and the last are usually best modified. Therefore, our analysis technique worked best if we just forced it to ignore all but backwards-NT, and just decide which of those to modify the GHR.

Figure 4.4 provides the best parameterized static result, compared to our simplest technique which blindly modifies the GHR at all backwards NT branches. These results are based on a gshare predictor using a maximum number of history bits executing select SPECint benchmarks. While we see that for small predictor sizes, the static analysis does indeed do a better job of selecting branches than the hardware-only technique, it does not seem to be enough of a gain to account for the concerns over this technique. But this does indicate there is room for future investigation in this direction.

## 4.4 Methodology

This section describes our simulation framework and benchmark selection. We modified a version of SMTSIM [Tul96] to implement our array of branch predictors. SMTSIM executes unmodified Alpha ISA code and supports out-of-order SMT or CMP processors. In this study, SMTSIM was executing a single threaded binary on a single core.

We also modified the framework provided by the Championship Branch Prediction Competition to execute the array of branch predictors via trace-based simulation.

The SPEC results, then, come from detailed simulation of the predictors running on a modern core, and includes, for example, the effects of delayed updates to the predictor. The details of the simulated core are not particularly important, however, as we only produce branch mispredict rates in this work. This is because the CBP results only allow trace-based simulation of mispredict rates and no direct performance results.

### 4.4.1 Branch Predictors

Gshare [McF93] is a standard implementation with varying size prediction tables. Filter [CEP96] is implemented using a three bit saturating counter and one bias bit per BTB entry. After eight sequential executions with the same outcome, the counter becomes saturated. Predictions for branches with a BTB

Table 4.1: Characteristics of 2Bc-gskew predictor

| | BIM | G0 | G1 | META |
|---|---|---|---|---|
| prediction table | 16K | 64K | 64K | 64K |
| history length | 4 | 13 | 21 | 15 |

saturated counter are given as the bias. If the bias prediction is correct, no update is performed. All other branches are predicted using a gshare predictor. The BTB assumed has 512 entries with 4 way associativity. The additional BTB hardware required by filter is not counted in the hardware budget for the filter predictor. Our default size for gshare and filter is 32Kb.

The Alloyed (Global/Local) Perceptron is a 32Kb implementation [JL02]. The Alloyed Perceptron has a 91 entry table and uses 34 bits of global and 10 bits of local history.

Our 2Bc-gskew predictor [SM99] is a 416Kb implementation with four prediction tables. Three of these tables are indexed with the GHR. The fourth is a meta predictor which chooses between the results produced by the 2-gskew predictor (two of the GHR indexed tables) and the bimodal predictor. We provided each entry with its own hysteresis bit–we did not simulate the space optimization of shared hysteresis bits. More details of the 2Bc-gskew predictor are given in Table 4.1. We did not attempt to create a reduced version of this predictor (similar in size to our other predictors). That predictor was carefully tuned in [SM99] for this size; additionally, this allows us to demonstrate that our technique is effective even on a very large branch predictor.

Our implementation of the Alpha 21264 predictor [Kes99] is 29Kb with a 4k entry choice prediction table, 4k entry global prediction table, a 1k entry local history table with 10 bits of history per entry, and a 1k entry local prediction table.

BTB misses are faithfully modeled and we assume branch instructions which miss in the BTB are repredicted when identified by decoding. Similar to [PE00] we use the number of conditional branch mispredictions per thousand instructions executed (MISP/KI) as our primary metric.

### 4.4.2 History Tuning

For each of our predictors, we evaluated all possible history lengths to ensure the strongest baselines performance. To determine the optimal history length we averaged the average MISP/KI for SPECint, SPECfp, and CBP. Most predictors benefited from the maximum available history lengths. However, some predictors, especially our smallest sizes of gshare and filter, benefited from low amounts of history. In fact, the 4Kb implementation of gshare uses only 2 bits of history.

### 4.4.3 Benchmark Selection

We choose eight benchmarks from the SPEC2000 suite. We intentionally select eight programs that are sensitive to the (overall) branch prediction accuracy in our simulation framework. We do this by filtering out those programs whose performance improved by less than 3% when a perfect branch predictor was introduced. We simulate 100 million instructions starting at a single execution Simpoint [SPHC02].

We also use the traces provided for the Championship Branch Prediction Competition.

## 4.5 Results

In the previous sections we have described a hardware technique which uses heuristics to perform for GHR modification (GHRM). For not-taken backward branches, the value inserted into the GHR is the PC of the backward branch. For function calls, the PC of the calling instruction is inserted into the GHR. Finally, for function returns, the PC of the return instruction is inserted into the GHR.

The GHRM techniques will likely have different impacts for different branch predictor sizes. For realistic predictor sizes, we compare the accuracy of GHRM when applied to simple predictors against the accuracy of more advanced and larger predictors. Lastly, we evaluate the effectiveness of GHRM when applied to these more advanced predictors.

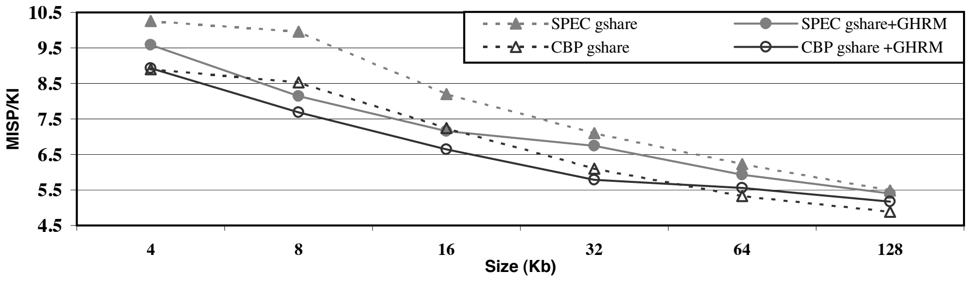


Figure 4.5: MISP/KI for different size gshare predictors for select SPEC2K benchmarks and CBP traces.

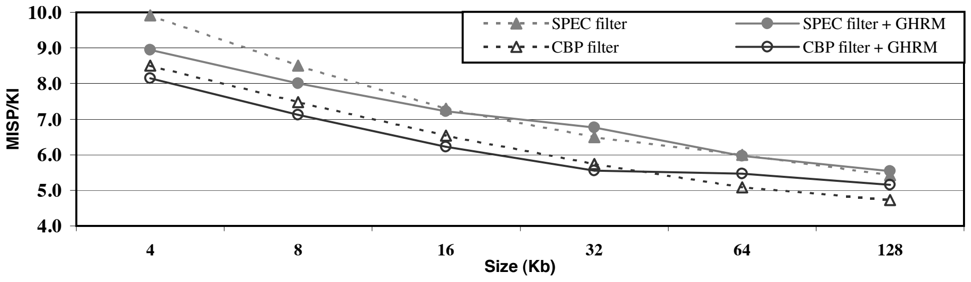


Figure 4.6: MISP/KI for different size filter predictors for select SPEC2K benchmarks and CBP traces.

### 4.5.1 Predictor Size

In general, the benefits of these techniques are most substantial when working with smaller predictors. This is because aliasing is most extensive in smaller predictors, and reducing aliasing is the most important result of our modifications. However, we also show that these approaches are still applicable to larger predictors. In particular, the 2Bc-gskew predictor is discussed in section 4.5.3.

We start by examining the gshare and Filter predictors initially. We do this because these are simple, highly effective predictors, and they are easily parameterized by size. The results for varying gshare and Filter sizes are contained in Figure 4.5 and Figure 4.6 respectively. The MISP/KI for each baseline predictor is provided for both the select SPEC2K benchmarks and the CBP traces.

The percentage reductions in MISP/KI are provided in Table 4.2. The results are highest for the small predictors, and actually go negative when the tables get large. There are still gains for individual branches, but this is where our simpler hardware-only technique breaks down because it cannot distinguish

Table 4.2: MISP/KI reductions per size for gshare+GHRM when compared against standard gshare and filter+GHRM compared against standard filter for both the select SPEC2K benchmarks and CBP traces.

| | Size (Kb) | | | | | |
|---|---|---|---|---|---|---|
| Predictor | 4 | 8 | 16 | 32 | 64 | 128 |
| SPEC gshare+GHRM | 6.5% | 18.2% | 12.7% | 5.0% | 4.8 % | 1.5% |
| CBP gshare+GHRM | -0.5% | 9.9% | 8.1% | 5.3% | -4.2% | -5.7% |
| SPEC filter+GHRM | 9.8% | 6.0% | 1.1% | -4.3% | 0.6% | -1.9% |
| CBP filter+GHRM | 4.1% | 4.9% | 4.7% | 3.1% | -7.4% | -8.8% |

Table 4.3: Gshare and filter heuristic configurations: either function calls (FC), return (RTR) instructions, and/or not-taken backward branches (NT-BB) can trigger GHR modifications.

| | Gshare | | | Filter | | |
|---|---|---|---|---|---|---|
| Size (Kb) | FC | RTR | NT-BB | FC | RTR | NT -BB |
| 4 | T | T | T | T | F | T |
| 8 | T | F | T | T | F | F |
| 16 | T | F | T | T | F | F |
| 32 | T | F | F | T | F | F |
| 64 | F | F | T | F | F | T |
| 128 | F | F | T | F | F | T |
| 256 | F | F | T | F | F | T |

between instances–in this region, approaches like our static technique may become more attractive.

Our technique (GHRM) uses the best configuration of heuristics (which specific points to modify the GHR) for each predictor type and size. The actual configurations used are contained in Table 4.3. We see in this table that as predictors get larger, and the effect of aliasing is reduced, we tend to get less aggressive; for example, resetting the GHR on returns tends to only be beneficial with the smallest predictors.

