UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Architectural support for efficient on-chip parallel execution

Permalink
https://escholarship.org/uc/item/0r1593xh

Author
Brown, Jeffery Alan

Publication Date
2010

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Architectural Support for Efficient On-chip Parallel Execution

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

in

Computer Science

by

Jeffery Alan Brown

Committee in charge:

Professor Dean Tullsen, Chair
Professor Brad Calder
Professor Sadik Esener
Professor Tim Sherwood
Professor Steven Swanson

2010

Copyright
Jeffery Alan Brown, 2010
All rights reserved.

The dissertation of Jeffery Alan Brown is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____

Chair

University of California, San Diego

2010

DEDICATION

To Mom.

EPIGRAPH

Do what you think is interesting, do something that you think is fun and worthwhile, because otherwise you won't do it well anyway.

—Brian Kernighan

Numerical examples, are good for your soul.

—T. C. Hu

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . ix

List of Tables . . . x

Acknowledgements . . . xi

Vita and Publications . . . xiv

Abstract of the Dissertation . . . xvi

Chapter 1 Introduction . . . 1
- 1.1 Complications from Parallelism . . . 2
- 1.2 Memory Latency & Instruction Scheduling . . . 3
- 1.3 Cache Coherence on a CMP Landscape . . . 4
- 1.4 Thread Migration . . . 5
  - 1.4.1 Explicit Thread State: Registers . . . 6
  - 1.4.2 Implicit State: Working Set Migration . . . 7

Chapter 2 Experimental Methodology & Metrics . . . 8
- 2.1 Execution-driven Simulation . . . 8
  - 2.1.1 SMTSIM . . . 9
  - 2.1.2 RSIM . . . 10
- 2.2 Metrics . . . 11
  - 2.2.1 Weighted Speedup . . . 11
  - 2.2.2 Normalized Weighted Speedup . . . 13
  - 2.2.3 Interval Weighted Speedup . . . 14
  - 2.2.4 Interval IPC & Post-migrate Speedup . . . 15

Chapter 3 Handling Long-Latency Loads on Simultaneous Multithreading Processors . . . 17
- 3.1 Introduction . . . 17
- 3.2 The Impact of Long-latency Loads . . . 18
- 3.3 Related Work . . . 20
- 3.4 Methodology . . . 22

3.5 Metrics 26
3.6 Detecting and Handling Long-latency Loads 26
3.7 Alternate Flush Mechanisms 31
3.8 Response Time Experiments 33
3.9 Generality of the Load Problem 36
3.10 Summary 39

Chapter 4 Coherence Protocol Design for Chip Multiprocessors 42
4.1 Introduction 42
4.2 Related Work 44
4.3 A CMP Architecture with Directory-based Coherence . . 46
4.3.1 Architecture 46
4.3.2 Baseline Coherence Protocol 49
4.4 Accelerating Coherence via Proximity Awareness 51
4.5 Methodology 56
4.6 Analysis and Results 56
4.7 Summary 63

Chapter 5 The Shared-Thread Multiprocessor 65
5.1 Introduction 65
5.2 Related Work 67
5.3 The Baseline Multi-threaded Multi-core Architecture . . 68
5.3.1 Chip multiprocessor 69
5.3.2 Simultaneous-Multithreaded cores 69
5.3.3 Long-latency memory operations 70
5.4 Shared-Thread Storage: Mechanisms & Policies 71
5.4.1 Inactive-thread store 71
5.4.2 Shared-thread control unit 72
5.4.3 Thread-transfer support 72
5.4.4 Scaling the Shared-Thread Multiprocessor 74
5.4.5 Thread control policies — Hiding long latencies . 75
5.4.6 Thread control policies — Rapid rebalancing . . . 79
5.5 Methodology 82
5.5.1 Simulator configuration 82
5.5.2 Workloads 84
5.5.3 Metrics 85
5.6 Results and Analysis 86
5.6.1 Potential gains from memory stalls 86
5.6.2 Rapid migration to cover memory latency 87
5.6.3 Rapid migration for improved scheduling 89
5.7 Summary 91

Chapter 6 Fast Thread Migration via Working Set Prediction 93
6.1 Introduction 93
6.2 Related Work 96
6.3 Baseline Multi-core Architecture 97
6.4 Motivation: Performance Cost of Migration 98
6.5 Architectural Support for Working Set Migration 102
6.5.1 Memory logger 103
6.5.2 Summary generator 107
6.5.3 Summary-driven prefetcher 108
6.6 Methodology 110
6.6.1 Simulator configuration 110
6.6.2 Workloads 110
6.6.3 Metrics 112
6.7 Analysis and Results 113
6.7.1 Bulk cache transfer 113
6.7.2 Limits of prefetching 115
6.7.3 I-stream prefetching 116
6.7.4 D-stream prefetching 117
6.7.5 Combined prefetchers 118
6.7.6 Allowing previous-instance cache re-use 119
6.7.7 Impact on other threads 120
6.7.8 Adding a shared last-level cache 121
6.7.9 Simple hardware prefetchers 121
6.8 Summary 122

Chapter 7 Conclusion 124
7.1 Memory Latency in Multithreaded Processors 125
7.2 Cache Coherence for CMPs 126
7.3 Multithreading Among Cores 127
7.3.1 Registers: Thread Migration & Scheduling 128
7.3.2 Memory: Working Set Prediction & Migration . . 129
7.4 Final Remarks 130

Bibliography 131

LIST OF FIGURES

Figure 1.1: Example chip-level parallel architecture 2

Figure 2.1: Interval weighted speedup compression function 15

Figure 3.1: Performance impact of long-latency loads 19
Figure 3.2: Throughput benefit of a simple flushing mechanism 28
Figure 3.3: Comparison of long-load detection mechanisms 29
Figure 3.4: Performance of flush-point selection techniques 30
Figure 3.5: Performance of alternative flush mechanisms 33
Figure 3.6: Mean response times in an open system 35
Figure 3.7: Impact on alternate SMT fetch policies 37
Figure 3.8: Performance with different instruction queue sizes 38

Figure 4.1: Baseline chip multiprocessor 47
Figure 4.2: A traditional multiprocessor 48
Figure 4.3: Proximity-aware coherence 52
Figure 4.4: Potential benefit from proximity-aware coherence 58
Figure 4.5: Reduction in L2 miss-service latency 60
Figure 4.6: Reply-network utilization 61
Figure 4.7: Speedup from proximity-aware coherence 62

Figure 5.1: The Shared-Thread Multiprocessor 71
Figure 5.2: Idle time from memory stalls in fully-occupied cores 87
Figure 5.3: Performance of stall-covering schemes 88
Figure 5.4: Performance of dynamic schedulers 89

Figure 6.1: Baseline multi-core processor 99
Figure 6.2: Performance cost of migration 100
Figure 6.3: Memory logger overview 104
Figure 6.4: Summary-driven prefetcher 109
Figure 6.5: Impact of bulk cache transfers 114
Figure 6.6: The limits of a future-oracle prefetcher 115
Figure 6.7: Comparison of instruction stream prefetchers 116
Figure 6.8: Comparison of data stream prefetchers 117
Figure 6.9: Speedup from combined instruction and data prefetchers 118
Figure 6.10: Speedup when allowing cache reuse 120

LIST OF TABLES

Table 3.1: Single-threaded benchmarks 23
Table 3.2: Multi-threaded workloads 24
Table 3.3: Processor configuration 25

Table 4.1: Architecture details 57
Table 4.2: Workloads 57

Table 5.1: Architecture details 83
Table 5.2: Component benchmarks 84
Table 5.3: Composite workloads 85

Table 6.1: Activity record fields 103
Table 6.2: Baseline processor parameters 111
Table 6.3: Prefetcher activity and accuracy 119

ACKNOWLEDGEMENTS

I thank my advisor, Dean Tullsen, for making this dissertation possible; for setting an example of professional and personal integrity for us all to aspire toward; and especially for standing by me during the tough years – when experiments weren't working – displaying unwavering confidence at times when I was ready to panic.

I thank the additional members of my thesis committee – Brad Calder, Sadik Esener, Tim Sherwood, and Steven Swanson – for donating their valuable time to me, for reviewing my work, and overseeing the completion of this dissertation. I send further thanks to Geoff Voelker and Glenn Reinman, for contributing at earlier stages of this work, and to Geoff for providing a voice of calm counsel and reassurance over the years. To you all, I'm humbled by your assistance; thank you.

Thanks to my mother, to which I owe everything in life, for always being there for me with unconditional moral support; for burying me with books since before I could walk, then engaging and encouraging me ever since: from reviewing spelling while driving to school ("o-c, e-a-n"), to listening with bewildered patience each time I would explain how I *still* wasn't finished with graduate school. I love you Mom; you are the best.

I wouldn't be who I am today without my big brother. The years we spent finishing each others' sentences were the happiest of my life. I only ever wanted to be like you; I would give absolutely anything to have you back.

I thank my stepfather, for setting a lifelong example of what it means to stand up and be a man; for teaching me that anything worth doing, is worth doing right; for teaching me to be curious, to always wonder what's over the next hill, around the next bend in the river; for supporting us in the most important ways. Five by five, Chief. I'm sorry I couldn't finish this a few months sooner for you.

I thank my father, for his endless patience during my formative years; for encouraging me to ask questions; for always having time for one more "why?", no matter how exhausted he was; for teaching me that being "just a kid" was not something to keep me from comprehending or attempting grown-up things.

Life at UCSD has been enhanced by all the great people I've met in the CSE department. Thanks to my lab-mates for keeping life fun. To John, Tim, and Jeremy: thanks for the hilarious antics over the years; it's a miracle we didn't break anything. Thanks to Jeremy for the times we spent together, those many hours on sysadmin duty; to Rakesh for teaching me the power of a smile, and of perspective; to Leo for listening to me complain during the worst of times; to Jack – a quirky guy, to be sure – for making me laugh, and for being as dependable a friend as they come. Thank you to my roommates Nathan and Jack, for putting up with my strange hours, my griping about school, and most of all for keeping life at home drama-free: school would've been impossible without that. I'm going to miss UCSD — it's a great place to be!

Chapter 3 contains material from "Handling Long-Latency Loads in a Simultaneous Multithreading Processor", by Dean M. Tullsen and Jeffery A. Brown, which appears in *Proceedings of the 34th annual International Symposium on Microarchitecture (MICRO)*. The dissertation author was the secondary investigator and author of this paper. The material in Chapter 3 is copyright ©2001 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 4 contains material from "Proximity-Aware Directory-based Coherence for Multi-core Processor Architectures", by Jeffery A. Brown, Rakesh Kumar, and Dean Tullsen, which appears in *Proceedings of the Nineteenth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*. The dissertation author was the primary investigator and author of this paper. The material in Chapter 4 is copyright ©2007 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 "The Shared-Thread Multiprocessor", by Jeffery A. Brown and Dean M. Tullsen, which appears in *Proceedings of the 2008 ACM International Conference on Supercomputing (ICS)*. The dissertation author was the primary investigator and author of this paper. The material in Chapter 5 is copyright ©2008 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 6 contains material from "Fast Thread Migration via Cache Working Set Prediction", by Jeffery A. Brown and Dean M. Tullsen, which has been submitted for possible publication by the Association for Computing Machinery in *Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*. The dissertation author was the primary investigator and author of this paper.

VITA

2000	Bachelor of Science in Computer Science *summa cum laude* University of California, San Diego
2000	Internship Computing Sciences Research Center at Bell Labs Murray Hill, New Jersey
2001–2003	Graduate Research Fellow National Science Foundation
2002	Master of Science in Computer Science University of California, San Diego
2002	Internship Intel Corporation, Microarchitecture Research Lab Santa Clara, California
2004	Internship Intel Corporation, Microarchitecture Research Lab Santa Clara, California
2006	Instructor Department of Computer Science & Engineering University of California, San Diego
2010	Doctor of Philosophy in Computer Science University of California, San Diego

PUBLICATIONS

"The Shared-Thread Multiprocessor" Jeffery A. Brown, Dean M. Tullsen. *Proceedings of the 2008 ACM International Conference on Supercomputing (ICS)*, pages 73–82, June 2008.

"Proximity-Aware Directory-based Coherence for Multi-core Processor Architectures" Jeffery A. Brown, Rakesh Kumar, Dean Tullsen. *Proceedings of the Nineteenth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 126–134, June 2007.

"Speculative Precomputation on Chip Multiprocessors" Jeffery A. Brown, Hong Wang, George Chrysos, Perry H. Wang, John P. Shen. *Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation (MTEAC)*, pages 35–42, November 2002.