It should be noted that the negative results are primarily the effects of the variety of compilation systems. For example, if we could optimize for SPEC alone, we get positive results for this entire range. The same is true for the CBP results. These differences in program behavior are likely related to the CBP benchmarks

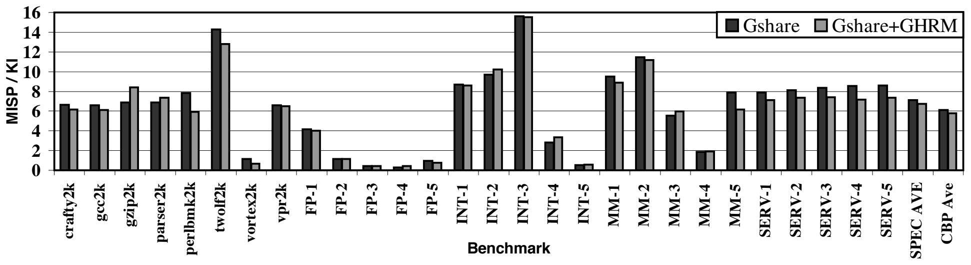


Figure 4.7: Baseline and improved 32Kb gshare for each benchmark. (Lower bars indicate better accuracy.)

being generated differently than our SPEC binaries. As such, trying to find a single best configuration for both resulted in significantly lowered results in many cases. This implies that if the compilation systems are universally aware of what the hardware is doing, and at least do not generate code that is at cross purposes, the potential gain from these techniques can be much higher than shown here. It may also indicate that a simple dynamic technique that chose (at a coarse granularity) which of our eight configuration combinations was most effective could also provide good results.

From Figure 4.5, we can see that for some of the smaller sizes of gshare, gshare+GHRM enables branch prediction accuracies similar to predictors of twice the size. Filter (Figure 4.6), because of its ability to filter out highly biased branches, benefits less from these techniques but still shows noticeable improvements at smaller sizes. These results validate the assertion that in some cases, our simple and nearly cost-free branch prediction modification can provide the same performance with a significant decrease in predictor size.

Figure 4.7 provides the MISP/KI for each of the selected benchmarks. For the select SPEC CPU2000 benchmarks, almost all benchmarks benefit from applying GHRM. For the CBP traces, applying GHRM has a positive benefit on the majority of traces, most notably the server traces.

### 4.5.2 Heuristic Configuration

In the previous section, different heuristics were said to be effective for the same predictors given different benchmark sets. The most notable difference is

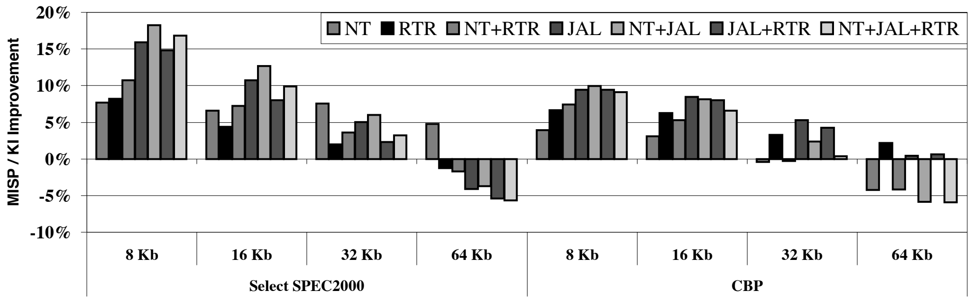


Figure 4.8: MISP/KI improvement for different heuristic configurations for varying sizes of gshare.

that the CBP traces benefit more from function call modification than the select SPEC2K benchmarks. This likely comes primarily from the difference in code generation.

Although one configuration may offer the best performance for each predictor of a given size, often other heuristics offer competitive performance. Figure 4.8 provides the MISP/KI improvement for both working sets given each configuration using different sizes of gshare. Different sizes of gshare are provided up to 64Kb where the utility of the configurations drops (as shown in Figure 4.5). This figure shows us that while some configurations may provide the best performance, many still provide reasonable benefits.

### 4.5.3 Other Predictors

The primary goal of these techniques is to aid simpler branch predictors. However, these techniques can benefit other predictors as well. Even in the absence of significant aliasing, this mechanism allows faster training (when the branch working set changes) and retraining (when branch bias or patterns change). Figure 4.9 provides the percentage reduction in MISP/KI for a larger set of predictors, some but not all of similar size. The GHRM configuration selected for each of these predictors is provided in Table 4.4.

All predictors benefit from these techniques. The Alloyed Perceptron has the largest MISP/KI reduction, 14.4% for the select SPEC2K benchmarks and 10.0% for the CBP traces. The Alloyed Perceptron predictor results are partic-

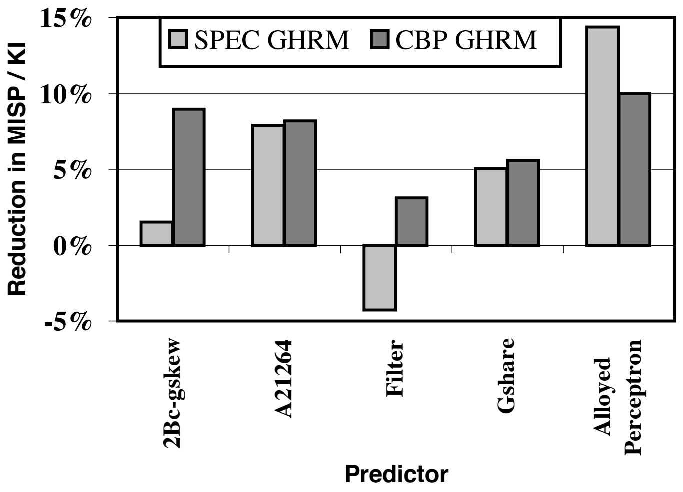


Figure 4.9: Reduction in MISP/KI for various predictors. Gshare and filter are 32Kb implementations.

Table 4.4: The best heuristic configuration for each predictor.

| | Configuration | | |
|---|---|---|---|
| Predictor | FC | RTR | NT-BB |
| 2Bc-gskew | T | F | F |
| A21264 | T | F | T |
| Filter | T | F | F |
| Gshare | T | F | F |
| Alloyed Perceptron | T | F | T |

ularly interesting. Perceptron predictors are specifically targeted at reducing the impact of non-correlated branches, and are largely successful at doing so. But it removes the effect of the noise only probabilistically, and therefore some of the noise always remains. We find that even in the context of that predictor, we are able to further remove the impact of useless noise.

The other interesting results are for the 2Bc-gskew predictor. This result is in contrast to some of our earlier results that might indicate that this technique is only useful for small predictors. This predictor, with multiple large tables, and significant features to tolerate aliasing, still takes significant advantage of our ability to identify non-correlated regions of code.

### 4.5.4 Non-Select Benchmarks

The prior results are shown for the select SPEC CPU2000 benchmarks. When averaging the benefit of our technique across all, not just select, SPEC CPU2000 benchmarks we found that 8Kb, 16Kb, 32Kb implementations of gshare achieved 9%, 6%, and 1% MISP/KI improvements. Our techniques were not beneficial for larger sizes of gshare. All the other predictors except 2Bc-gskew and 16Kb or greater implementations of filter also saw a reduction in MISP/KI.

SPECfp is not included in the select benchmarks and has some interesting characteristics. The IPC performance of these benchmarks was not found to be highly tied to the branch prediction accuracy. In addition, nearly all our branch predictors were able to predict SPECfp with high accuracy. In fact, the average MISP/KI for SPECfp is less than 1 for our 32Kb gshare implementation.

One particularly interesting result relates to a methodology common in recent branch prediction research. Following this precedent, as described in Section 4.4, we tuned all of our baseline predictors to find the optimal amount of GHR history to use. This methodology, though, has a tendency to over-tune the predictor for a small set of benchmarks (where a real predictor would be tuned for a much larger set). In fact, this methodology significantly reduced the magnitude of our overall gains, which were always higher when our predictor and the baseline used the same amount of history. In the case of the smallest predictor (4 Kb gshare), this over-tuning becomes very apparent – although we used all of SPEC to find the optimal length, the greater importance of branch prediction in the integer benchmarks created an optimal history length (2 bits!) which was an extremely poor choice for the FP benchmarks. So when applying our techniques, we see tremendous improvements in those benchmarks–a 47% reduction in average MISP/KI for SPECfp. But we did not choose to highlight these results in this chapter, because we feel they are more an artifact of the standard methodology.

But this does highlight an important advantage of our branch predictor: it enables the use of longer histories by eliminating the artifacts that create the pressure to use shorter history than that dictated by the size of the branch history tables. For these small predictors, the best tuned predictor using our optimizations consistently used more history length than a tuned predictor without our optimizations. If we had not followed this methodology, and instead assumed all predictors use the expected amount of history, our reported results would be higher. For example, again for the smallest gshare, our techniques provide a 21% reduction in mispredicts for SPEC-select and 10% reduction for CBP (as opposed to the 6.5% and -0.5%, respectively, reported in Table 4.2).

### 4.5.5 Result Summary

These results demonstrate that simple heuristics can be used for improved branch prediction accuracy at little cost by reducing the amount of noise in the global path history. For most predictors, modifying the GHR when a backward branch is not taken or when encountering a function call provides a reasonable reduction in MISP/KI.

In general, our results show that the technique of modifying the GHR is most useful for smaller predictors. However, the techniques are not limited to simple predictors and are shown to eliminate a significant percentage of MISP/KI in more complicated predictors.

## 4.6 Chapter Summary

This chapter demonstrates that artificially modifying the GHR before regions of limited branch correlation (RLBC) can improve branch predictor performance during single threaded execution.

By performing GHR modifications based on program heuristics, improved branch predictor accuracy can be achieved. These heuristics target loop exits, function calls, and function returns. For 32Kb predictors, our techniques offer up to a 12% overall decrease in MISP/KI. For small gshare predictors, these techniques can provide as much as a 14% reduction in MISP/KI. All predictors examined benefit from these techniques and only a minor hardware modification is required for implementation.

These techniques enable processors to achieve higher branch prediction accuracy, increased performance, and reduced power consumption with simple branch predictors. Simple branch predictors have the advantages of easier design and verification as well as lower hardware cost and lower power consumption. In addition, simpler branch predictors modified by our technique can provide solid single-thread performance while freeing resources for potential better use throughout the processor (more cores, better interconnect, larger shared caches, etc.).

## Acknowledgments

The work in this chapter was supported in part by the Reserve Officers Association Henry J. Reilly Memorial Scholarship, NSF Grant CCF-0702349, and Semiconductor Research Corporation Grant 2005-HJ-1313.

This chapter contains material from "Creating Artificial Global History to Improve Branch Prediction Accuracy", by Leo Porter and Dean M. Tullsen, which appears in *Proceedings of the Twenty-Third International Conference on Supercomputing*. The dissertation author was the primary investigator and author of this paper.

# Chapter 5

# Exploiting Hetereogeneity for Improved Throughput, Energy, and Fairness

Multi-core processors introduce the opportunity of having resources shared among multiple cores. Shared resources enable individual threads to consume more than their share when the resource is under-utilized. However, when shared resources are over-utilized they introduce a new challenge: resource contention. In this chapter we evaluate thread scheduling decisions to reduce such resource contention, specifically in the context of multi-core processors in high performance computing (HPC) systems.

In HPC systems, multi-core processors are ubiquitous. The utility of single chip multi-processors (CMPs) is in their potential for high thread throughput through the simultaneous execution of multiple threads per die. In executing multiple threads per die, hardware structures which were previously dedicated to a single thread are now shared among multiple threads. It is critical to understand and manage the interactions of these threads and their contention for shared resources as such contention impacts performance (execution time relative to single-threaded execution), fairness (do all threads suffer performance degradations equally), and energy efficiency (do the interactions of these threads result in poor energy consumption).

Symbiotic coscheduling was first proposed for Simultaneous Multithreaded Processors [TEE$^+$96, ST00]. Snavely and Tullsen [ST00] show that threads can cooperatively share microarchitectural features for improved performance. CMPs offer a similar problem, but the level of sharing is restricted, in many cases, to shared caches and off-chip bandwidth.

Scheduling for CMP processors has received heightened attention with proposals varying from user-informed scheduling [WS06a, WS06b] to techniques for guaranteed quality of service [GSYY09, KCS04, LLD$^+$08]. Our work is targeted for HPC, where a large number of threads commonly need scheduling and users are typically willing to submit characteristics of jobs for improved scheduling [BAG00]. In such a domain, the scheduler should have the freedom to group threads together on a single multi-core processor. How those threads interact has a large impact on performance, fairness, and energy. All three factors are critically important. With burgeoning power bills for large systems, energy efficiency has become a first-class concern [BBC$^+$08]. In addition, we recognize that for HPC environments where users are charged for system usage, fairness is a primary consideration [Jac, LGC97].

The aim of this chapter is to provide intuition for improved static scheduling decisions on existing multi-core HPC systems. To support this aim, we identify specific critical features that can be extracted through basic profiling and evaluate which are most critical to coscheduling decisions. We then examine a number of static scheduling policies and evaluate their effectiveness in multiple domains. On a system of Nehalem [KDMK08] processors we evaluate coscheduling decisions and their impact on performance and fairness. Under different scheduling conditions and on a system of dual-socketed Westmere [KBM$^+$10] processors, we also evaluate performance, fairness, and power.

The contributions of this chapter include:

- We introduce the cache hit rate vector (CHRV) as a highly effective means of workload characterization to improve thread scheduling. The CHRV is independent of specific processor features: only a single profile pass is required for processors with different memory hierarchies.
- We evaluate a number of policies related to workload characterization and demonstrate that a simple policy, based on process CHRVs, can provide improved scheduling decisions.
- In contrast to prior work, we identify Working Set Overflow (WSO) as a highly effective metric for job scheduling. This metric is shown to be more effective than traditionally used measures for predicting contention (L2 miss rates, working set size, etc.).
- We demonstrate that these policies are consistently effective across each of our scheduling environments on two different processors.

We show our policies predict coschedules with consistently high scaled throughput (99% of the best case), low energy delay product (99% of best case and 7% better than average), and significantly improved fairness (148% better than worst case, 48% better than average).

## 5.1 Related Work

Job scheduling has been a research topic for decades. Our work focuses on static job scheduling (assign threads to processors and do not move them) using process characteristics to exploit coschedule symbiosis.

Process characteristics have been used in prior work to aid scheduling decisions. Denning first proposed using process working sets to improve the performance of virtual memory [Den68]. Ghosal et al. perform static processor allocation on multiprocessor systems based on thread parallelism profiles [GST91].

Snavely and Tullsen first proposed the notion of symbiotic scheduling for SMT processors [ST00]. Weinberg and Snavely evaluate the potential of symbiotic workload space sharing on an HPC platform [WS06a] and the users' ability to accurately determine resource bottlenecks on that platform [WS06b]. Paired gang scheduling pairs I/O bound jobs with compute intensive jobs for better overall throughput in HPC [WF03]. Our work is complementary, in that previous works identify the utility of symbiotic coschedules, however their work stops at pairing two jobs together and does not address fairness or energy.

Shared levels of cache have long been recognized as a point of contention. Yan and Zhang aim to predict worst case run-times by using profiled control flow information to predict i-cache contention between two threads [YZ08]. Anderson et al. evaluated grouping processes by L2 miss rates to avoid coscheduling those with high miss rates, for real-time tasks on multicores [ACD06]. Fedorova et al. recognize the potential of using heuristics for better scheduling on SMT processors. They use predicted L2 miss rates based on reuse distance [BH04] to inform their scheduler on in-order, simulated, SMT processors [FSSN05].

Cache partitioning for Quality of Service and fairness has been evaluated for SMT processors [CKS$^+$04] as well as CMPs [GSYY09, KCS04, LLD$^+$08]. Hardware support for QoS or Fairness has limitations — namely higher expense, less flexibility, and longer time to market, and proposed software solutions often require dedicated time slicing to ensure disadvantaged threads make fair progress on existing systems [FSS07].

## 5.2 Methodology

This section describes our experimental framework and benchmark selection for the work in this chapter. Our experiments were performed on the Gordon cluster [HJG$^+$10, NS10]. We used both standard nodes with dual socketed Xeon E5530 (Nehalem) processors and I/O nodes with Xeon X5650 (Westmere) processors. Characteristics of each processor are contained in Table 5.1.

SPEC2006 benchmarks were selected for their wide range of HPC applications. Eight of these benchmarks (*perlbench*, *gcc*, *dealII*, *povray*, *omnetpp*, *cactusADM*,*tonto*, and *sphinx3*) were excluded due to limitations of the instrumentation framework, leaving twenty-one benchmarks for our analysis. The rest in-

Table 5.1: Architectural Features

| Features | Nehalem | Westmere |
|---|---|---|
| Cores | 4 | 6 |
| Clock Rate (GHz) | 2.4 | 2.67 |
| L1 cache | 32 KB private | 32 KB private |
| L2 cache | 256 KB private | 256 KB private |
| L3 cache | 8 MB shared | 12 MB shared |

clude benchmarks drawn from the scientific domains of Fluid Dynamics, Quantum Chemistry, Molecular Dynamics, Quantum Computing, Finite Element Analysis, Linear Optimization, Structural Mechanics, and Weather. Table 5.2 contains basic details from [SPE].

To obtain cache statistics per benchmark, we used the PEBIL [LTCS10] static instrumentation tool to instrument the binaries and pass the data addresses to the PEBIL cache simulator. Profiling was performed on reference inputs as we expect users to have a solid knowledge of inputs which are representative of their working set.

To evaluate effective coschedules, we looked at a number of key metrics. To evaluate throughput, we used scaled throughput (or weighted speedup, weighted by single threaded performance) from [ST00] shown in Equation 5.1.

$$ST_{co} = \sum_{i=0}^{n} \frac{S_i}{M_i} \tag{5.1}$$

In this equation, $S_i$ is the runtime of benchmark $i$ when run alone and $M_i$ is the runtime of benchmark $i$ when coscheduled. $n$ is the number of processes coscheduled. Efficient coschedules maximize this metric.

To evaluate energy consumption when running a fixed number of jobs (Fixed Jobs), we use Energy Delay Product (EDP) from Equation 5.2.

$$EDP = P_{ave} * T^2 \tag{5.2}$$

$P_{ave}$ is the average power consumed over execution time $T$. The more energy efficient a coschedule, the lower EDP.

Fairness was evaluated following the convention of Kim et al. [KCS04] as

Table 5.2: Related field for selected SPEC2006 benchmarks

| Benchmark | Field |
|---|---|
| astar | Artificial Intelligence |
| bwaves | Computational Fluid Dynamics |
| bzip2 | Compression |
| calculix | Structural Mechanics |
| gamess | Quantum Chemical Computations |
| GemsFDTD | Computational Electromagnetics |
| gobmk | Artificial Intelligence |
| gromacs | Chemistry-Molecular Dynamics |
| h264ref | Video Compression |
| hmmer | Computational Biology |
| lbm | Computational Fluid Dynamics |
| leslie3d | Computational Fluid Dynamics |
| libquantum | Physics |
| mcf | Combinatorial optimization |
| milc | Physics/Quantum Chromodynamics |
| namd | Structural Biology |
| sjeng | Artificial Intelligence |
| soplex | Linear Program Solver |
| wrf | Weather Forecasting |
| xalancbmk | XML Transformation |
| zeusmp | Physics/Magneto-hydrodynamics |

shown in Equation 5.3

$$Fairness = \sum_{i=0}^{n}\sum_{j=0}^{n}\left|\frac{S_i}{M_i} - \frac{S_j}{M_j}\right| \tag{5.3}$$

For a given number of coscheduled threads ($n$), this represents the difference in their weighted speedup. For a completely fair coschedule (all threads are slowed down by exactly the same factor), this sum would be zero.

We evaluate coschedules on the Nehalem nodes for scaled throughput and fairness by randomly selecting eight SPEC2006 benchmarks and partitioning them into two sets of four coschedules. Each four-thread coschedule results in those four threads running together on the quad core processor. The other partition of jobs run together at a different time or on a different processor. When individual benchmarks complete, they are re-started. This allows us to examine the

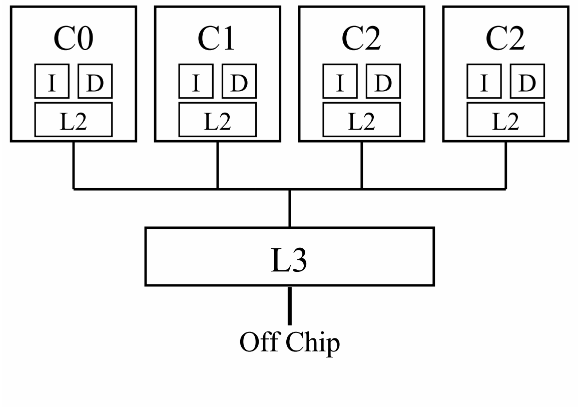


Figure 5.1: Cache Layout on a Nehalem Processor. Westmere Processors have similar layout with two additional cores.

interaction of the threads without noise from tail effects (when not all jobs have finished) or from particular phase behaviors between threads [SPHC02], and models throughput-oriented workloads well. To additionally reduce the noise, benchmarks and their data are copied to the local solid state drive and are statically compiled to avoid communication with shared libraries.