“Code-Red: a case study on the spread and victims of an Internet worm” David Moore, Colleen Shannon, Jeffery Brown. *ACM SIGCOMM Internet Measurement Workshop (IMW)*, pages 273–284, November 2002.

“Handling Long-Latency Loads in a Simultaneous Multithreading Processor” Dean M. Tullsen, Jeffery A. Brown. *Proceedings of the 34th International Symposium on Microarchitecture (MICRO)*, pages 318–327, December 2001.

“Network Performance Visualization: Insight Through Animation” Brown J.A., McGregor A.J., Braun H-W. *Proceedings of the 1st Passive and Active Measurement Workshop (PAM)*, pages 33–41, April 2000.

ABSTRACT OF THE DISSERTATION

Architectural Support for Efficient On-chip Parallel Execution

by

Jeffery Alan Brown

Doctor of Philosophy in Computer Science

University of California, San Diego, 2010

Professor Dean Tullsen, Chair

Exploitation of parallelism has for decades been central to the pursuit of computing performance. This is evident in many facets of processor design: in pipelined execution, superscalar dispatch, pipelined and banked memory subsystems, multithreading, and more recently, in the proliferation of cores within chip multiprocessors (CMPs). As designs have evolved, and the parallelism dividend of each technique have been exhausted, designers have turned to other techniques in search of ever more parallelism.

The recent shift to multi-core designs is a profound one, since available parallelism promises to scale farther than at prior levels, limited by interconnect degree and thermal constraints. This explosion in parallelism necessitates changes in how hardware and software interact. In this dissertation, I focus on hardware aspects of this interaction, providing support for efficient on-chip parallel execution in the face of increasing core counts.

First, I introduce a mechanism for coping with increasing memory latencies in multithreaded processors. While prior designs coped well with instruction latencies in the low tens of cycles, I show that long latencies associated with stalls

for main memory access lead to pathological resource hoarding and performance degradation. I demonstrate a reactive solution which more than doubles throughput for two-thread workloads.

Next, I reconsider the design of coherence subsystems for CMPs. I show that implementation of a traditional directory protocol on a CMP fails to take advantage of the latency and bandwidth landscape typical of CMPs. Then, I propose a CMP-specific customization of directory-based coherence, and use it to demonstrate overall speedup, reduced miss latency, and decreased interconnect utilization.

I then focus on improving hardware support for multithreading itself, specifically for thread scheduling, creation, and migration. I approach this from two complementary directions. First, I augment a CMP with support for rapidly transferring register state between execution pipelines and off-core thread storage. I demonstrate performance improvement from accelerated inter-core threading, both by scheduling around long-latency stalls as they occur, and by running a conventional multi-thread scheduler at higher sample rates than would be possible with software alone. Second, I consider a key bottleneck for newly-forked and newly-rescheduled threads: the lack of useful cached working sets, and the inability of conventional hardware to quickly construct those sets. I propose a solution which uses small hardware tables that monitor the behavior of executing threads, prepares working-set summaries on demand, and then uses those summaries to rapidly prefetch working sets when threads are forked or migrated. These techniques as much as double the performance of newly-migrated threads.

Chapter 1

Introduction

Much progress in processor architecture can be characterized as the successful exploitation of parallelism in computation, by increasingly-parallel hardware. Each generation of hardware is capable of performing many more simultaneous operations than its predecessor, prompting the recurring challenge of keeping that hardware occupied with useful work. There is an ever-growing gap between the amount of raw parallelism available in hardware, and the effectiveness with which we are able to use it. As transistor development continues apace, architecture design becomes more demanding: with ever-more transistors at our disposal, we must expose and exploit ever-more parallelism in order to efficiently utilize them.

The raw hardware in a processor is highly parallel by nature, yet the amount of parallelism available at a point in computation – the number of independent operations which can begin at that point – varies greatly, both with overall workload and also moment-to-moment within a single workload. A central theme in processor design is the organization of hardware, and the software interface to that hardware, so that computation can be expressed as fragments which hardware can execute efficiently, overlapping operations wherever possible for higher performance.

The focus of this dissertation is enabling efficient, on-chip parallel execution. We will consider a prototypical chip-level parallel architecture, some roadblocks to utilizing increasingly-parallel hardware, and methods to mitigate those problems.

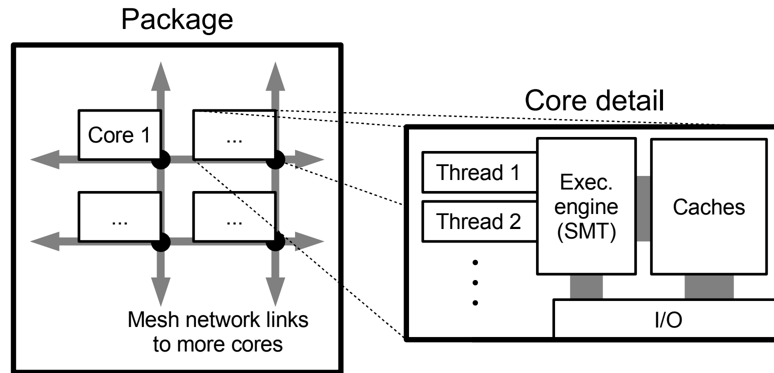


Figure 1.1: Example chip-level parallel architecture.

1.1 Complications from Parallelism

Consider the prototypical chip-multiprocessor, or CMP, depicted in Figure 1.1. This chip is composed of several tiles, interconnected by an on-chip mesh network; each tile contains a superscalar, simultaneous-multithreaded general-purpose processor core, cache storage, and network interface hardware. Such a processor has abundant opportunities for parallel execution:

- Instruction-level parallelism (ILP): Pipelined execution breaks each instruction into multiple steps, and overlaps the steps of different instructions, allowing the next instruction to begin execution before a given instruction is complete. Furthermore, superscalar hardware allows multiple independent instructions to begin execution at the same time. Both features multiply potential parallelism.

- Same-core, thread-level parallelism (TLP): During execution, there are points where a given software thread does not offer many independent instructions, leaving the execution hardware mostly idle. One approach to boost resource utilization is Simultaneous Multithreading (SMT): modest additional hardware allows instructions from multiple software threads to contend for executions resources in each cycle. Aside from explicit synchronization operations, instructions from different threads are guaranteed independent of each other,

enabling better utilization of each core's parallel execution hardware without relying on additional speculation.

- Cross-core parallelism: The tiles shown in Figure 1.1 operate independently of each other, except when communicating either with each other for cache-coherence purposes, or with off-chip devices (e.g. memory). As such, software threads submitted to different cores execute independently; each additional core contributes another set of execution units, offering yet more parallelism throughout the system.

While this example is a contemporary design that offers significant parallelism and promises to scale well with increased transistor counts, we will see that straightforward implementations of multi-core designs fall short of their potential. The introduction of more complicated interconnects and contending cores give rise to bottlenecks which are not a factor in single-core designs. We will identify, explore, and address several of these problems.

1.2 Memory Latency & Instruction Scheduling

One common performance obstacle for general-purpose processors is long-latency memory operations – i.e. loads that miss in local caches – which can occur unpredictably. While such a request is pending, the processor cannot complete dependent operations, effectively decreasing the amount of parallelism available in that instruction stream.

Simultaneous multithreading processors were introduced in part to better utilize processors in such situations, by freely scheduling instructions from independent threads to cover deficits in the instruction-level parallelism available from any one thread. SMT copes well with short- to medium-term latencies – e.g. cache-misses which are serviced on-chip – since it is able to bring in additional instructions as needed, on a cycle-by-cycle basis, without the thread-switching overhead that prevents prior multithreading techniques from being profitably applied at these time scales.

In Chapter 3 of this dissertation, we show that the use of SMT introduces a new scheduling problem: SMT processors do not handle very-long-latency operations as well as other models of multithreading; even the best among the resource-allocation policies of prior work fall victim to resource hoarding, which arises in the face of long-latency loads. By holding resources when stalled for a long-latency load, a stalled thread impedes the execution of others, even though there is no explicit dependence between them, and without any performance gain that might justify the resource imbalance. The root of the problem is that, even under the best of the prior policies, stalled threads are able to both hold previously-acquired execution resources, and to continue to trickle in additional instructions; the scheduler has no way to revoke these resources.

We present hardware techniques to identify instances of resource-hoarding behavior, before they lead to pathological slow-down: we detect when a thread is stalled on a load which has missed in the L2 cache – a simple "cycles-since-issue" timer suffices – and then flush dependent instructions from the responsible thread, using existing speculative-execution facilities. After reclaiming execution resources from the stalled thread, we block it from fetching until the offending load completes, allowing co-scheduled threads better use of the system in the meantime. We show that this leads to significantly higher and more predictable throughput for all threads.

1.3 Cache Coherence on a CMP Landscape

In current multi-core designs, common hardware memory models feature shared memory, wherein any core can directly access any piece of physical memory at any time. Cache coherence is necessary in such systems to ensure that caching is done safely: individual cores have their own caches for the sake of performance, and when one core attempts to write to a block which is also cached on another, action must be taken to prevent inconsistency. As coherence policy governs all interaction between processors and memory modules, both the policy and implementation are performance-critical details of our increasingly-parallel systems.

Cache coherence has been well-studied over the years, traditionally in the context of large distributed-memory machines. Traditional systems present individual processors grouped with memory modules into compute nodes, which are separated from other nodes through a relatively high-latency interconnect; local memory is “closer” than peer processors or their memory. Multi-core designs offer a different processor-memory topology: the processors themselves are tightly grouped, often fabricated together on the same die, with a fast on-die interconnect; memory modules are separated from the processors by another, high-latency interconnect. Peer processors are then “closer” than any memory, and this offers different communication trade-offs.

In Chapter 4 we evaluate a directory-based coherence protocol on a CMP system. We show that simply implementing traditional directory protocols within a chip does not provide the most effective solution, due to the altered communication latency landscape. We demonstrate an improved protocol which prefers communication with on-chip peers over off-chip memory: in the service of L2 cache misses, we query the caches of “bystander” nodes listed as sharers, before resorting to main-memory access; these queries are further ordered to minimize reply-network bandwidth utilization. Our protocol customizations decrease overall miss-service latency by searching on-chip where possible, and decrease the total amount of communication performed system-wide by preferring shorter return paths for data-carrying replies.

1.4 Thread Migration

As system core counts increase and software is adapted to utilize these cores, thread-level parallelism is taking on ever-increasing significance in the quest for performance. Though this progression presents interesting programming model and operating system challenges, the efficiency of underlying thread-management mechanisms is inescapably critical: software solutions to the problems of expressing TLP still rely on the underlying hardware to carry out scheduling decisions.

Our growing dependence on thread-level parallelism ensures that thread-

management mechanisms will hold a central role in enabling system-wide performance scaling. Any inefficiencies in thread activation, deactivation, spawning, or migration will take a toll, both by decreasing the rate at which scheduling decisions can profitably be made, and increasing the minimum granularity for profitably creating new threads – or activating worker threads from an idle pool – to exploit short-term or irregular parallelism.

1.4.1 Explicit Thread State: Registers

While several historical parallel systems featured efficient hardware context-switching, these machines sacrificed single-thread performance, required extensive additional hardware for thread state storage, or required workloads be expressed as data-flow streams. Parallel systems today, by contrast, typically perform scheduling using minimal hardware support – privileged software makes scheduling decisions, which are then effected by executing a series of ordinary loads and stores followed by a specialized jump instruction – with minimum latencies of hundreds to thousands of cycles.

In Chapter 5 we start with a multi-core SMT processor, to which we add a mechanism for rapidly moving architected thread state between execution pipelines and shared, off-core inactive-thread storage. We implement these context switches by halting a thread's normal execution, and then injecting register "spill" and "fill" pseudo-instructions into the pipeline. Each of these pseudo-instructions uses the existing renaming and dependence-resolution hardware to copy one logical register value between the core execution state and a small transmit/receive buffer, which is in turn transferred to or from the shared inactive-thread storage. This enables low-latency "multithreading-like" context switching across cores, with latencies in the dozens of cycles.

Using minimal additional hardware, this approach combines the relative simplicity of contemporary multi-core designs with the flexibility and high utilization of large-scale multithreaded systems. We demonstrate that this system can perform scheduling quickly enough to allow for thread scheduling to take advantage of idle resources during memory stalls, and to reschedule batches of threads to de-

tect and avoid inter-thread resource conflicts more effectively than with unassisted software.

1.4.2 Implicit State: Working Set Migration

Even with the hardware support we provide in Chapter 5, the performance cost of creating or moving a thread remains high, requiring tens of thousands of commits for performance to recover afterward. The most significant source of this degradation is due to the lack of cache-resident working set. While architected thread state (e.g. register values) can be easily cataloged and transferred, instruction and data working sets are not explicitly exposed to hardware, and migrated or spawned threads are left to implicitly recreate their own working set as they generate demand references. Unfortunately, the lack of a resident working set limits the amount of ILP visible to the processor – everything becomes stalled for cache misses – severely restricting the rate at which the thread can generate new demand references to bring in more of its working set.

In Chapter 6, we frame this working-set problem more specifically in terms of performance degradation subsequent to individual thread migrations. We show that performance suffers greatly in these instances, due to cache effects, effectively placing a lower limit on the grain-size which is available for scheduling. We also show that simply copying existing cache state is ineffective over the time scales in which performance suffers the most.

We introduce a mechanism to address this. Our system creates compact summaries of a threads' working sets as they execute, and uses that summary data to efficiently prefetch useful instruction and data working sets when a thread is moved to a different core. Thread summary data is passively collected by a set of small hardware tables which are inspired by prefetcher design – with one table detecting striding memory accesses, another detecting repeated accesses to heap objects, etc. – and summary data is compacted into a simple range-encoded format for transfer to other cores and later prefetching. We evaluate a range of behavior-specific hardware tables, and find that even a combination of the simplest ones as much as doubles the performance of threads shortly after migrations.

Chapter 2

Experimental Methodology & Metrics

The central ideas we present in this dissertation apply to a variety of instruction sets and underlying processor architectures. However, for the sake of experimental evaluation, we must select a specific execution platform, an implementation methodology, suitable workloads, and evaluation metrics which fairly capture the phenomena we seek to address. In the following sections, we describe our choices for each of these.

2.1 Execution-driven Simulation

We rely on processor and memory-system simulators to implement and evaluate our proposed architectures. Simulation has long been an accepted basis for architectural experimentation, in part due to the relative ease of simulation compared to the enormous expense of physically implementing modern microprocessors. While recent advances in prototyping technology such as FPGAs have brought experimental hardware into the reach of university courses and research projects, simulation-driven evaluation remains dominant in the field of computer architecture research.

Simulation allows us to evaluate experimental models at the level of detail with which we conceive them; a researcher need only implement the details they

consider significant, versus requiring full implementation at the lowest levels before measurements are possible. Simulation readily admits the use of oracle techniques, such as latency-free communication, or perfect knowledge of future behavior; this is useful for limit studies. Within a given research budget, simulation allows for more rapid exploration, and hence a wider variety of models and parameters may be considered, but with the pitfall than an incautious researcher may overlook important factors when constructing models, leading to unrealistic results.

There are a variety of approaches to simulation, offering different trade-offs in simulation speed, accuracy, and implementation complexity. In this dissertation we study multi-core processors featuring aggressive cores, designed to exploit instruction-level parallelism (ILP) in general-purpose computation. Our workloads are multi-threaded, featuring both cooperative multithreading (with communicating threads) and competitive multithreading (with independent threads). Both high-ILP execution and multithreading lead to substantial re-ordering of operations: individual cores aggressive overlap execution with outstanding memory operations, while concurrent execution allows re-ordering among different threads; these re-orderings have been shown [PRA97a, KT98] to significantly affect the resulting performance. Trace-based simulators are, in general, unable to capture the execution and network effects of dynamic reordering; given its importance in determining overall performance, we thus rely on execution-driven simulation in this dissertation.

We use two well-known processor and memory-system simulators: SMTSIM and RSIM.

2.1.1 SMTSIM

Our primary simulator – which we use for all research in this dissertation except for that in Chapter 4 – is a descendant of the original SMTSIM [Tul96]. We have extended the original simulator to support multiple cores, multiple levels of coherent private caches, and a MESI coherence protocol [PP84] atop a broadcast-based interconnect.

We use SMTSIM to model an out-of-order superscalar processor, includ-

ing speculative execution, executing unaltered native DEC OSF/1 Alpha binaries. This simulator models all typical sources of latency, including instruction execution, dependence stalls, cache misses, branch mispredictions, TLB misses, and stalls for coherence transfers. Conflicts are modeled for many types of resources, including renaming registers, issue queue entries, functional units, commit bandwidth, etc. Latency and bandwidth constraints are modeled for all cache, memory, and interconnect resources. Wrong-path behavior is included, execution down wrong paths between branch misprediction and branch misprediction recovery.

SMTSIM is highly configurable; we take advantage of this, using different processor and memory-system configurations for our different studies. In Chapter 3, we simulate a single-core two-way multithreaded processor with eight-wide issue, to demonstrate in detail the effect of memory stalls on co-scheduled threads. For Chapter 5, we configure SMTSIM as a four-core processor with a shared L2 cache and two threads per core, and then evaluate extending "multithreading-style" thread-switching across cores. In Chapter 6, we again model a four-core processor, but this time with a deeper private memory hierarchy and a single thread per core, in order to emphasize the ability of our working-set migration system. In each of these chapters, we provide additional details of the specific configurations used for that chapter.

2.1.2 RSIM

For the coherence research we present in Chapter 4, we used a derivative of RSIM [PRA97b], an event-driven multiprocessor and mesh-network simulator. While SMTSIM models a coherent multi-core processor, it simulates a bus-based interconnect, which is inappropriate for a processor with more than a handful of cores due to broadcast scalability problems.

In order to experiment atop a more scalable on-chip network, of the sort we envision for future chip multiprocessors, we sought a more capable network model for simulation. In order to experiment with the coherence implementation itself, we sought workloads which rely significantly on the coherence system; shared-memory parallel benchmarks in particular. RSIM provides both elements out-of-the-box: it

has a very detailed simulation of a 2-D mesh network, and support for benchmarks from several parallel benchmark suites.

RSIM has detailed models of execution cores, split L1 caches, private L2 caches, and a 2-D mesh network. However, RSIM was constructed to model a more traditional "cabinet-level" distributed shared-memory multiprocessor. We modified RSIM to simulate on-chip multiprocessing with an on-chip network; significant changes were required to model the proposed directory-based coherence implementations.

2.2 Metrics

We rely on a variety of metrics when performing our experimental evaluations and when presenting results. These include conventional metrics rooted in simulated execution time – speedup, request service latency, task response time – as well as simulated operation counts, e.g. total network message transfers. While simulated-time speedup is the de facto standard for evaluating experiments on single-thread workloads, most of the experiments we present in this dissertation involve parallel workloads, some with cooperating threads and some with competing threads.

For cooperatively-threaded parallel workloads, speedup is still an appropriate metric, so long as the workloads are evaluated over equivalent amounts of progress in the underlying task. This is the case for the coherence research we present in Chapter 4: we simulate the entire benchmark execution, so simulated times across experimental configurations are directly comparable. We report conventional speedup in that chapter.

2.2.1 Weighted Speedup

Workloads composed of competing independent threads pose a methodological challenge: unless great care is taken to ensure that every component thread makes the same amount of progress in each experiment, traditional metrics such as speedup or aggregate instructions-per-cycle (IPC) are easily skewed by the run-

time behavior of individual threads. In parallel execution, each experiment leads to a different interleaving of instructions from each thread, since the behavior of each thread influences (by way of resource contention) how it gets interleaved with others.

This problem is most dramatic when we evaluate policies which may bias execution against a particular thread or threads. For example, if we were to measure total IPC over a fixed number of system-wide commits, this metric would tend to unrealistically favor policies which prefer threads that exhibit higher IPC, by starving lower-IPC threads of execution resources for the duration of simulation. Any policy which favors high-IPC threads can boost the total IPC by increasing the contribution from those favored threads. Such a situation is unlikely to yield performance gain in a real system, however: while the IPC over a particular measurement interval might be higher, in a practical system low-IPC threads cannot be deferred forever. Eventually, the system would be left to execute a workload inordinately heavy in low-IPC threads, and the artificially-generated gains would disappear. (This problem is explored in more detail in previous work [ST00].)

Many of our experiments are simulated over time-scales which do not capture the reality of having to execute low-IPC threads eventually. This motivates the use of a metric which guides us toward better overall performance while taking into account the relative progress made by each thread in a given experiment. We evaluate multithreaded performance in terms of *weighted speedup* (WSU), as defined in [ST00], and used frequently in other studies of SMT and multi-core architectures:

$$\text{Weighted Speedup} \;=\; \sum_{i \in threads} \frac{IPC_{i,experimental}}{IPC_{i,standalone}} \tag{2.1}$$

In weighted speedup, each thread's experimental IPC is derated by its single-thread IPC, as measured in separate execution on a single core featuring the same hardware configuration, over the same dynamic instructions committed during the experimental run. To supply standalone IPC values, we simulate each benchmark in isolation – on each hardware configuration – logging simulated cycle counts to a database every ten thousand commits. When computing weighted

speedups, we query the database for each thread, retrieving the two logged points nearest the experimental commit count; we apply linear interpolation to estimate the baseline time to the given commit number.

We use weighted speedup in its original form (Equation 2.1) to report overall performance results for the Shared-Thread Multiprocessor in Chapter 5.

2.2.2 Normalized Weighted Speedup

We make two modifications to the standard weighted speedup formula when evaluating overall performance in the load-flushing experiments of Chapter 3:

- Instead of derating each thread by its stand-alone IPC, we derate each thread by the IPC it achieves on the baseline processor when run within the same mix of threads.
- We introduce an additional normalizing factor, dividing by the overall thread-count in each experiment.

The resulting equation we use to compute weighted speedup for the load-flushing research in Chapter 3 is:

$$\text{Normalized WSU} = \frac{1}{|threads|} \cdot \sum_{i \in threads} \frac{IPC_{i,experimental}}{IPC_{i,baseline}} \tag{2.2}$$

Despite the altered baseline, the spirit of the metric is the same: to make it impossible to quote artificial speedups by simply favoring high-IPC threads. We make this modification for two reasons: first, since Chapter 3 focuses on single-core SMT execution with a constant set of threads for each simulation, a given set of threads is always co-scheduled, which provides us a well-defined baseline for comparison; and second, when multiple threads are running slowly together, we benefit from any of them running faster, regardless of how they would perform when run independently.

The normalizing factor in Equation 2.2 affords us the convenience of plotting weighted speedups across workloads with different thread counts on a common axis; as it is constant for any particular workload, it does not introduce any additional bias between policies.

2.2.3 Interval Weighted Speedup

While we use weighted speedup (Equation 2.1) to report overall performance results for the Shared-Thread Multiprocessor of Chapter 5, some of the scheduling policies therein require online estimates of multithreaded performance. Weighted speedup as a goal function is not suitable for online decision-making, since it requires detailed single-threaded execution detail, $IPC_{i,single}$, which is not available at run-time.

This raises a new challenge: finding a suitable basis for evaluating IPC samples, such that all values are available at run-time, and which provides a reasonable baseline for the estimation of the changes in performance. IPC itself is a dangerous metric on which to base runtime optimization decisions, for the same reasons that it is a misleading indicator of overall multithreaded performance. We introduce a new metric, *interval weighted speedup*, an adaptation of traditional weighted speedup to our online scheduling environment.

To enable online evaluation, we use the aggregate IPC of each thread over the entire previous round of scheduling as the basis, in place of $IPC_{i,single}$, when evaluating the performance of the $IPC_{i,exper}$ samples taken during each sample interval. This strikes a balance, providing a measure of stability in values – particularly when comparing alternative schedules within a given round – yet still adapting over time to changes caused by earlier scheduling decisions.

We make one further modification to weighted speedup: given a direct application of the Equation 2.1 over shorter time scales, it is possible for individual quotients within the overall sum to generate very large outputs, e.g. when the basis IPC for an application is abnormally low due to execution conditions. It is important to prevent one such component from dominating the overall sum; while such a sample may cause only a short-term degradation when used for an individual scheduling decision, it can be disastrous when used with schedulers which aggregate data over multiple rounds of scheduling. We guard against this by compressing each thread's contribution to the sum from $[0, \infty)$ down to the range $[0, 4]$, using a smooth sigmoid function which is nearly linear in the range near 1.0 (where samples are most frequent).