The evaluation of coschedules on the Westmere follows a similar methodology. We run two studies, one in which we fix the amount of time, as in the Nehalem study (Fixed Time), and the other in which we execute each job three times and ran until completion (Fixed Jobs). Unlike the Nehalem study, we randomly selected twelve benchmarks and then selected coschedules of six jobs on one socket and six jobs on the other socket. In addition to observing the performance of these coschedules, we also use a “Watts Up” [Wat] device to measure power and energy consumption. Our limited number of power measurement devices limited the number of coschedules we could evaluate.

In addition to the architectural features (clock rates, L3 cache size, etc.) that differ between the Nehalem and Westmere seen in Table 5.1, these studies also diverge in one key detail. In our study on the Nehalem processors, each coschedule of four threads runs alone (i.e. had dedicated off-socket bandwidth) thus four threads shared off socket bandwidth, memory, and solid state drive (SSD), time-sharing the processor. In the Westmere processor, both coschedules of six threads run together (six on one socket, six on the other) thus causing all twelve threads to share the same off-socket bandwidth and resources.

## 5.3 Motivation

When jobs are co-located on a CMP, most of the interaction between threads occurs due to sharing of memory resources. As shown in Figure 5.1, the interaction between the coscheduled threads include contention for last level cache (L3), the shared bus between private L2 caches and the L3, and the off-chip memory bandwidth.

The cross-thread interference experienced by one thread is the product of two factors:

1. The portion of shared resources occupied by co-scheduled threads.
2. The sensitivity of this application to other threads' utilization of those resources.

A thread can be sensitive to resource occupancy even if it is not actively using the resource. For example, although a thread may be content with the data held in its own private caches, the inclusion property of the caches implies that when lines are evicted from the L3 by a competing application, those lines are evicted from the private caches as well. Thus, even smaller applications may be sensitive to the behavior of other threads. To predict these behaviors, we need to know more about the behavior of each benchmark.

Figure 5.2 demonstrates benchmark characteristics in the cache depth format [WS08], simulating a series of possible cache sizes. These highly varying behaviors impact the combined behavior of coscheduled threads. For example, *namd* achieves a high hit rate with a relatively small cache (128KB) and larger cache sizes provide little improvement. In contrast, a benchmark like *libquantum* plateaus early, experiences significant gains when provided with about 16MB of cache, and again plateaus. Lastly, benchmarks like *mcf* and *lbm* benefit from nearly any increase in cache size.

The values in Figure 5.2 for a single benchmark represents the Cache Hit Rate Vector (CHRV). To obtain the CHRV, L1 caches of $2^n$KB sizes (for $0 \leq n \leq$

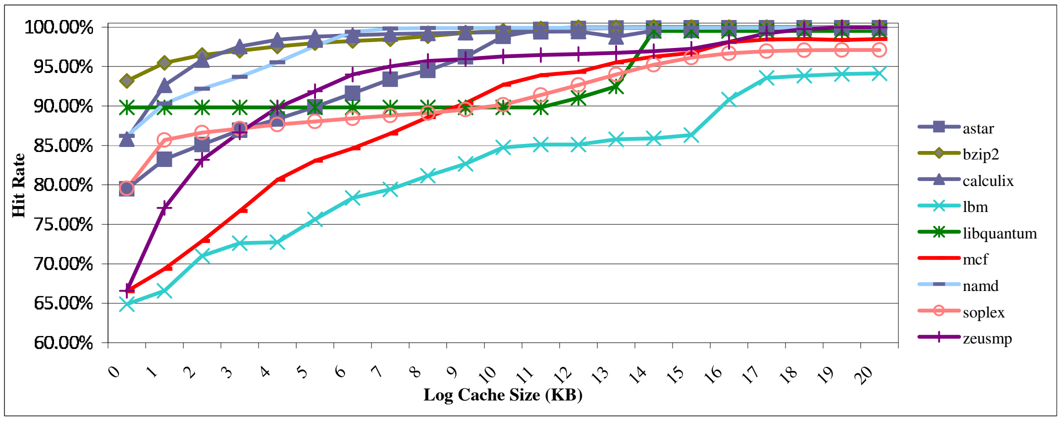


Figure 5.2: Cache hit rates for varying sizes of caches.

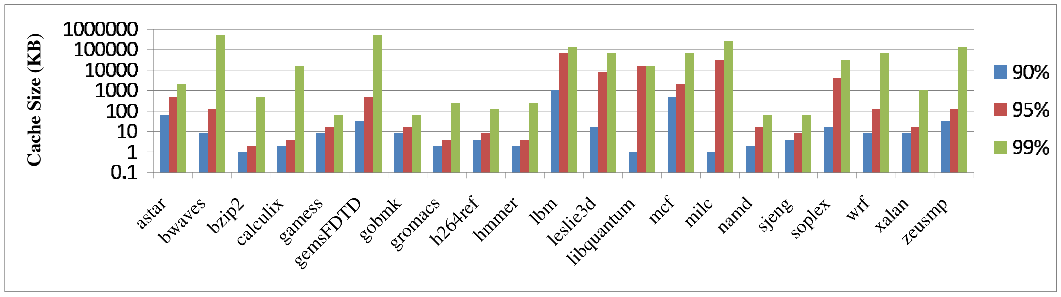


Figure 5.3: Working set sizes (90%, 95%, and 99%) per benchmark

20) are simulated. In terms of simulation delay, the additional cost of simulating multiple caches (rather than one cache) is relatively small compared with the overhead for instrumented code. This vector plays a key role in our prediction policies in Section 5.5.

To address these different behaviors, we also evaluate the classification of working set size for the application suite. We define working set size as the size of cache which achieves a threshold percentage of the hit rate (HR) of a near infinite cache. Figure 5.3 contains the working set size determined using 90%, 95%, and 99% as the threshold, relative to our near infinite cache (1GB). These metrics show a similar variance between benchmarks. For example, *gamess* requires a small cache to achieve a 90% HR and only requires slightly larger caches to achieve 95% and 99% HRs. Conversely, some benchmarks, like *bzip2* and *calculix*, require fairly small caches for 90% and 95% hit rates, but require much larger caches for a 99%

HR. Lastly, some benchmarks like *lbm* and *mcf* require large caches for even a 90% HR.

## 5.4 Exploiting Heterogeneity

SPMD (Single Program, Multiple Data) is a relatively dominant paradigm in HPC parallel computing. In this model, generated threads are highly homogeneous. However, prior work in Paired Gang Scheduling has already demonstrated the value of heterogeneity [WF03]. In Paired Gang Scheduling, I/O intensive threads are paired with compute intensive threads for better overall throughput. This can be accomplished, even if a single application does not support diverse threads, because a supercomputer installation is typically serving multiple diverse users and computations at once. Traditionally, those applications have been isolated into different partitions of the machine; however, scheduling diverse applications in close proximity affords greater execution efficiency [WCC$^+$07].

When highly homogeneous threads are co-scheduled onto a chip multiprocessor, resource utilization will almost always be unbalanced. Each thread will generally have a bottleneck resource. If the threads are completely homogeneous, all co-scheduled threads will stress the same bottleneck resource.

To demonstrate the value of heterogeneity in jobs, we created 50 runs consisting of four copies each of two random benchmarks. We then ran them on a Nehalem processor in coschedules (a coschedule is a partition of the eight jobs into co-executing sets of four). As a result, each coschedule is either purely homogeneous (four copies of the same benchmark), partially-heterogeneous (3 copies of one benchmark run with one of the other), or heterogeneous (2 copies of each benchmark run together). Scaled throughput varies significantly in some mixes. For those job mixes where scaled throughput was greater than ten percent different between the best and worst coschedule, mixed coshedules are *always* the correct choice. For fairness, of all 50 coschedules, the homogeneous workload was only the correct choice once and it only offered a 3% improvement over heterogeneous. Table 5.3 provides average scaled throughput and fairness over the 50

Table 5.3: Scaled Throughput and Fairness for job mixes which are homogeneous, semi-heterogeneous, and heterogeneous. Values are averaged over all 50 random job mixes. Higher values are better for Scaled Throughput. Lower values are better for Fairness.

| | Homogeneous | Semi-Heterogeneous | Heterogeneous |
|---|---|---|---|
| Scaled Throughput | 0.767 | 0.801 | 0.810 |
| Fairness | 10.29 | 7.03 | 3.48 |

job combinations. The improvements in both throughput (6%) and fairness (60%) demonstrate the value of heterogeneity.

From Paired Gang Scheduling and our basic study, the solution to the problem of scheduling threads on a dual core processor is relatively clear: co-schedule heterogeneous threads. However, when scheduling for modern four-core and six-core processors, heterogeneity is still essential; however, how to coschedule those heterogeneous threads while preserving fairness is much less clear.

One advantage of scheduling jobs into isolated, internally-homogeneous partitions is that it is generally considered fair (slowdowns may be variable, but you have only your own threads to blame). With heterogeneous co-scheduling (where your jobs are potentially competing with those of another user), users will desire much tighter guarantees of fairness. In this scenario, we observe unfairness when one thread takes more than its fair share of the resources, which happens to be very likely if the threads are diverse. We also find that unfair schedules tend to have the added disadvantage of being energy inefficient — among other things, they can create long tail effects (jobs that cannot make good forward progress until other jobs complete) which prevent the processor from fully exercising efficient power states. Each of these issues are addressed in the following sections.

## 5.5 Policies for Coschedule Prediction

Although the cache/memory performance of each application is clearly critical to understanding and improving scheduling, it is not clear exactly what measurable characteristics of memory behavior are best correlated with our two factors of (1) resource occupancy, and especially (2) sensitivity to resource occupancy.

Table 5.4: Metric Correlations with Scaled Throughput and Fairness

| Metric | ST | Fairness |
|---|---|---|
| WS95 | -0.494 | 0.456 |
| WS99 | -0.619 | 0.452 |
| L2MR | -0.675 | 0.387 |
| L3MR | -0.479 | 0.301 |
| L2MRSec | -0.302 | 0.094 |
| L3MRSec | -0.325 | 0.115 |
| L2WSO | -0.749 | 0.530 |
| L3WSO | -0.803 | 0.608 |
| WS95-L2WSO | -0.548 | 0.454 |
| WS99-L2WSO | -0.800 | 0.495 |
| WSO-Combo | -0.809 | 0.593 |

Therefore, we profile each benchmark and collect a large set of potentially useful characteristics, including the cache hit rate vector (CHRV) described in Section 5.3. To determine which of these metrics may be useful in determining effective coschedules, we produced a training set of 450 runs of four thread coschedules on a Nehalem processor. Using this training set, we examined the Pearson correlation [Ric94] between various metrics and both the coschedule's scaled throughput and the coschedule's fairness.