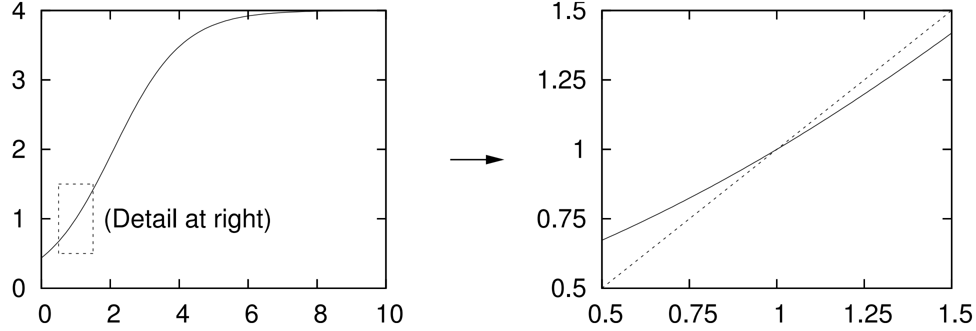


Figure 2.1: Interval Weighted speedup compression function, Equation 2.4.

The resulting equation, incorporating the alternate baseline and the compression function:

$$\text{Interval WSU} = \sum_{i \in threads} comp\Big(\frac{IPC_{i,sample}}{IPC_{i,basis}}\Big) \quad (2.3)$$

$$comp(x) = \frac{4.0}{1 + e^{(ln(3)+1-x)}} \quad (2.4)$$

Figure 2.1 shows the behavior of the range-compression function of Equation 2.4. Although we used it in our evaluation, the sigmoid function itself is not essential; linear scaling with clamping worked nearly as well.

To reiterate, as used within the Shared Thread Multiprocessor chapter, *weighted speedup* is the metric we use to evaluate and report performance, while *interval weighted speedup* is an online metric used internally to evaluate the effectiveness of scheduling decisions; the latter does not appear in any results.

2.2.4 Interval IPC & Post-migrate Speedup

For the thread-migration research of Chapter 6, we encounter another unconventional performance evaluation challenge. For that work, we wish to gauge the impact of architectural changes on performance in the immediate wake of migration operations, which we repeatedly induce. By triggering fairly infrequently, and capturing the immediate post-migration behavior at various intervals, we can capture both the short-term and long-term performance impact of each migration.

Given the relatively infrequent migrations used in our experimental setup, traditional whole-program metrics such as IPC are unsuitable for evaluating per-

formance over the shorter time scales we are most interested in, since those metrics will be dominated by the much longer periods between migrations. These long periods of undisturbed execution are desirable – they allow a given core to "warm up" to a thread's execution, providing a contrast for the next time that thread is migrated – but we do not want their statistics included in the post-migrate performance evaluation. Whole-program metrics also do not provide a useful way to analyze how the impact of a given migration varies across time scales.

We introduce a new performance metric for Chapter 6, the *interval IPC* of a migrating thread. After each migration, we concurrently measure the IPC over an exponential progression of commit intervals: we measure the time it takes to commit the 10^n instructions immediately following each migration operation, for $n \in \{0 \ldots 6\}$. We measure time from the first post-migrate fetch until the cycle of the 10^{n}th commit. This results in a vector of seven measured time values for each migration, and seven corresponding interval IPCs. For a particular experiment, we first compute an arithmetic mean for each interval, taken across all migrations of a single simulation, which results in seven mean interval IPCs per simulation. Rather than report interval IPCs directly, we compute ratios of these IPCs versus the interval IPCs of the same thread running on the baseline architecture and migrating at the same points in execution. We report these rations as either *post-migrate speedup* or *post-migrate slowdown*, depending on the sense of the ratio. We report means of these ratios, taken across our workload suite.

Chapter 3

Handling Long-Latency Loads on Simultaneous Multithreading Processors

3.1 Introduction

Simultaneous multithreading (SMT) [TEL95, TEE+96, YN95, HKN+92] is an architectural technique that allows a processor to issue instructions from multiple hardware contexts, or threads, to the functional units of a superscalar processor in the same cycle. It increases instruction-level parallelism available to the architecture by allowing the processor to exploit the natural parallelism between threads each cycle.

Simultaneous multithreading outperforms previous models of hardware multithreading primarily because it hides short latencies (which can often dominate performance on a uniprocessor) much more effectively. For example, neither fine-grain multithreaded architectures [ACC+90, LGH94], which context switch every cycle, nor coarse-grain multithreaded architectures [AKK+93, SBCvE90], which context switch only on long-latency operations, can hide the latency of a single-cycle integer add if there is not sufficient parallelism in the same thread.

What has not been shown previously is that an SMT processor does not

necessarily handle very long-latency operations as well as other models of multithreading. SMT typically benefits from giving threads complete access to all resources every cycle, but when a thread occupies resources without making progress, it can impede the progress of other threads. In a coarse-grain multithreaded architecture, by contrast, a stalled thread is completely evicted from the processor on a context switch; however, with SMT a stalled thread continues to hold instruction queue or reservation station space, and can even continue fetching instructions into the machine while it is stalled.

The ability of one SMT thread to uselessly degrade another, in the absence of any true dependence between them, is a pipeline-level impediment to our overall quest in this dissertation for on-chip parallel execution. In this chapter we demonstrate that an SMT processor can be throttled by a single thread with poor cache behavior; however, by identifying threads that become stalled, and limiting their use of machine resources, this problem can be eliminated. This provides not only significantly higher overall throughput, but also more predictable throughput, as threads with good cache behavior are much more insulated from co-scheduled threads with poor cache behavior.

3.2 The Impact of Long-latency Loads

We demonstrate the problem of long-latency loads using a simple experiment, with results depicted in Figure 3.1. For six combinations of two threads (the actual workloads and experimental configuration are described in Section 3.4), the figure shows three results: the IPC of each of the two threads running alone, and of the two threads running together on the SMT processor. In each case the light bars represent memory-intensive benchmarks, and the gray bars represents applications with good cache behavior.

This example shows that a thread exhibiting poor cache performance can become a significant inhibitor to another thread with good cache behavior. There are two factors that allow an application with poor cache locality to cripple co-scheduled applications. First, an application that regularly sweeps through the

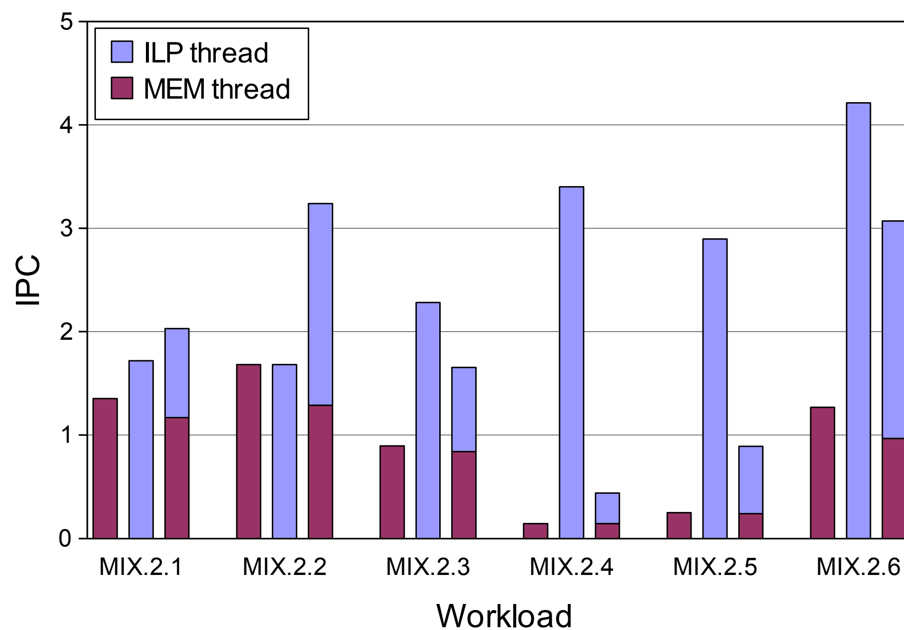


Figure 3.1: The performance of several two-thread mixes of memory-bound and ILP-bound applications. The stacked bars represent two-thread runs, while the single bars depict single-thread runs of the two component benchmarks of each group.

shared cache will evict data from the other applications, degrading their cache hit rates. Second, the memory-bound application can acquire and hold critical execution resources while it is not making progress due to long-latency memory operations, degrading every thread's performance. In this chapter, we focus on the latter problem.

Few applications contain sufficient parallelism to hide long memory operations (e.g., more than a dozen cycles). While multithreading allows other threads to hide that latency, if the stalled thread fills the instruction queue with waiting instructions, it shrinks the window available for the other threads to find instructions to issue. Thus, when parallelism is most needed – when one or more threads are no longer contributing to the instruction flow – fewer resources are available to expose that parallelism.

This is most clearly demonstrated for the instruction queues by the MIX.2.5 workload, for which the integer queue is on average 97% occupied when at least one L2 miss is outstanding, but only 62% occupied at other times. Besides instruction queues, other resources that are potentially held or used by a thread stalled waiting for a long memory operation include renaming registers and fetch/decode bandwidth. We will demonstrate that contention for shared resources is by far the dominant factor causing the poor performance shown in Figure 3.1.

3.3 Related Work

Simultaneous multithreading [TEL95, TEE+96, YN95, HKN+92] is an architectural technique that allows a processor to issue instructions from multiple hardware contexts, or threads, to the functional units of a superscalar architecture each cycle. The experiments in this chapter build upon the SMT architecture presented in [TEE+96]; previous SMT research has not exposed the problem (or solutions) examined here. One important reason for that has been the inability of pre-2000 instantiations of the SPEC benchmark suite to put significant pressure on a reasonable cache hierarchy.

Less aggressive models of multithreading are less prone to such problems.

Coarse-grain multithreading [AKK+93, SBCvE90] is aimed *only* at the long-latency load problem, and makes no attempt to address any other machine latency. Because coarse-grain architectures allow only one thread to have access to execution resources at any time, they alway flush stalled threads completely from the machine. Fine-grain multithreading [ACC+90, LGH94] could potentially have shared scheduling resources which exhibit this problem, depending on the architecture. However, these architectures (e.g., the Cray/Tera MTA [ACC+90]) have traditionally been coupled with in-order execution, where scheduling windows only need to keep a few instructions per thread visible.

We ignore the latency of synchronization operations (the other source of long and non-deterministic latencies) in this analysis. Tullsen, et al.[TLEL99] have shown the advantage of a synchronization primitive which both blocks and flushes a thread from the queue when it fails to acquire a lock; however, the performance implications of not flushing were not investigated, and that paper gives no indication that a similar technique is necessary for loads.

Previous work on the interaction of SMT processors and the cache hierarchy has focused on cache size and organization (Nemirovsky and Yamamoto [NY98]), bandwidth limitations (Hily and Seznec [HS98]), or cache partitioning [TEL95].

Cache prefetching [CB95, MLG92] attacks the long-latency load problem in a different way, seeking to eliminate the latency itself. Recent work in prefetching targets multithreaded processors specifically, using idle hardware contexts to initiate prefetching. These include Collins, et al. [CWT+01], Luk [Luk01], and Zilles and Sohi [ZS01].

Following the original publication of the work in this chapter, numerous later research projects have noted the impact of the resource-hoarding problems we have identified here, and have expanded upon our solution in different ways. El-Moursy et al. [EMA03] introduce several fetch-gating policies which seek to avoid the hoarding of instruction queue resources that we identify and react to here, without the cost of flushing. They evaluate several dynamic fetch-gating schemes, driven by outstanding load counts and predictions thereof, with an eye toward reducing queue occupancy with minimal performance degradation.

Balanced Multithreading (BMT) [TKTC04] takes advantage of the lull in forward progress following a stall for off-chip memory access, using it as an opportunity to replace the stalling thread with one that is ready for execution. By rotating additional threads into the processor at opportune times, BMT augments the ILP visible to an SMT execution engine by bringing in “fresh” threads in the face of memory stalls, instead of simply flushing instructions and blocking fetch as we consider here.

Cazorla et al. [CRVF04b] evaluate several of the policies in this chapter along with several from [EMA03]. They consider in more detail the effects of both overlapping and sequential memory stalls from multiple threads, and demonstrate several refinements to our policies which result in improved throughput and fairness. In further work [CRVF04a], they introduce a detailed feedback-directed hardware resource management policy which dynamically partitions the instruction queues and register files among threads to avoid the hoarding we characterize in this chapter.

Eyerman et al. [EE07] introduce a nuanced approach which predicts, for each long-latency load event, the amount of memory-level parallelism (MLP) available at that point in the program. In addition to flushing and stalling, their policy can also allow a thread to proceed a short distance past each memory stall in order to expose additional predicted MLP before corrective action is taken. In further work [EE09], they introduce a detailed cycle-accounting mechanism which explicitly accounts for cycles spent stalled for memory, as well as those dominated by the execution of co-scheduled threads; this provides a online quantification of the inter-thread resource conflicts we consider only after the fact.

3.4 Methodology

Table 3.1 summarizes the benchmarks used in our simulations. All benchmarks are taken from the SPEC2000 suite and use the reference data sets. Six are memory-intensive applications: those which, in our system, experience between 0.02 and 0.12 L2 cache misses per instruction on average, over the simulated por-

Table 3.1: The single-threaded benchmarks used in this chapter, along with the data set and the number of instructions emulated before beginning measured simulation.

Benchmark	*Input*	*Fast-forward* ($\times 10^9$)
Memory-intensive, "MEM"		
ammp	ref	1.7
applu	ref	0.7
art	c756hel.in (ref)	0.2
mcf	ref	1.3
swim	ref	0.5
twolf	ref	1.0
ILP-intensive, "ILP"		
apsi	ref	0.8
eon	cook (ref)	1.0
fma	ref	0.1
gcc	integrate.i (ref)	0.5
gzip	log (ref)	0.1
vortex	ref	0.5

tion of the code. The other six benchmarks are taken from the remainder of the suite and have lower miss rates, and hence higher inherent ILP. Table 3.2 lists the multithreaded workloads used in our simulations. All of the simulations in this chapter either contain threads all from the first group (the MEM workloads in Table 3.2), all from the second group (ILP), or an equal mix from each (MIX). Most of this chapter focuses on the MIX results; however, the other results are included to demonstrate the universality of the problem.

We simulate execution with a derivative of SMTSIM [Tul96], as introduced in Chapter 2. The baseline processor configuration used for most simulations is shown in Table 3.3. The instruction queues for our eight-wide processor are roughly twice the size of the four-issue Alpha 21264 (15 FP and 20 integer entries) [Com00]. In addition, the 21264 queues cannot typically remain completely full due to the implemented queue-add mechanism, a constraint we do not model with our queues. These instruction queues, as on the 21264, remove instructions upon issue, and thus can be much smaller than, for example, a register update unit [SV87] which holds instructions until retirement. Section 3.9 also investigates larger instruction

Table 3.2: The multi-threaded workloads evaluated.

ID	*Component Benchmarks*
ILP.2.1	apsi, eon
ILP.2.2	fma3d, gcc
ILP.2.3	gzip, vortex
ILP.4.1	apsi, eon, fma3d, gcc
ILP.4.2	apsi, eon, gzip, vortex
ILP.4.3	fma3d, gcc, gzip, vortex
MEM.2.1	applu, ammp
MEM.2.2	art, mcf
MEM.2.3	swim, twolf
MEM.4.1	ammp, applu, art, mcf
MEM.4.2	art, mcf, swim, twolf
MEM.4.3	ammp, applu, swim, twolf
MIX.2.1	applu, vortex
MIX.2.2	art, gzip
MIX.2.3	swim, gcc
MIX.2.4	ammp, fma3d
MIX.2.5	mcf, eon
MIX.2.6	twolf, apsi
MIX.4.1	ammp, applu, apsi, eon
MIX.4.2	art, mcf, fma3d, gcc
MIX.4.3	swim, twolf, gzip, vortex

Table 3.3: Processor configuration.

Parameter	*Value*
Fetch width	8 instructions per cycle
Fetch policy	ICOUNT.2.8 [TEE+96]
Pipeline depth	8 stages
Min branch misprediction penalty	6 cycles
Branch predictor	2K gshare
Branch Target Buffer	256 entry, 4-way associative
Active List Entries	256 per thread
Functional Units	6 Integer (4 also load/store), 3 FP
Instruction Queues	64 entries (32 integer, 32 FP)
Registers For Renaming	100 integer, 100 FP
Inst Cache	64 KiB, 2-way, 64-byte lines
Data Cache	64 KiB, 2-way, 64-byte lines
L2 Cache	512 KiB, 2-way, 64-byte lines
L3 Cache	4 MiB, 2-way, 64-byte lines
Latency from previous level (with no contention)	L2 10 cycles, L3 20 cycles Memory 100 cycles

queues.

The policies of the SMT fetch unit have a significant impact on our results. Our baseline configuration uses the ICOUNT.2.8 mechanism from [TEE+96]. The ICOUNT mechanism fetches instructions from the thread or threads least represented in the pre-execute pipeline stages. This mechanism already goes a long way towards preventing a stalled thread from filling the instruction queue (Section 3.9 shows how much worse the load problem becomes without ICOUNT), but we show that it does not completely solve the problem. In particular, if the processor is allowed to fetch from multiple threads per cycle, it becomes more likely a stalled thread (while not of the highest priority) can continue to dribble in new instructions. Our baseline fetch policy (ICOUNT.2.8) does just that, fetching eight instructions total from two threads. Section 3.9 also looks at fetch policies that only fetch from one thread per cycle, demonstrating that the problem of long-latency loads persists even in that scenario.

3.5 Metrics

As discussed in Chapter 2, this type of study represents a methodological challenge in accurately reporting performance results. In multi-threaded execution, every run consists of a different mix of instructions from each thread, making aggregate IPC (instructions per cycle) a questionable metric. We evaluate performance in this chapter using a modified version of weighted speedup which we dub *normalized weighted speedup*, or *normalized WSU*; see Section 2.2 for a discussion of weighted speedup, specifically Section 2.2.2 and Equation 2.2 for details of the primary metric used in this chapter.

In addition to the normalized weighted speedup metric used throughout this chapter, in Section 3.8 we also follow the lead of [ST00] by using open system experiments, measuring mean job response time to assess the benefit of these optimizations in a dynamic system with jobs entering and leaving the processor over time.

3.6 Detecting and Handling Long-latency Loads

This section details our primary mechanisms for (1) identifying that a thread or threads are likely stalled, and (2) freeing resources associated with those threads.

Identifying stalled threads in most cases operates on two assumptions: that only loads can incur sufficient latency to require this type of drastic action, and that if a load takes long enough, it is almost certain to stall the thread. (See [TLEL99] for a study of synchronization mechanisms on SMT, which is the other potential source of long thread stalls). Note that in an out-of-order processor, the notion of a “stalled” thread is much fuzzier than in an in-order processor. In an out-of-order processor, only those instructions dependent on the load will get stuck in the instruction queue, but if the memory latency is long enough, eventually the thread will run out of instructions that are independent of the load (or the active list/reorder buffer will fill with the stalled load at the head). At that point, the thread has gone from partially stalled to fully stalled.

Freeing resources requires removing instructions from the processor. In most of our experiments we assume that the processor uses the exact same flushing mechanism that is used for a branch misprediction, which can flush part of a thread starting at a given instruction. Such a flush frees renaming registers and instruction queue entries.

We make the following assumptions in all of the experiments in this chapter. First, that we always attempt to leave one thread running; we do not flush or block a thread if all others have already been flushed or blocked. Second, that any thread which has been flushed is also blocked from further fetching until the load returns from memory. Third, that the processor core receives little advance warning that a load has returned from memory. In our case, the two-cycle cache fill time allows us to possibly begin fetching one cycle before the load data is available (roughly four cycles too late to get the first instructions in place to use the returned data immediately). A mechanism that accurately predicted the return of a load, or received that information from the memory subsystem early, would allow the thread to bring the instruction stream back into the scheduling window more quickly, achieving higher performance than shown here at the cost of some complexity.

We will examine two mechanisms for identifying long-latency loads. *Trigger on miss* assumes we get a signal from the L2 cache on each miss, and that the processor can attribute that miss to a particular thread and instruction. We also assume that a TLB miss triggers a flush, on the assumption that most TLB misses will incur expensive accesses to fill the TLB and will often also result in cache misses after the TLB is reloaded. If the TLB miss is handled by software in the same thread context, the processor must not flush until after the miss is handled. A simpler mechanism, *trigger on delay*, just initiates action when an instruction has been in the load queue more than L cycles after the load was first executed. For most of our experiments, L is 15. That is more than the L2 hit time (10 cycles), plus a few more cycles to account for the non-determinism caused by bank conflicts and bus conflicts.

Figure 3.2 shows just the latter mechanism (T15: trigger a flush after 15

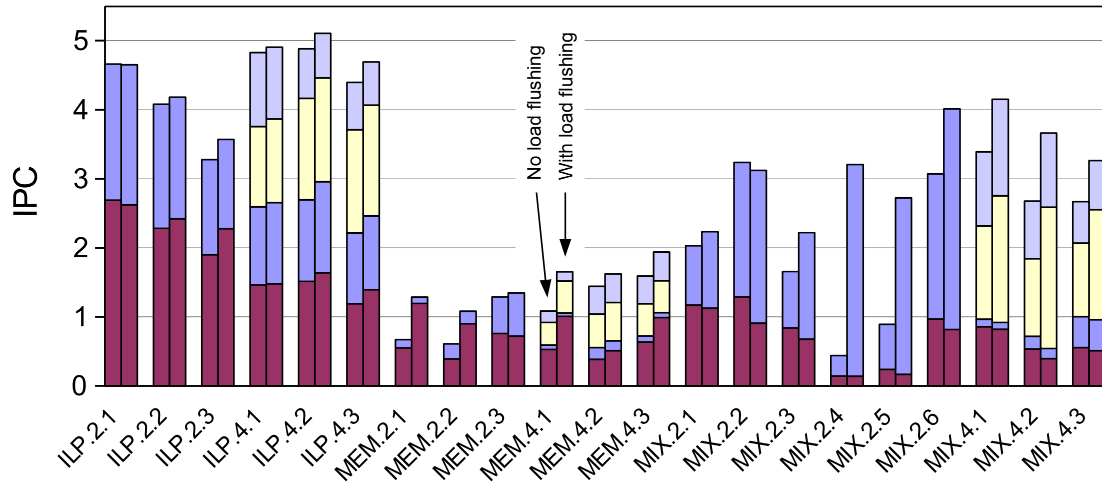


Figure 3.2: The instruction throughput of all workloads with a simple mechanism for flushing threads waiting for long-latency loads. The contributions of each thread to the total IPC are shown by the segmented bars. For the MIX results, the memory-intensive benchmarks are those closest to the bottom of the graph. In each pair, the left bar uses no flushing, and the right bar uses the *T15* policy.

cycles) compared to regular execution (no flushing) for all combinations of workloads. This graph plots instructions per cycle, for each thread, and shows that the performance gains are mostly coming from the non-memory threads being allowed to run unencumbered by the memory threads, with the memory threads suffering slightly. Because of the difficulties with using IPC as a performance metric, as discussed in Section 3.4, further graphs will show normalized weighted speedup results instead; however, Figure 3.2 does give insight into how the speedups are achieved. This figure also shows that long-latency load flushing is effective even when the threads are uniform: all memory-bound or all ILP-bound. The average normalized weighted speedup for the ILP workloads is 1.03 and for the MEM workloads is 1.25. Subsequent results will focus on the mixed workloads, however.

Figure 3.3 shows more mechanisms for identifying long-latency loads, including TM (trigger on L2 miss), T5, T15, and T25 (trigger a flush after a load becomes 5, 15, or 25 cycles old), and T15S (S for selective: only flush if some resource is exhausted, such as instruction queue entries or renaming registers). T5 flushes after L1 misses and T15 after L2 misses. T25 is an interesting data point,

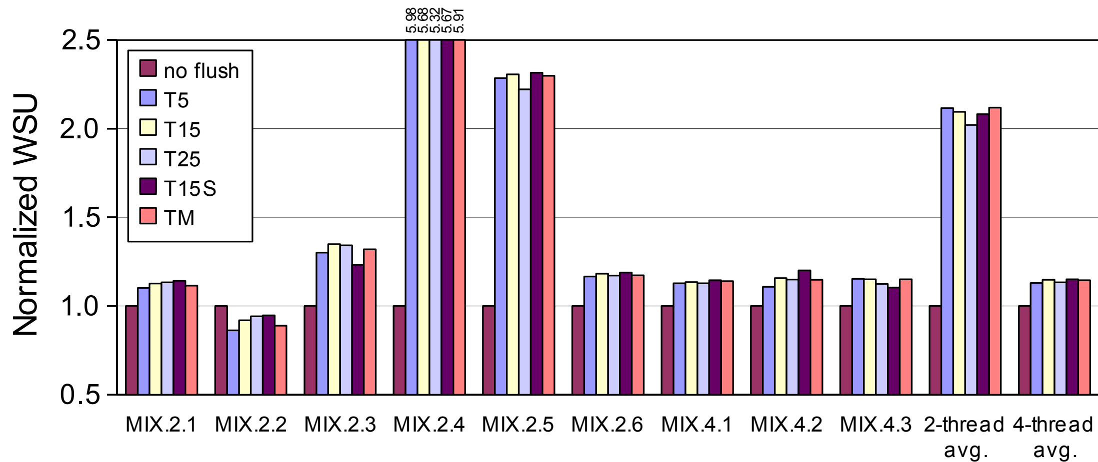


Figure 3.3: The normalized weighted speedup of flushing after long loads, comparing several mechanisms for identifying long-latency loads.

because an L3 miss takes at least 30 cycles; it will identify the same misses as T15, but identify them later.