A number of raw metrics did not correlate well with either scaled throughput or fairness. For example, the sum of L2 misses per benchmark in the coschedule did not correlate well with either scaled throughput ($r = -0.28$) or fairness ($r = 0.08$). However, each of the metrics described in this section correlates relatively well with both scaled throughput and fairness (as shown in Table 5.4).

*WS{95,99}:* Using our working set data from Figure 5.3, this metric is the sum of working set sizes (95% or 99%) for each job. This policy requires either the CHRV or solely the sizes required for 95% or 99% hit rates.

*{L2,L3}MR:* This metric is the sum of miss rates for either the L2 or the L3. This requires either the actual or simulated L2 and L3 miss rates for the target machine.

*{L2,L3}MRSec:* This metric is the sum of L2 or L3 misses per second for the coscheduled threads. This requires the single threaded run time per benchmark

as well as the actual or simulated L2 and L3 miss rates for the target machine.

*{L2,L3} WSO:* This metric (working set overflow) uses the value in the CHRV for the size of the L2 or L3. This value is the miss rate which would be experienced by an L1 cache the size of the L2 or L3. Per benchmark, it represent the percentage of the working set that overflows that size cache. This differs from the actual L2 or L3 miss rates, which require simulation of the lower level caches — e.g., an actual L3 cache has highly inexact LRU information, as locality information is hidden by the other caches. This metric is the sum of such miss rates for the coscheduled benchmarks and only requires the CHRV.

*WS{95,99}-L2WSO:* This metric, and the next, try to combine the two factors that govern thread interaction — what resources are used by other threads and how sensitive the remaining threads are to that resource usage. This metric, expressed in Equation 5.4, uses WS95 or WS99 (the working set sum) to represent the utilization of shared resource and L2WSO (how often a thread goes out beyond the private caches) to represent the sensitivity of those threads to that utilization. From our training set runs, we found L2WSO to be a good measure of sensitivity. If we define sensitivity to be the standard deviation in performance seen by a benchmark in all coschedules, we found a high correlation ($r = 0.73$) between the sensitivity and the L2WSO. This metric only requires the CHRV, not a full simulation of the actual memory hierarchy.

$$\text{WS99-L2WSO} = \sum_{i=0}^{n}\sum_{j=0}^{n} \text{WS99}_i * \text{L2WSO}_j \tag{5.4}$$

*WSO-Combo:* Based on the intuition from the prior policy, this metric (Equation 5.5) uses the L2WSO as a proxy for thread sensitivity and L3WSO as a proxy for the thread's occupancy of shared resources. WSO-Combo only requires the CHRV.

$$\text{WSO-Combo} = \sum_{i=0}^{n}\sum_{j=0}^{n} \text{L3WSO}_i * \text{L2WSO}_j \tag{5.5}$$

The Pearson correlations for scaled throughput and fairness for each of these metrics are provided in Table 5.4. The best policy, WSO-Combo, which uses the CHRV, correlates better with scaled throughput and fairness than those policies which do not use the CHRV. This higher correlation is statistically significant for scaled throughput ($d = 0.1333$, $p < 0.001$) and for fairness ($d = 0.137$, $p < 0.001$).

Because we seek higher values of scaled throughput and lower values for fairness, we hope to see a metric correlated in opposite directions (one negative, one positive) with the two. It is not a given that policies that improve throughput will necessarily also improve fairness, but it turns out to be true for each of these.

### 5.5.1 Scheduling Policies

The metrics described in the previous section are translated directly into static scheduling policies. In each case, we use the metric to balance that factor across the coschedules. For example, with Nehalem processors and eight threads to run, we partition the jobs into two coschedules of four threads, finding the combination that best balances the particular metric between the two coschedules. Therefore, the L2WSO scheduling policy seeks to balance the sum of the L2WSO values between the two coschedules.

The effectiveness of these policies for predicting coschedules for scaled throughput, energy, and fairness, is shown in Section 5.6.

### 5.5.2 Profiling Requirements

A number of these metrics are system dependent. For example, to determine L2MRSec, a profiling pass must be run in which the L1 and L2 are simulated and the single threaded run time on the system is determined. Similarly, L3MR requires simulation of the L1, L2, and L3 configuration for that particular system.

A key advantage to the WSO metrics and the Cache Hit Rate Vector (CHRV) is that they do not require we simulate the actual memory hierarchy on which the job is run. This is especially important when assigning jobs to heterogeneous clusters. By providing the CHRV with a job, the job can be scheduled on diverse systems without a need for re-profiling.

To demonstrate the ubiquity of CHRV, we evaluate our performance first on Nehalem processors where the jobs were originally profiled, then on Westmere processors. For our non WSO metrics, we re-profile for the new system. For the WSO metrics reliant on the CHRV, the Westmere is challenging as the L3 size on Westmere (12MB) is not provided by the CHRV. The CHRV is still used as defined (using an average of 8MB and 16MB valued for the 12MB data) to reinforce that no additional profiling is required, even when the exact cache size is not part of the vector.

## 5.6 Evaluation

Our evaluation consists of three main components. First, we apply our proposed scheduling polices to coscheduling decisions on a system of Nehalem processors, assuming a throughput workload (fixed time study). The second applies these same policies to a different system (Westmere) with similar coscheduling assumptions (fixed time), to determine if these policies continue to apply in this new domain. Lastly, we evaluate the policies after further adjusting the coscheduling assumptions (now running a fixed number of jobs) on the Westmere system.

For all three components we evaluate scaled throughput (weighted speedup) and fairness. Due to a limited number of power measurement devices, energy is only examined in the two coscheduling scenarios on the Westmere.

### 5.6.1 Nehalem

Given the methodology described in Section 5.2, eight benchmarks are randomly chosen (allowing for duplicates) and all possible combinations of coschedules (four threads in each of two coschedules) are then run on the system (in a complete sense–every possible coschedule was evaluated). This resulted in thirty-five pairings of four jobs per schedule. By running all possible combinations, it was possible to know the best and the worst possible coschedule for each of the evaluation criteria (scaled throughput and fairness). The policies are then applied to determine which coschedule would be chosen if that policy were used.

In many of the results of this section, we use our knowledge of the worst and best coschedules to compare our results. “Best” are the results for an oracle

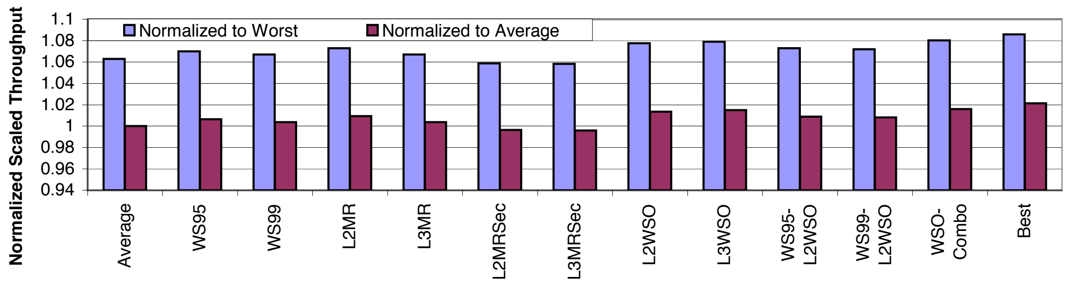


Figure 5.4: Nehalem: Scaled Throughput by Policy. Normalized to average and worst coschedule scaled throughput.

scheduler. "Normalized to Worst" are the results of each policy when compared against the pathological worst case. Thus, a value of 1.2 implies a 20% improvement over the Worst schedule. Similarly, "Normalized to Average" are the results of each policy compared against a random scheduler.

Thirty such selections of job mixes comprised of eight random benchmarks are evaluated. The results shown in this section average the results across these thirty.

Figure 5.4 demonstrates scaled throughput improvements for our policies compared against both the worst coschedule and the average coschedule. For random selections of jobs, the the worst and the best coschedules differ by about 8%, and average and best differ by only about 2% in terms of scaled throughput. Consistently avoiding the pathological bad case is important (which all of our policies except WS95 do), but the selected coschedules often offer similar scaled throughput compared with average. Given that observation, it is still possible to evaluate the effectiveness of our policies in this context. Nearly all the policies provide improvements over the average coschedule (except L2MRSec and L3MRSec). The best performer is our new policy, WSO-Combo, which provides 99% of the performance available from the best coschedule.

Although the average performance range is small, we will see that this hides a high variance in per-thread performance, resulting in a much wider range of fairness results. Fairness differs dramatically between the worst and best coschedule. In Figure 5.5 we see that the worst and best fairness differs by a factor of three. Even the difference between average and best is significant — 68%. A number

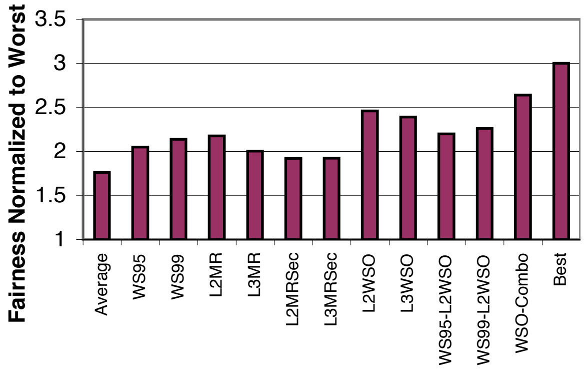


Figure 5.5: Nehalem: Fairness by Policy. Normalized to worst coschedule fairness.

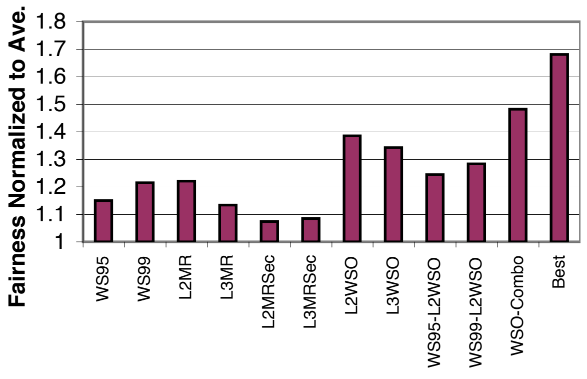


Figure 5.6: Nehalem: Fairness by Policy. Normalized to average coschedule fairness.

of our policies provide improvements over average as shown in better detail in Figure 5.6. Our proposed policies of L2WSO and WSO-Combo provide solid improvements (39% and 48% respectively) to fairness.