The results are both clear and mixed. It is clear that flushing after loads is important, but the best method of triggering a flush varies by workload. Triggering after 15 cycles and triggering after a cache miss are consistently good. The selective flush is best in several cases, but also performs poorly in other cases. When it performs poorly, it is because a flush is often inevitable (especially since the stalled thread can still fetch instructions to fill the queue); then, being selective only delays the flush until some harm has actually been done and allows the doomed thread to utilize precious fetch bandwidth in the meantime. In other cases, being conservative about flushing (see both T15S and T25) pays off. This is not so much because it reduces the number of flushes, but because it allows more loads from the doomed thread to get into the memory subsystem before the flush. Thus, performance is best when we can find the right balance between the need to purge a memory-stalled thread from the machine, and the need to exploit memory-level parallelism within the memory-bound thread. That balance point varies among the workloads displayed here.

When there is little contention for the shared resources, flushing after loads can hinder one thread without aiding the other(s); in our simulations, we only see

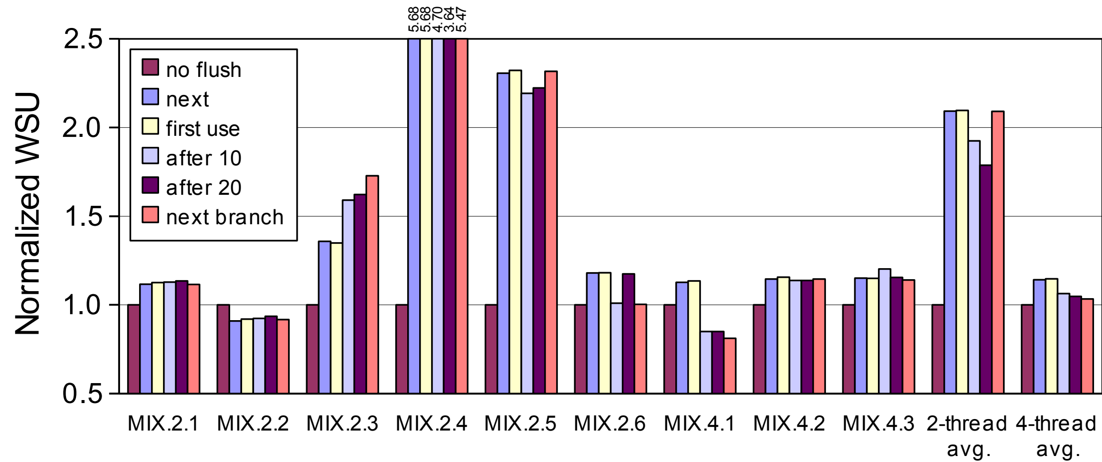


Figure 3.4: The normalized weighted speedup of several techniques for selecting the flush point, after a long-latency load triggers a flush.

that in the MIX.2.2 workload.

The average normalized weighted speedup for load flushing in this figure is over 2.0 when two threads are active, and 1.13–1.15 for four threads. The two-thread case is extreme because it is easy for a stalled thread to eventually take over the processor when we are fetching from two threads every cycle. However, the four-thread case shows that even when that effect is no longer dominant, all threads still suffer from the "equal portion" of the machine which is held by a stalled thread.

Once a thread is identified as stalled and selected for flushing, the processor must choose an instruction to flush forward from; we examine several schemes. *Next* flushes beginning with the next instruction after the load. *First use* flushes at the first use of the loaded data. *After 10* and *after 20* flush beginning 10 or 20 instructions beyond the load. *Next branch* flushes at the next branch. This mechanism simplifies load flushing on processors that checkpoint only at branches. The Alpha 21264 and 21364 checkpoint all instructions, and would have more flexibility in choosing a flush point. The results presented so far have all used the flush after *first use* technique. Figure 3.4 shows the performance of different flush point selection techniques; the T15 load identification scheme was used for these experiments.

These results show some definite trends. When the load problem is most drastic (in the two-thread workloads, particularly MIX.2.4), it is critical to flush as close to the problem as possible, to minimize the held resources. In those cases, flushing on next, first-use, and (sometimes) first-branch all fit that bill. When the load problem is less critical, sometimes being more liberal about where to flush can actually help. However, because there is so much more to gain when the load problem is most evident, the average results are dominated by mechanisms that flush close to the load.

Further results in this chapter will use the *trigger after 15 cycles* scheme to identify long loads, and will flush beginning with the *first use*. This policy will be simply denoted as T15.

The results in this section demonstrate that flushing after a long-latency load can be extremely effective in allowing non-stalled threads to make the best use of the execution resources. Flushing a thread is a fairly drastic action to take on an SMT processor, but appears warranted across a wide variety of workloads. Among the questions examined in the next section is the effectiveness of less drastic measures to solve the long-load problem.

3.7 Alternate Flush Mechanisms

This section investigates a wider range of mechanisms to free execution resources during long-latency loads. It seeks to answer these questions: (1) is the complexity and performance cost of flushing on long-latency loads necessary, and (2) what further benefits might be gained from more complex mechanisms?

One simpler alternative would be to only moderate fetching. That is, do not flush, but immediately stop fetching from a thread experiencing an L2 miss. This does not clear space occupied by the stalled thread, but prevents it from taking more than its share while it is not making progress. This is the *stall fetch* scheme of Figure 3.5.

Alternatively, we could make it more difficult for a thread to ever occupy too much of the shared queue. Certainly, a statically partitioned queue does not

experience the load problem. However, that is a dear price to pay, sacrificing the most efficient use of the queue at other times, especially when not all contexts are active. A middle ground solution, however, would be a hard limit on how many instructions a single thread could have in the pre-execute portion of the pipeline (presumably this limit could be turned off when executing in single-thread mode). We experimented with several different limits, and the best performer appears as *pseudo-static* in Figure 3.5. For that policy, no thread is allowed to fetch a new block when it has more than 20 instructions in the queue stage or earlier if we have two threads active, or more than 15 instructions if there are four threads active.

More complex mechanisms are also possible. Only slightly more complex is a hybrid of T15S and *stall fetch*. This mechanism stops fetching as soon as a long-latency load is detected, but only flushes if a resource is exhausted. Stopping fetch for the offending thread immediately increases the chances that no resource will be exhausted and no flush will be necessary, if all other threads' queue pressure is light. This policy is labeled T15SF: stall, then flush.

The last scheme examined adds *stall buffers* to the processor. This is intended to eliminate the delay in getting instructions back into the instruction queues. Instructions that belong to a thread that is stalled, and are themselves not ready to issue, will be issued (subject to issue bandwidth constraints) to the stall buffer using the normal issue mechanism. When the load completes, instructions will be dispatched to the instruction queue (temporarily over-riding the rename-issue path), again subject to normal dispatch bandwidth. This eliminates the delay in resuming the stalled thread, allowing it to make more progress as parallelism allows. Typically, only the load-dependent instructions will go into the stall buffer, so even a small buffer can allow many independent instructions after the load to execute and avoid unnecessary squashing. Once the stall buffer fills, the thread is flushed starting with the instruction that found the buffer full.

Figure 3.5 shows the results. Just stalling fetch improves performance over no flushing, but falls far short of the other solutions. The pseudo-statically partitioned queue also falls short due to the inherent inefficiencies of placing artificial limits on threads' use of the queues. The stall and flush mechanism (T15SF) is a

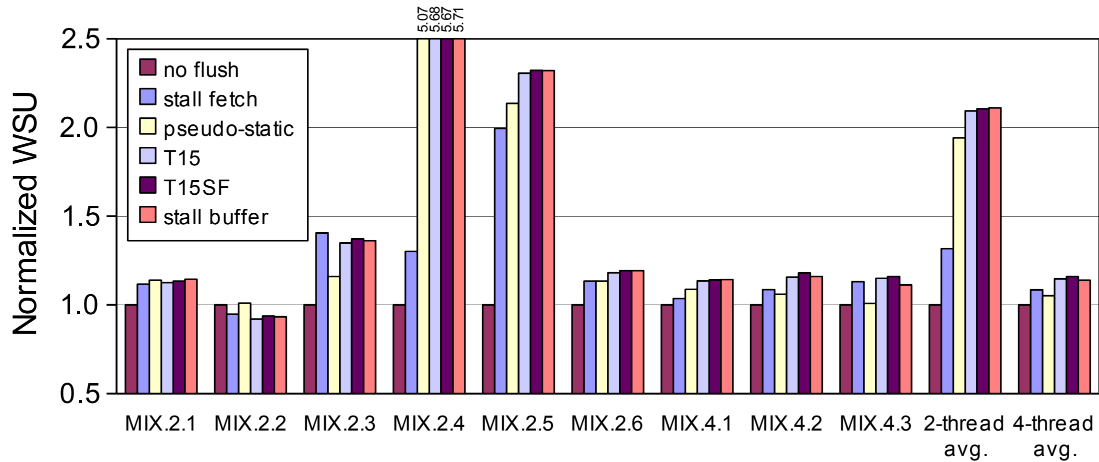


Figure 3.5: The performance of several alternatives to our baseline (T15) load flushing mechanism.

small change to our previous scheme and does show an improvement over that approach (T15). The performance of the stall buffer is disappointing. It only solves half the problem: while relieving the instruction queue, it puts more pressure on the renaming registers and increases those conflicts.

The T15 mechanism strikes a good balance between implementation complexity and performance, on a wide variety of workloads.

3.8 Response Time Experiments

While our use of normalized weighted speedup addresses most of the methodological concerns with this research, there are some questions which we can only answer definitively by comprehensively modeling of an open system, with jobs arriving and departing. For example, one possible issue is whether even normalized weighted speedup appropriately accounts for the fact that a continued bias against the slow threads may mean that they stay in the system longer, causing problems for more threads. In fact, we will show that this isn't the case, which is not obvious from the previous experiments.

In this experiment, we modified the simulator to allow jobs to enter the simulated system at various intervals, and run for a predetermined number of

instructions. Because the runtime intervals were, by necessity, much less than the actual run times of these programs, we still fast-forwarded each job to an interesting portion of execution before entering it into the system. Since the MEM threads run more slowly, we used a mix of two ILP threads to every MEM thread; this yielded a fairly even mix of jobs in the processor at any point in time. We ran eighteen total jobs in each simulation, with MEM jobs run once each, and ILP jobs run twice each. In such an experiment, the only useful measure of performance is average response time (execution time), since the instruction throughput is for the most part a function of the schedule rather than the architecture. The mean response times were calculated using the geometric mean due to the wide disparity in response times for different jobs. In these simulations, all jobs execute for 300 million instructions, then exit the system. In the *light load* experiment, jobs arrive every 200 million cycles, in *medium load*, they arrive every 150 million cycles, and in *heavy load*, they arrive every 100 million cycles. For the baseline cases, there were on average 2.9, 3.4, and 4.5 jobs in the system for the light, medium, and heavy loads, respectively.

Figure 3.6 presents the results of the three experiments. For each experiment, the ILP and MEM thread response times are shown computed separately as well as combined. The results show dramatic decreases in response time through the use of load flushing. Surprisingly, these decreases are not restricted to the ILP threads: the MEM threads gain very significantly as well, despite being the target of bias. The gains are significant with both light loads, where the average number of jobs in the system is closer to the worst-case of two threads, and with heavy loads, where the average number of jobs is much higher.

These results expose two phenomena not shown in the previous sections. First, when one thread inhibits the progress of other threads, it only causes further queueing delays as more jobs enter the system. Conversely, if a thread can accelerate a co-scheduled job's exit from the system, it gains a larger share later to accelerate its own progress. This is the source of the high speedup for the MEM threads. With the medium-load workload, load flushing reduced the average number of jobs in the system from 3.4 to 2.5, which benefited every job.