It should be noted that the low variance in performance we observe stems from reporting on average performance. Large systems, such as data centers, typically report performance results in terms such as the 95th percentile response time. A fair scheduling policy will show significant performance gains by those measures.

### 5.6.2 Nehalem Case Study

The fairness metric we use may be less familiar than our other performance metrics. This section, therefore, outlines a single concrete example of how

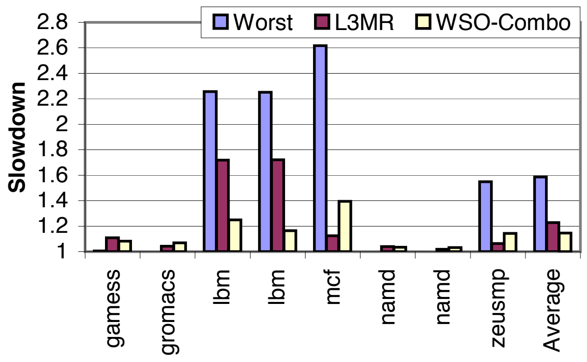


Figure 5.7: Nehalem Example: Slowdown per benchmark (compared to single-threaded execution) given different coschedules.

scheduling decisions impact relative performance of individual threads, and how that translates into different values of the fairness metric. This example has been chosen because it has a larger than typical difference between worst and best case for scaled throughput.

The jobs in this job mix include *gamess*, *gromacs*, two copies of *lbm*, *mcf*, two copies of *namd*, and *zeusmp*. From Figure 5.3, we can see that *gamess* and *namd* have small working sets, *zeusmp* has a medium sized working set, and *lbm* and *mcf* have large working sets. For these benchmarks, working set size corresponds well with the sensitivity of the benchmarks.

Figure 5.7 contains the results from running these eight threads in three different coschedules. The coschedule with the worst scaled throughput (0.75) and fairness (18.3), has *mcf*, *zeusmp*, and both copies of *lbm* in the first coschedule and the other four threads–*gamess*, *gromacs*, and both copies of *namd*–in the second coschedule. The four applications with the largest working sets are coscheduled together which leads to heavy contention for shared resources. As a result, *mcf*, *zeusmp*, and both copies of *lbm* suffer serious slowdowns while the other four threads run nearly unaffected relative to single threaded execution. This schedule is both unfair and suboptimal for global performance.

In Figure 5.7, the L3MR policy is chosen for comparison as it offers, on average, the best scaled throughput of the non-WSO policies. L3MR coschedules *gamess*, *namd*, and both copies of *lbm* in one coschedule and *gromacs*, *mcf*, *namd*, and *zeusmp* in the other coschedule. This coschedule better balances the applications with large working sets but still coschedules both copies of *lbm* together. As a result, each copy of *lbm* continues to experience a serious slowdown, but the scaled throughput (0.85) and fairness (9.9) are better.

The best policy, WSO-Combo, selects the best possible coschedule for scaled throughput (0.88) and fairness (5.8). WSO-Combo selects one copy of *lbm*, *mcf*, and both copies of *namd* for one coschedule and selects *gamess*, *gromacs*, *zeusmp*, and the other copy of *lbm*. By spreading *lbm* across the two coschedules and spreading the other sensitive applications (*mcf* and *zeusmp*) across the two coschedules, the workloads are better balanced. *mcf* and one copy of *lbm* still suffer higher slowdowns than the other coscheduled threads, but the overall slowdowns are significantly better balanced.

In the context of scheduling for a large server cluster, fairness is critical. If a user had submitted either the *lbm* or *mcf* jobs, they would be severely displeased with either the Worst or L3MR coschedule decision as their jobs would make poor progress compared to the other jobs in the system. By using our WSO-Combo policy, higher fairness is provided, ensuring all users experience similar performance.

### 5.6.3 Westmere

For the Westmere processor, twelve jobs were randomly selected per job mix. The large number of potential coschedules prohibit running all possible coschedules. Instead, twenty random coschedules (partitioning of the 12 jobs into sets of six) were selected. Of those twenty coschedules, the scheduling policies were evaluated for performance and fairness. As mentioned in Section 5.2, these coschedules were run in concert on dual socketed nodes (all twelve threads were always run together) with the only difference between the runs being the assignment of coschedules between the two processors. This potentially limits the impact of the coschedule decisions as the primary (and potentially only) difference between the coshedules are L3 sharing. (In the Nehalem studies, the time sharing of one processor by each coschedule enabled balancing of off-chip resource utilization, a benefit potentially unavailable to the Westmere studies.)

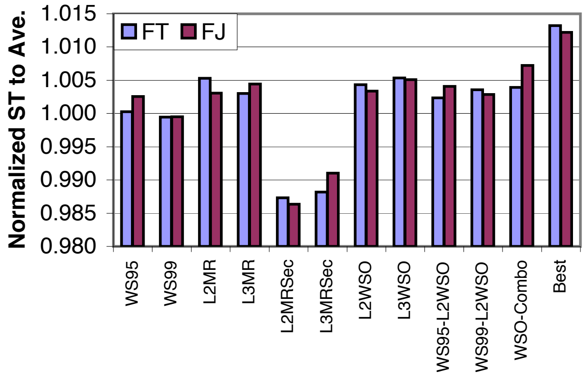


Figure 5.8: Westmere: Normalized Scaled Throughput (ST) by Policy given Fixed Time (FT) or Fixed Jobs (FJ). Normalized to average coschedule ST.

The experiments were either run for a fixed period of time “Fixed Time (FT)” or for a fixed number of job completions – three job completions per job – “Fixed Job (FJ).” Fourteen job mixes participated in our results for FT and eight job mixes participated in our results for FJ.

The 12MB size required for the L3 was not profiled (not a power of 2) and was not represented in our cache hit rate vector. We use the average of 8MB and 16MB in our policies which depend on the cache hit rate vector and the L3 size, highlighting the fact that a generic CHRV suffices.

Figure 5.8 shows that, similar to the Nehalem, scaled throughput on the Westmere did not vary significantly between our twenty randomly selected coschedules. Despite low variance in average performance, our metrics which rely on WSO, especially WSO-Combo, continues to provide solid performance, nearly (99%) of the best possible performance (for the 20 random coschedules). These trends continue for both the FT and FJ experiments.

Figure 5.9 demonstrates that the low variance in average performance again masks a significant fairness concern. Although only twenty of 462 possible combinations were run, there is still a large difference between best and worst fairness. Additionally, our policy, WSO-Combo, continues to provide the best fairness.

Lastly, we measure energy consumption for six of the FT and FJ job mixes. The FT experiment made it difficult to compare energy consumption as different amounts of total work were done in each experiment. To adjust for the differences in work done, we use scaled throughput $ST_{co}$ as a proxy for response time changes.

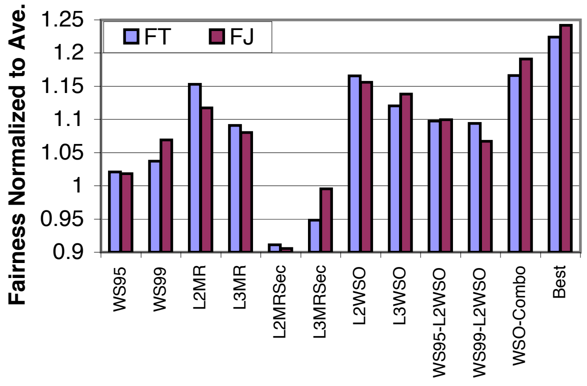


Figure 5.9: Westmere: Normalized Fairness by Policy given Fixed Time (FT) or Fixed Jobs (FJ). Normalized to average coschedule fairness.

Thus we use Energy Delay Product (EDP) from Equation 5.6 for the Fixed Time runs.

$$EDP = P_{ave} * (\frac{T}{ST_{co}})^2 \tag{5.6}$$

EDP did not differ remarkably in our fixed time runs (worst is within 7% of best and best is within 2% of average). Averaged across our measured coschedules, WSO-Combo and L3WSO selected the most efficient coschedules but were not significantly better than other coschedules or the average (less than 1% difference).

The fixed-job (FJ) experiments provided larger differences in terms of energy consumption. The best coschedules consumed 12% less energy than the worst and 4% less than average. Similarly, WSO-Combo predicted the lowest energy consumption of any of the policies, saving 11% compared to worst and 3% less than average. In terms of EDP, WSO-Combo offered a 23% reduction in EDP over the worst and a 7% reduction from average.

Since WSO-Combo selects fair coschedules, it in turn limits the most delayed benchmark and, hence, selects energy efficient coschedules. Thus we see that even if we do not strictly care about fairness, unbalanced workload progress can negatively impact tail effects (as jobs finish in a staggered fashion), resulting in increased total runtime and increased energy costs.

Our best scheduling policy, then, provides significant gains in job fairness while in no way sacrificing performance or energy efficiency. This policy relies on the CHRV provided from a single profile run on one of the two processors. The effectiveness of our policy and the CHRV is shown across two different processor configurations and two different scheduling assumptions.

## 5.7 Chapter Summary

In multi-core processors, the value of shared resources is their ability to adapt to the needs of each core while offering potentially improved performance for more than one core. However, shared resources introduce the challenge of mitigating contention between threads. To reduce this resource contention, we can exploit thread heterogeneity to better balance demand. Depending on how these heterogeneous threads are scheduled, fairness can vary significantly. For job schedulers deciding on jobs between multiple users, fairness is a significant concern, particularly if we can provide fairness while still maintaining high throughput and energy efficiency.

This paper defines and describes the cache hit rate vector (CHRV) which is available via profiling. The CHRV effectively classifies the memory behavior of an application and, combined with metrics proposed, can provide improved scheduling decisions. In addition, a single profile run can produce a CHRV which is relevant for scheduling decisions on processors with different cache sizes. Our recommended policy of WSO-Combo provides scaled throughput which is 99% of the best and fairness which is 48% better than average on a Nehalem processor. On Westmere processors, the same policy provides scaled throughput which is 99% of best, energy which is 99% of best, EDP which is 7% better than average, and fairness which is between 17% and 19% better than average given different workload assumptions.

## Acknowledgments

This chapter contains material from “Exploiting Thread Heterogeneity for Improved Throughput, Energy, and Fairness on Multicore Processors”, by Leo Porter, Allan Snavely, Anthony Gamst, and Dean M. Tullsen, which has been

submitted for possible publication by the Association for Computing Machinery in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC 11).* The dissertation author was the primary investigator and author of this paper.