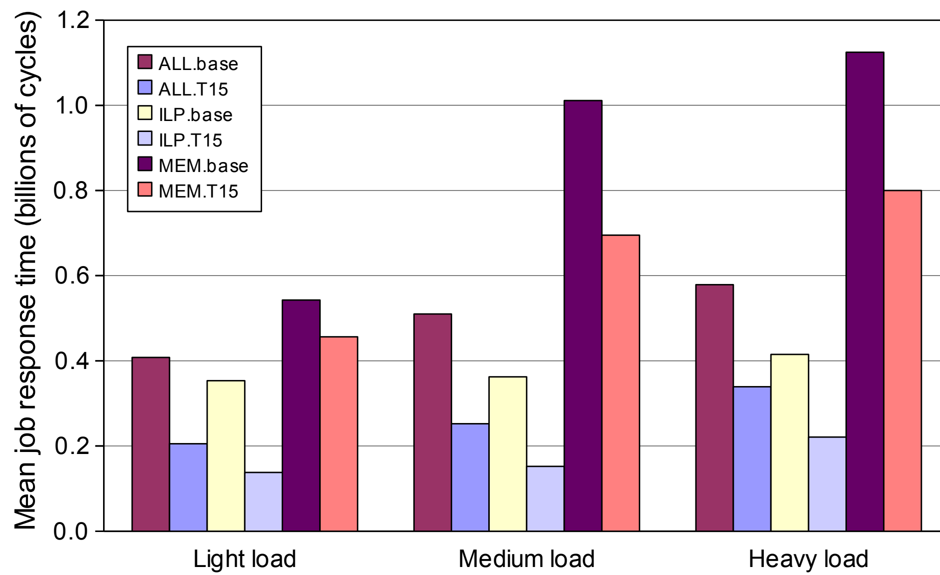


Figure 3.6: The mean response times of jobs in open system experiments. Each group of bars corresponds to two simulations: one in the baseline configuration, and one using the *T15* policy (trigger 15 cycles after load issue, flushing instructions from first-use). Geometric means are shown for *ALL* jobs in each simulation, as well as for the *MEM* and *ILP* subsets of threads in those same simulations.

The second phenomenon which degraded the performance of the no-flushing results was the exaggeration of the two-thread problem seen in earlier results. Since this experiment saw anywhere from zero to eight threads in the system at any one time, we would hope that it would not spend too much time in the disastrous two-thread scenario. However, just the opposite took place, as the poor performance of the two-thread case made it something of a local minimum that the system constantly returned to, for some of the experiments. When more than two threads were in the system, throughput would improve, returning the system more quickly to dual execution. Similarly, the system was unlikely to move to single-thread execution if two-thread throughput was low. Thus we see that the poor dual-thread performance highlighted by previous sections will take a much larger toll on overall throughput than might be expected statistically — if it is not eliminated using the techniques outlined here.

3.9 Generality of the Load Problem

The benefit from flushing after long loads will vary with the parameters of the architecture. This section shows how the technique works under different assumptions about fetch policies and instruction queue size. By varying those parameters which most impact the applicability of this mechanism, this section demonstrates that these techniques solve a real problem that exists across a wide range of assumed architectures.

The effectiveness of, and necessity for, flushing after loads will necessarily vary with cache sizes and cache latency. We do not explore this space here, however, because we will be able to rely on two constants for the foreseeable future that will ensure the continued and increasing need for this technique: there will always be memory-bound applications, and memory latencies will continue to grow.

The ICOUNT fetch policy attempts to prevent a thread from ever taking more than its share of the processor. One reason that threads are able to circumvent it in these experiments is that, with a fetch policy that allows two threads to fetch concurrently, a thread not of the highest priority is still able to add instruc-

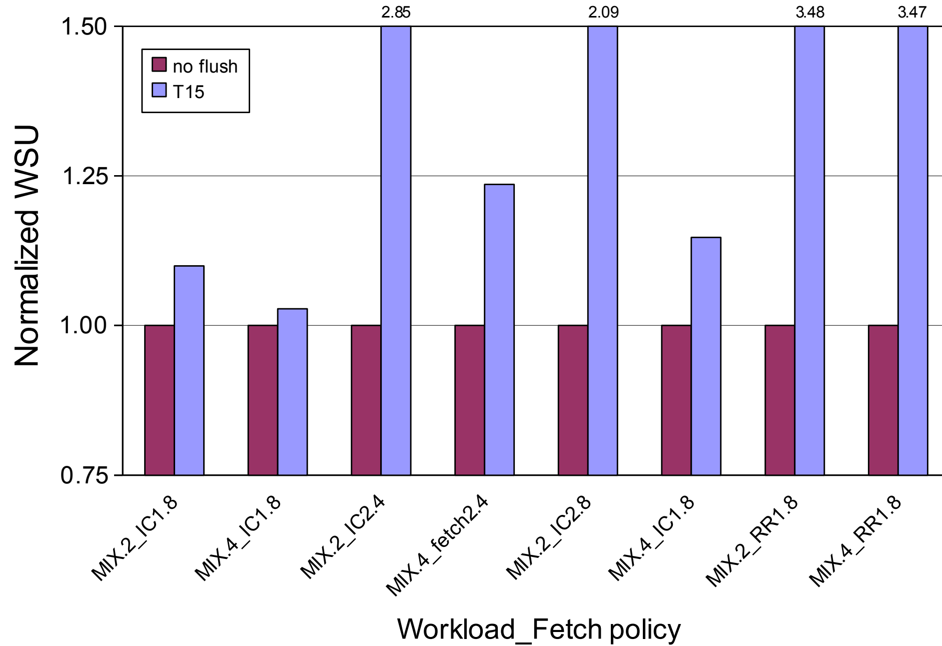


Figure 3.7: The normalized weighted speedup of load flushing for different SMT fetch policies.

tions. Figure 3.7 examines the speedups achieved with various fetch policies, using the terminology from [TEE+96]. The ICOUNT.1.8 policy fetches up to eight instructions from a single thread each cycle. With that scheme a thread cannot fetch more instructions unless it is the least represented thread that is ready to fetch. The ICOUNT.2.4 policy fetches four instructions from each of two threads for a maximum of eight. The ICOUNT.2.8 policy fetches up to eight instructions from the highest-priority (least-represented) thread, and if fewer than eight instructions are available from the first thread, attempts to fill the remaining space in the fetch window with instructions from the second-highest priority thread. ICOUNT.2.8 is the baseline policy used throughout this chapter. The RR.1.8 uses round-robin priority for fetch rather than ICOUNT.

Figure 3.7 shows that the ICOUNT.1.8 fetch policy goes a long way toward solving the problem, but it is not sufficient: there is still a significant gain for flushing, especially with two threads. This is because even if the machine is successful at preventing a thread from occupying more than an equal portion of the processor, it still loses that equal portion of the instruction window to find parallelism in

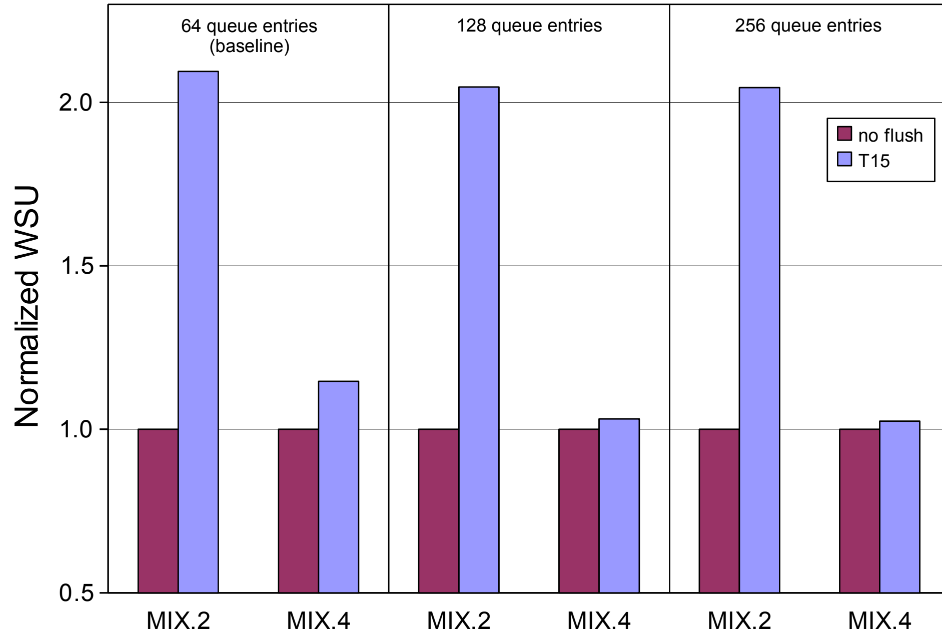


Figure 3.8: The normalized weighted speedup of load flushing for different instruction queue sizes. MIX.2.* is the average for all six MIX.2 workloads, and MIX.4.* is the average for the MIX.4 workloads.

other threads. Fetching from a single thread is not a panacea, anyway, because the ICOUNT.1.8 policy also has a performance cost not seen in this graph (because the speedups are normalized to different baselines). With load flushing applied, the ICOUNT.1.8 result is 9% slower than the ICOUNT.2.8 result with four threads for the MIX experiments, a result that confirms those in [TEE+96]. The ICOUNT.2.4 results show even greater gains than the ICOUNT.2.8 results. This comes from the fact that the ICOUNT.2.4 scheme gives the top two threads equal access to fetch, unlike the ICOUNT.2.8 scheme. With round-robin instruction fetching, we see to what extent the ICOUNT scheme was protecting the processor from load stalls. With round-robin fetching (the RR.1.8 results), flushing after loads is absolutely essential to good performance, regardless of the number of threads.

The size of the instruction scheduling window (in this case, the instruction queues) will also impact how easy it is for a context to monopolize the structure. Figure 3.8 shows the performance of load flushing for two larger queue sizes (in addition to the previous results for 64 total queue entries). As the queues become

larger, the processor does become more tolerant of long-latency loads when sufficient thread parallelism exists. With fewer threads, however, it only takes the stalled thread a little longer to take over the queue, regardless of size.

Another factor that would also affect these results is the presence of other memory latency tolerance techniques, such as memory prefetching (either hardware or software). While techniques such as these are less important on a multithreaded processor, it can be expected that they will be available. In fact, some research exploits the existence of threads to create prefetching engines [CWT+01, Luk01, ZS01].

We expect this technique to coexist efficiently with – and in some cases supplant – prefetching, in terms of performance. No current prefetchers provide full coverage of cache misses for all important applications; so, a prefetcher could be used to boost the performance of a particular memory-intensive benchmark, while a load-flushing technique would still protect system throughput when the prefetcher fails. A hardware prefetcher for a processor that included this load-flushing mechanism would have the luxury of focusing on achieving high accuracy, because high coverage will be less important.

Some environments, however, are inappropriate for prefetching. When memory bandwidth is limited or heavily shared [TE93], the extra bandwidth generated by prefetching might be unacceptable, but load-flushing incurs no such cost. The extra bandwidth required for prefetching is also undesirable for low-power applications; however, the cost of re-execution after a flush may also be unacceptable, in which case stalling fetch or a static or pseudo-static partitioning of the instruction queues might become more desirable.

3.10 Summary

A thread with a high concentration of long-latency cache misses can reduce the throughput of a co-scheduled thread by as much as a factor of ten. This happens when the memory-bound thread constantly fills the instruction scheduling window with instructions that cannot be issued due to dependence on these long-

latency operations. The co-scheduled thread cannot get enough instructions into the processor to expose the parallelism needed to hide the latency of the memory operation. Thus, we lose the primary advantage of multithreading.

In this chapter, we have addressed this problem by forcing a thread waiting for a long-latency load to give up resources, using the same mechanism used for branch mispredictions, and allowing the thread to resume fetching once the load returns from memory. This technique achieves a 15% speedup with four threads active, and more than doubles the throughput with two threads active. Response time experiments show that under various load levels the average response time is cut by about a factor of two, including a significant reduction even for the memory-bound jobs our techniques bias against.

We have also shown that less aggressive techniques (e.g. just stalling fetch, or limiting full access to the instruction queues) can help, but do not provide the speedups achieved by flushing. More aggressive techniques, such as providing dedicated buffers for holding stalled instructions, do provide some further gains, but may not justify the additional cost.

Acknowledgements

The work in this chapter was funded in part by NSF CAREER grant MIP-9701708, a Charles Lee Powell faculty fellowship, and equipment grants from Compaq Computer Corporation.