# Chapter 6

# Thesis Summary

For more than four decades, we have enjoyed exponential improvements in single-thread performance. Predicted and driven by Moore's Law, the number of transistors per processor has doubled approximately every 18 months. Until the middle of the last decade, those additional transistors translated directly to improved single thread performance. It was during this past decade that the trends that drove advancements in single thread performance (pipelining, dynamic scheduling, etc.) reached their limits. Although single-thread performance has become nearly stagnant, overall thread throughput is improving; driven by the advent of simultaneous multithreaded and multi-core processors. Hence, the overall focus of processor design has shifted from single-thread latency to overall thread throughput.

Beyond transforming the landscape of processor design for architects, this shift has a significant impact on computing as a whole. In the past, programmers could rely on processors to improve the performance of their programs *transparently* with each new processor generation. Today, if programmers wish to improve their program's performance on the next generation of processor, they must design the program to exploit these increasingly parallel resources. For programmers (and companies) accustomed to the paradigm of the past, new processors capable of offering improved single-thread performance are highly desirable.

In addition to the desire to continue the paradigm of transparent single-thread performance improvements, single-thread latency remains highly germane in general. Although program latency can be improved by rewriting the software to exploit these parallel resources, it may be prohibitively expensive to do so. Additionally, some programs lack inherent parallelism and legacy binaries are predominately single-threaded.

Clearly, solutions which provide improvements to single-thread performance are still highly desirable. In this dissertation, we have examined a number of such techniques which address single-thread performance on mult-core processors.

Speculative Multithreading (SpMT) improves single thread performance by leveraging the multiple thread contexts available in multi-core processors. Unfortunately, the cost of memory hardware capable of speculative execution has been a major barrier for its adoption. In Chapter 2 we examined leveraging an investment in a recently proposed parallel memory design, Hardware Transactional Memory (HTM), to provide SpMT at a fraction of the cost. We demonstrated that HTM can only provide limited performance as proposed. To address this problem, we provide a number of inexpensive modifications to HTM including support for thread ordering, forwarding of values between threads, and word-granularity conflict detection. Using the SPEC CPU2000 benchmarks on a dual-core processor with basic register prediction hardware, SpMT only provides a 5% performance gain with conventional hardware. This improvement is boosted to 26% using our techniques. The value of these optimizations persist (at different magnitudes) for a number of different architectural designs (more cores, better register predictors, etc.).

Despite these significant gains, SpMT is still limited by performance losses due to frequent thread migrations and coherence invalidations. To address these limitations, in Chapter 3 we evaluate a recently proposed technique, Working Set Migration (WSM), in the context of SpMT. WSM was previously demonstrated to be effective at mitigating performance loss at the point of thread migration for artificial workloads. For the more realistic workload of SpMT, one of the heuristics proposed by that previous work can significantly improve SpMT performance. Using this heuristic, we reduce the average memory access time of select SPEC CPU2000 benchmarks by 50% and improve whole program performance by 9%.

Deviating from SpMT, we also recognized that the utility of various processor components changes in the multi-core landscape. For example, resources previously allocated to boosting the performance of a single core could be reallocated to provide better overall system performance via more cores, better interconnects, larger shared caches, etc. In Chapter 4 we focus on providing high branch prediction accuracy with fewer resources allocated to the branch predictor (freeing those resources for use elsewhere). To improve the branch prediction of smaller predictors which use branch history to provide predictions, we identify regions where branch history can be disruptive. For these regions, we propose a low-cost heuristic which can reduce this interference and, in turn, provide improved branch prediction accuracy. For some sizes of predictors, branch predictors using our technique are as accurate as predictors of twice the size. Although our technique was targeted at smaller branch predictors in the multi-core landscape, the technique is also effective for some larger predictors.

In multi-core processors, shared caches are valuable because they boost performance when either a single thread is using the shared cache alone or when many threads share the larger combined resource. Unfortunately, as multiple threads use a single resource, contention occurs. We addressed this contention in Chapter 5. We recognize that, in high performance computing, homogeneous threads are commonly coscheduled. We validate the intuitive assumption that homogeneous threads stress the same resources, increasing contention and worsening performance. By running coscheduled SPEC CPU2006 benchmarks on Nehalem and Westmere processors, we demonstrate that heterogeneous workloads can provide improved scaled throughput and fairness. However, scheduling heterogeneous threads introduces new complications. We propose a number of heuristics for static thread scheduling using profiled information. A novel heuristic, WSO-Combo, proves to be highly effective at predicting coschedules which offer high throughput (99% of best possible), high energy efficiency (99% of best possible), and significantly improved fairness (48% better than average).

Single thread performance remains a paramount concern for hardware designers. Although it is unlikely single-thread performance will enjoy another period of exponential growth in the near future, improvements of any reasonable magnitude are valuable. In this dissertation, we provide a number of techniques which address and improve single-thread performance in multi-core processors. These techniques effectively leverage multi-core hardware to improve single thread performance, improve branch prediction accuracy when given fewer resources, and reduce contention between coscheduled threads.

# Bibliography

[AAK$^+$05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *11th International Symposium on High-Performance Computer Architecture*, Feb. 2005.

[ACD06] James H. Anderson, John M. Calandrino, and UmaMaheswari C. Devi. Real-time scheduling on multicore platforms. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium*, Apr. 2006.

[AD98] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *31st International Symposium on Microarchitecture*, Sep. 1998.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: an infrastructure for computer system modeling. In *Computer Vol. 35, Issue. 2*, Feb. 2002.

[BAG00] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *HPC ASIA 2000, 4th International Conference on High Performance Computing*, 2000.

[BBC$^+$08] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, www.cse.nd.edu/Reports/2008TR-2008-13.pdf, 2008.

[BGH$^+$08] Jayaram Bobba, Neelam Goyal, Mark D. Hill, Michael M. Swift, and David A. Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *35th Annual International Symposium on Computer Architecture*, June 2008.

[BH04] Erik Berg and Erik Hagersten. StatCache: a probabilistic approach to efficient and accurate data locality analysis. In *International Symposium on Performance Analysis of Systems and Software*, 2004.

[BPT11] Jeffrey A. Brown, Leo Porter, and Dean M. Tullsen. Fast thread migration via cache working set migration. In *17th International Symposium on High-Performance Computer Architecture*, Feb. 2011.

[BT08] Jeffery A. Brown and Dean M. Tullsen. The shared-thread multiprocessor. In *Proceedings of the 22nd international conference on Supercomputing*, pages 73–82, June 2008.

[CCM96] I-Cheng K. Chen, John T. Coffey, and Trevor N. Mudge. Analysis of branch prediction via data compression. In *7th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996.

[CCM+06] J.W. Chung, H. Chafi, C.C. Minh, A. McDonald, B. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *Sixth International Symposium on High-Performance Computer Architecture*, Feb. 2006.

[CEP96] Po-Yung Chang, M. Evers, and Y.N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. In *International Conference on Parallel Architectures and Compilation Techniques*, Oct 1996.

[CKS+04] Francisco J. Cazorla, Peter M.W. Knijnenburg, Rizos Sakellariou, Enrique Fernández, Alex Ramirez, and Mateo Valero. Predictable performance in SMT processors. In *1st Conference on Computing Frontiers*, 2004.

[CMT00] Marcelo Cintra, José; F. Martńez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *27th Annual International Symposium on Computer Architecture*, June 2000.

[CNV+06] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Nov. 2006.

[CPT08] Bumyong Choi, Leo Porter, and Dean M. Tullsen. Accurate branch prediction for short threads. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

[CSCT02] Jamison D. Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. Pointer cache assisted prefetching. In *36th Annual ACM/IEEE international symposium on Microarchitecture*, Nov. 2002.

[CTTC06] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *33rd Annual International Symposium on Computer Architecture*, June 2006.

[CWT$^+$01] Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher J. Hughes, Yong-Fong Lee, Daniel M. Lavery, and John P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *28th Annual International Symposium on Computer Architecture*, July 2001.

[Den68] Peter J. Denning. The working set model for program behavior. *CACM*, 11(5), May 1968.

[DFL$^+$06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[EM98] A. N. Eden and T. Mudge. The YAGS branch prediction scheme. In *31st Annual ACM/IEEE international symposium on Microarchitecture*, Nov. 1998.

[EPP98] Marius Evers, Sanjay J. Patel, and Yale N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *In Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998.

[FS96] Manoj Franklin and Gurindar S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, 45(5), May 1996.

[FS06] Stanley L.C. Fung and J. Gregory Steffan. Improving cache locality for thread-level speculation. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, 2006.

[FSS07] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.

[FSSN05] Alexandra Fedorova, Margo Seltzer, Christoper Small, and Daniel Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX Annual Technical Conference*, 2005.

[GS05] Fei Gao and Suleyman Sair. Exploiting intra-function correlation with the global history stack. In *5th International Conference on Systems, Architectures, Modeling, and Simulation*, 2005.

[GST91] D. Ghosal, G. Serazzi, and S.K. Tripathi. The processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5), May 1991.

[GSYY09] Nan Guan, Martin Stigge, Wang Yi, and Ge Yu. Cache-aware scheduling and analysis for multicores. In *7th ACM International Conference on Embedded software*, 2009.

[HCW$^+$04] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (TCC). In *11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[HHS$^+$00] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE MICRO*, 20(2), Mar. 2000.

[HJG$^+$10] Jiahua He, Arun Jagatheesan, Sandeep Gupta, Jeffrey Bennett, and Allan Snavely. Dash: a recipe for a flash-based data intensive supercomputer. In *2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer. Software transactional memory for dynamic-sized data structures. In *22nd Annual Symposium on Principles of Distributed Computing*, July 2003.

[HM93] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *20th International Symposium on Computer Architecture*, May 1993.

[HM08] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. In *IEEE Computer*, July 2008.

[HP02] J. Hennessy and D. Patterson. *Computer Architecture a Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 2002.

[HWC$^+$04] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *31st International Symposium on Computer Architecture*, June 2004.

[Jac] D. Jackson. New issues and new capabilities in HPC scheduling with the Maui scheduler. `http://www.linuxclustersinstitute.org/Linux-HPC-Revolution/Archive/PDF01/JacksonUtah.pdf`.

[JL02] Daniel A. Jiménez and Calvin Lin. Neural methods for dynamic branch prediction. In *ACM Transactions on Computer Systems*, Feb. 2002.

[JRS97] Quinn Jacobson, Eric Rotenberg, and James E. Smith. Path-based next trace prediction. In *30th Annual ACM/IEEE international symposium on Microarchitecture*, 1997.

[JSN98] Toni Juan, Sanji Sanjeevan, and Juan J. Navarro. Dynamic history-length fitting: a third level of adaptivity for branch prediction. In *25th Annual International Symposium on Computer Architecture*, 1998.

[KBM$^+$10] N.A. Kurd, S. Bhamidipati, C. Mozak, J.L. Miller, T.M. Wilson, M. Nemani, and M. Chowdhury. Westmere: A family of 32nm ia processors. In *Solid-State Circuits Conference Digest of Technical Papers*, Feb. 2010.

[KCS04] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *13th International Conference on Parallel Architectures and Compilation Techniques*, 2004.

[KDMK08] N. Kurd, J. Douglas, P. Mosalikanti, and R. Kumar. Next generation intel; micro-architecture (nehalem) clocking architecture. In *IEEE Symposium on VLSI Circuits*, June 2008.

[Kes99] R. E. Kessler. The alpha 21264 microprocessor. *IEEE MICRO*, 19(2):24–36, Mar. 1999.

[KT98] Venkata Krishnan and Josep Torrellas. Hardware and software support for speculative execution of sequential binaries on a chip-multiprocessor. In *12th International Conference on Supercomputing*, July 1998.

[LCM97] Chih-Chieh Lee, I-Cheng Chen, and Trevor Mudge. The bi-mode branch predictora. In *30th Annual ACM/IEEE international symposium on Microarchitecture*, 1997.

[LGC97] J. Labarta, S. Girona, and T. Cortes. Analyzing scheduling policies using Dimemas* 1. *Parallel Computing*, 23(1-2), 1997.

[LLD$^+$08] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *14th International Symposium on High Performance Computer Architecture*, Feb. 2008.

[Loh04] Gabriel Loh. The frankenpredictor. In *The 1st JILP Championship Branch Competition (CBP-1)*, 2004.

[Loh05] Gabriel H. Loh. A simple divide-and-conquer approach for neural-class branch prediction. In *14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.

[LTCS10] M.A. Laurenzano, M.M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems Software*, March 2010.

[MBM$^+$06] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-based transactional memory. In *12th International Symposium on High-Performance Computer Architecture*, Feb. 2006.

[McC85] E.M. McCreight. The dragon computer system. In *Proceedings of the NATO Advanced Science Institute on Microarchitecture of VLSI Computers*, 1985.

[MCC$^+$05] Austen McDonald, JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Brian D. Carlstrom, Lance Hammond, Christos Kozyrakis, and Kunle Olukotun. Characterization of TCC on chip-multiprocessors. In *14th International Conference on Parallel Architectures and Compilation Techniques*, Sep. 2005.

[McF93] Scott McFarling. Combining branch predictors. In *DEC WRL Technical Note TN-36*. DEC Western Research Laboratory, 1993.

[MG00] Pedro Marcuello and Antonio Gonzalèz. A quantitative assessment of thread-level speculation techniques. In *14th International Symposium on Parallel and Distributed Processing*, May 2000.

[MG02] Pedro Marcuello and Antonio Gonzélez. Thread-spawning schemes for speculative multithreading. In *Second International Symposium on High-Performance Computer Architecture*, page 55, Washington, DC, USA, Feb. 2002. IEEE Computer Society.

[MGQS$^+$08] Carlos Madriles, Carlos García Quiñones, Jesús Sánchez, Pedro Marcuello, Antonio Gonzàlez, Dean M. Tullsen, Hong Wang, and John P. Shen. Mitosis: A speculative multithreaded processor based on precomputation slices. In *IEEE Transactions on Parallel and Distributed Systems*, July 2008.

[MGT98] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. In *12th International Conference on Supercomputing*, Nov. 1998.

[MHW05] K. Moore, M. Hill, and D. Wood. Thread-level transactional memory. In *Technical Report 1524, Computer Sciences Dept., UW-Madison*, 2005.

[MSWP03] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Third International Symposium on High-Performance Computer Architecture*, 2003.

[NS10] M.L. Norman and A. Snavely. Accelerating data-intensive science with Gordon and Dash. In *2010 TeraGrid Conference*, 2010.

[PCC07] Christoph Von Praun, Luis Ceze, and Calin Cascaval. Implicit parallelism with ordered transactions. In *12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Sep. 2007.

[PCT09] Leo Porter, Bumyong Choi, and Dean M. Tullsen. Mapping out a path from hardware transactional memory to speculative multithreading. In *18th International Conference on Parallel Architectures and Compilation Techniques*, 2009.

[PE00] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *Sixth International Symposium on High-Performance Computer Architecture*, 2000.

[PO05] Manohar K. Prabhu and Kunle Olukotun. Exposing speculative thread parallelism in SPEC2000. In *10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

[PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *11th Annual International Symposium on Computer Architecture*, Jan. 1984.

[PSR92] Shien-Tai Pan, Kimming So, and Joseph T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *SIGPLAN Not. Vol. 27 No. 9*, 1992.

[RHL05] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *32nd Annual International Symposium on Computer Architecture*, June 2005.

[Ric94] John A. Rice. *Mathematical Statistics and Data Analysis.* Duxbury Press, 2nd edition, June 1994.

[RMS98] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *8th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1998.

[RRW08] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *41st Annual ACM/IEEE international symposium on Microarchitecture*, Nov. 2008.

[SAHL04] Srikanth T. Srinivasan, Haitham Akkary, Tom Holman, and Konrad Lai. A minimal dual-core speculative multi-threading architecture. In *IEEE International Conference on Computer Design*, Oct. 2004.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[SDS08] Arrvindh Shriraman, Sandhya Dwarkadas, and Michael L. Scott. Flexible decoupled transactional memory support. In *35th Annual International Symposium on Computer Architecture*, June 2008.

[SEP98] Jared Stark, Marius Evers, and Yale N. Patt. Variable length path branch prediction. *SIGPLAN Not.*, 33(11), 1998.

[Sez05] Andre Seznec. Analysis of the o-geometric history length branch predictor. In *32nd Annual International Symposium on Computer Architecture*, 2005.

[Sez07] Andre Seznec. The l-tage branch predictor. In *Journal of Instruction-Level Parallelism, vol. 9*, May 2007.

[SFKS02] André; Seznec, Stephen Felix, Venkata Krishnan, and Yiannakis Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. In *29th Annual International Symposium on Computer Architecture*, June 2002.

[SM98] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *4th International Symposium on High-Performance Computer Architecture*, Jan. 1998.

[SM99] A. Seznec and P. Michaud. De-aliashed hybrid branch predictors. In *Technical Report RR-3618, Inria*, Feb. 1999.

[SMCK08] Yiannakis Sazeides, Andreas Moustakas, Kypros Constantinides, and Marios Kleanthous. The significance of affectors and affectees correlations for branch prediction. In *3rd International Conference on High Performance and Embedded Architectures and Compilers*, 2008.

[Smi98] James E. Smith. A study of branch prediction strategies. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[SMJ$^+$10] M. Aater Suleman, Onur Mutlu, José A. Joao, Khubaib, and Yale N. Patt. Data marshaling for multi-core architectures. In *37th Annual International Symposium on Computer Architecture*, June 2010.

[SPE] SPEC. 2006 website. `http://www.spec.org/cpu2006/`.

[SPHC02] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[SR01] Gurindar S. Sohi and Amir Roth. Speculative multithreaded processors. *Computer*, 34(4), 2001.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *Fourteenth Annual ACM symposium on Principles of distributed computing*, Aug. 1995.

[ST00] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.

[SZG$^+$09] Sadagopan Srinivasan, Li Zhao, Brinda Ganesh, Bruce Jacob, Mike Espig, and Ravi Iyer. CMP memory modeling: How much does accuracy matter? In *5th Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[TC08] Marc Tremblay and Shailender Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC®processor. In *IEEE International Solid-State Circuits Conference*, Feb. 2008.

[TEE+96] Dean M. Tullsen, Susan J. Eggers, Joel S. Emer, Henry M. Levy, Jack L. Lo, and Rebecca L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *23rd Annual International Symposium on Computer Architecture*, May 1996.

[TFWS03] Renju Thomas, Manoj Franklin, Chris Wilkerson, and Jared Stark. Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history. In *30th Annual International Symposium on Computer Architecture*, 2003.

[TKTC04] Eric Tune, Rakesh Kumar, Dean M. Tullsen, and Brad Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *37th Annual ACM/IEEE international symposium on Microarchitecture*, pages 183–194, Dec. 2004.

[TTG97] Maria-Dana Tarlescu, Kevin B. Theobald, and Guang R. Gao. Elastic history buffer: A low-cost method to improve branch prediction accuracy. In *International Conference on Computer Design*, 1997.

[Tul96] D.M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, Dec. 1996.

[VGSS01] T.N. Vijaykumar, S. Gopal, J.E. Smith, and G. Sohi. Speculative versioning cache. *IEEE Transactions on Parallel and Distributed Systems*, 12(12), Dec. 2001.

[Vij98] T. N. Vijaykumar. Compiling for the multiscalar architecture. In *Ph.D. Thesis. University of Wisconsin-Madison*, 1998.

[Wat] Watts. Watts up? website. `http://www.wattsupmeters.com`.

[WCC+07] P.H. Wang, J.D. Collins, G.N. Chinya, H. Jiang, X. Tian, M. Girkar, N.Y. Yang, G.Y. Lueh, and H. Wang. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007.

[WF03] Y. Wiseman and D.G. Feitelson. Paired gang scheduling. *Parallel and Distributed Systems, IEEE Transactions on*, 14(6), June 2003.

[WS06a] J. Weinberg and A. Snavely. Symbiotic space-sharing on sdsc's datastar system. In *12th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2006.

[WS06b] J. Weinberg and A. Snavely. User-guided symbiotic space-sharing of real workloads. In *20th ACM International Conference on Supercomputing*, June 2006.

[WS08] J. Weinberg and A. Snavely. Chameleon: A framework for observing, understanding, and imitating memory behavior. In *PARA08: Workshop on State-of-the-Art in Scientific and Parallel Computing*, May 2008.

[YBM+07] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volosand Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *13th International Symposium on High-Performance Computer Architecture*, Feb. 2007.

[YP93] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium on Computer Architecture*, May 1993.

[YZ08] Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared l2 instruction caches. *IEEE Real-Time and Embedded Technology and Applications Symposium*, 2008.

[ZCSM02] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *SIGPLAN Notices*, 2002.