This chapter contains material from “Handling Long-Latency Loads in a Simultaneous Multithreading Processor”, by Dean M. Tullsen and Jeffery A. Brown, which appears in *Proceedings of the 34th annual International Symposium on Microarchitecture (MICRO)*. The dissertation author was the secondary investigator and author of this paper. The material in this chapter is copyright ©2001 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 4

Coherence Protocol Design for Chip Multiprocessors

4.1 Introduction

Multi-core architectures, particularly chip multiprocessors (CMPs), are becoming increasingly popular as a means to enhance the throughput and power efficiency of processors. While initial implementations of multi-core technology from the general purpose processor industry contained two to four cores [MMG+06, GAD+06, KST04], core counts are increasing: Sun has shipped systems with eight cores per die [KAO05], and Intel has demonstrated processors with 48 [HDH+10] and 80 [VHR+07] cores. Every core added to a system increases the amount of overall hardware parallelism, raising the ceiling on system-wide throughput; this is the primary hardware trend which underlies our overall push, in this dissertation, toward chip-wide parallel execution.

As the number of cores on a processor die grows, and as more and more shared memory programs are run on these processors, cache coherence is fast becoming a central issue for multi-core performance. Cache coherence is the mechanism that allows us to retain the value of a given memory block in multiple processor caches at one time, and still maintain system-wide agreement about the value of that memory block at any point in program execution. Cache coherence

requires that we maintain enough information about the possible locations of the data across various caches so that we can find the data when a new consumer requests a copy, and so that we can communicate the obsolescence of cached values when someone writes to a memory block which is being shared. Since cache coherence involves significant amount of communication, wire speed and bandwidth are the primary limiters to the performance and scalability of cache coherence. As interconnection speeds fail to scale well with processor speeds [HMH01] and as interconnection bandwidth overhead (in terms of area and power) worsens over time [KZT05], novel mechanisms and policies are needed for accelerating coherence for a given wire speed and bandwidth.

In this chapter we examine the design of effective coherence mechanisms for a multi-core architecture that has multiple L2 caches, uses a directory-based cache coherence protocol, and features a scalable point-to-point interconnect. Multi-core implementations with small numbers of cores can use a snoop-based cache coherence protocol, which relies on a broadcast-based interconnect – typically a bus – to implicitly provide global communication of value and state changes, and to provide a single ordering of all accesses. However, broadcast-based interconnects such as buses scale poorly as the number of cores grows, motivating the switch to scalable interconnects and directory-based coherence solutions, which work without the need for any broadcast medium. Toward this end, we explore the use of directory protocols for multi-core processors, and tuning the protocols for the unique needs and opportunities provided by chip multiprocessors. We propose and evaluate a novel directory-based coherence scheme that improves the performance of parallel programs on such a processor.

We show that simply implementing traditional directory protocols within a chip does not provide the most effective solution. This is because those protocols were designed to work under very different topologies than those found on a chip multiprocessor. Some examples of topological assumptions which hold for a multiple-chip multiprocessor but *not* a single-chip multiprocessor, are that, for a given requester and home node:

1. The home node's main memory and the home node's directory are close to

each other, and about the same distance (latency) from the requesting node;

2. Once a request has reached the home node directory, the home node memory is closer than the caches of other nodes; on a traditional multiprocessor, the biggest latency barriers are between nodes. On a CMP, the latencies between nodes are small, and the dominant latency barrier is to off-chip memory, regardless of which node it is associated with.

3. The relative distance between nodes varies little. On a traditional multiprocessor, the communication latency between a given node to the nearest node and to the farthest node are often within a factor of two of each other, because latency is dominated in most cases by the off-chip and off-board latencies. On a chip multiprocessor, although all latencies are smaller in absolute terms, the relative latencies vary significantly; a core six hops away takes considerably longer to access than one a single hop away.

To take advantage of these differences, we introduce *proximity-aware* directory-based coherence. Proximity-aware coherence is motivated by the observation that, while a cache line can reside in multiple caches in the shared state, there is no guarantee that the line will be present in the cache of the home node corresponding to that line. Instead of requiring that the home node always source a given block of data – contacting very slow off-chip memory if it happens to not exist in the home node's caches – proximity-aware coherence enables the closest sharer to source the data on a read or write request. This results in decreased latency and bandwidth utilization, especially when the line is not present in the home node's cache but is present in the "shared" state in some other cache. Even when the line is present in the home node's cache, proximity-aware coherence can still help in reducing the bandwidth pressure on the interconnect.

4.2 Related Work

Directory-based protocols [LLG+92, LL97] have been proposed for scalable coherence on distributed shared memory multiprocessors. The coherence proto-

col for SGI Origin [LL97] was a four-state "MESI" (Modified, Exclusive, Shared, Invalid) protocol assuming sequential memory consistency. Directory coherence for the DASH multiprocessor [LLG+92], on the other hand, utilized a three-state protocol assuming weak memory consistency. Both these machines featured distributed shared memory (DSM) and implemented cache coherence with distributed directories that were stored at each node, but off-chip from the processor itself.

While directory-based coherence has been popular for DSM systems, it has not been studied in much detail for CMPs utilizing private L2 caches. Most of the current CMP implementations and proposals either have shared L2 caches with directories [BGM+00] or private L2 caches with snoop-based coherence [KZT05]; neither of those approaches scale well as the number of cores increases. Huh et al. [HKS+05] discuss a CMP model with directory-based coherence for their study of optimal degree of sharing for NUCA caches. They assume a central directory with constant access time. Zhang and Asanovic [ZA05] also consider directory-based coherence for one of their CMP models; they assume directory caches distributed by cache set indices.

Our implementations of directory-based coherence assume a distributed directory, with an on-chip directory controller and directory cache at each node. Caching the directory state was proposed [GWM90, ON90] as a means of reducing the memory overhead entailed by directories. Michael and Nanda [MN99] propose integrating directory caches inside the coherence controllers to minimize directory access time. Acacio et al. [AGGD02] study the impact of having first level directory on-chip caches.

One of our proposed policies, proximity-aware coherence, relies on location awareness to source shared data. CC-NUMA and COMA architectures [DT99, ZT97] also use spatial awareness for minimizing latencies. However, those architectures improve performance by retaining local copies of data that would otherwise require remote access. Proximity-aware coherence, on the other hand, does not require changing the mapping of data to sharers.

While we assume a conventional interconnect, Eisley and Peh [EPS06] move much of the coherence-related control and data storage into the network. Tradi-

tional sharer sets are replaced by *virtual trees* maintained within the routers themselves, with routers serving as active participants in coherence decisions. Through different mechanisms, their work and ours realize similar latency benefits on parallel workloads.

Chang and Sohi [CS06], seeking to combine the best attributes of both shared and private L2 caches, introduce a scheme for globally managing data placement, replication, and migration across the caches of all cores; even single-threaded workloads benefit through the use of neighboring cache resources. New policies manage storage through a centralized directory-like structure suitable for coordinating small numbers of cores. While their cache-management scheme is orthogonal in concept to that of a coherence protocol, both it and our own optimizations improve performance by avoiding off-chip memory accesses.

Token Coherence [MHW03] provides a framework for decoupling policies for coherence performance and correctness, the former implemented as *performance protocols*, and the latter ensured by *correctness substrates*. The specific performance protocol considered in that work, *TokenB*, broadcasts requests in order to avoid resorting to main memory accesses unnecessarily. Our proximity-based coherence scheme seeks the same goal, and could itself be expressed as a particular performance protocol atop a token-based system.

4.3 A CMP Architecture with Directory-based Coherence

Here we describe the processor architecture that we use for our study, as well as the baseline implementation of directory-based coherence for this architecture.

4.3.1 Architecture

Our experimental architecture is a chip multi-processor consisting of 16 cores arranged as a 4×4 mesh of tiles. Each tile contains an in-order core with private Level-1 instruction and data caches, a private unified Level-2 cache, a

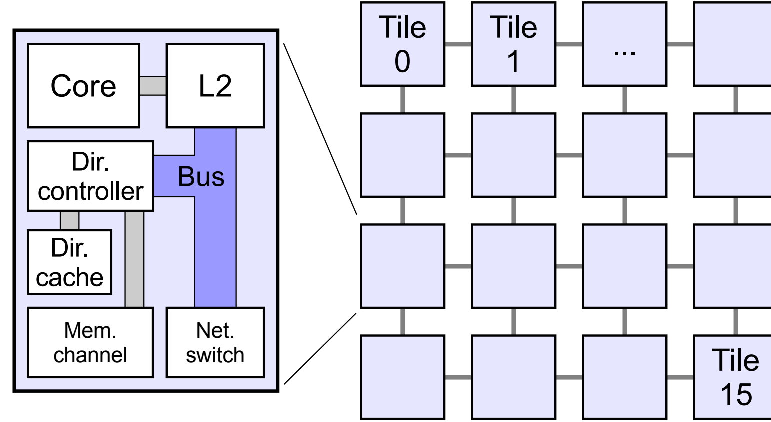


Figure 4.1: The baseline chip multiprocessor architecture, with 16 tiles. Each tile contains a core (with L1 caches), an L2 cache, a directory controller, a directory cache, a network switch, and a memory channel.

directory controller, and a network switch connecting to the on-chip network, as shown in Figure 4.1. Memory – both directory and regular program memory – is accessed through on-chip memory controllers, with one located on each tile. Each memory channel provides access to a different range of physical memory addresses. The architecture resembles a conventional mesh-connected multi-chip multiprocessor. The optimizations considered in this chapter exploit, among other things, the non-uniform latencies between cores inherent in a mesh architecture. This non-uniformity (in particular, the ratio of the latency for communicating with a distant node to that for communicating with an adjacent node) will only increase with larger CMPs; as wire delays increase, the absolute difference between these latencies will increase as well. Note that, while our baseline is a canonical architecture, the techniques outlined here can apply to any system with multiple (L2) caches, whether those caches each serve a single core, or each serve a cluster of cores.

We contrast this architecture with that of a more traditional multiprocessor (Figure 4.2), which is composed of multiple chips and typically multiple boards. A coherence protocol that is designed for the traditional multiprocessor will not exploit the topology of a chip multiprocessor well; it assumes that, for a given memory address, node memory is close to the home node directory, and that

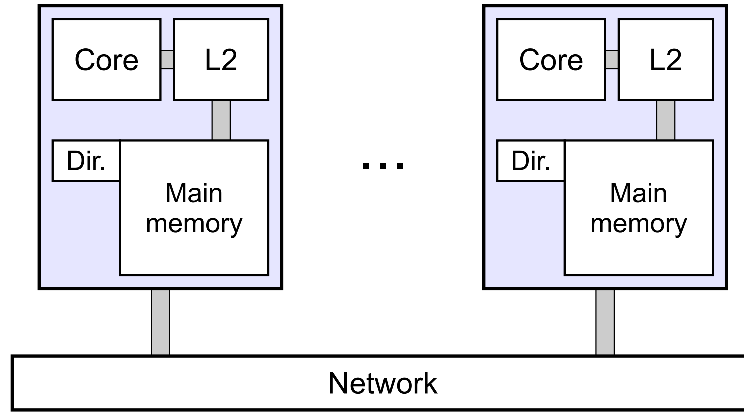


Figure 4.2: A more traditional multiprocessor (multi-chip, multi-board).

remote caches are far away. Neither assumption is true on a chip multiprocessor.

We assume that the L2 cache is tightly coupled to the rest of the tile. The tag and status storage are kept separate from the data arrays and close to the core and router for quick tag resolution. Accesses to resources on other tiles require that traffic travel through the network switch and over the on-chip network, experiencing varying access latencies depending on the distance between the tiles and the loads on the links between them.

We also assume *directory caches* (DC), one per node, which cache directory state as it is used by the directory controllers. Instead of accessing the off-chip directory memory for each coherence operation, the directory controller accesses the DC instead. All state changes are made to the contents of the DC itself. Only when there is a miss in the DC does the directory controller need to make an off-chip access to determine the coherence state of a line; given that only a single node is designated as the point of contact for directory information for any given cache line (its "home node"), the corresponding coherence state cannot exist in another node's DC, so the directory caches themselves need not be coherent.

The directory cache is organized as a set-associative cache where each cache line holds state corresponding to multiple contiguous memory blocks, to exploit spatial locality. A new entry is created in the DC for every line that is loaded. The directory cache replacement policy is LRU. Note that this organization decouples

L2 tags from the coherence directory tags. This enables low-latency access to the coherence state of a line, even when the line is not present in the L2 of the home node.

A generic four-state "MESI" protocol [LL97] adapted for CMPs is used as the baseline protocol for on-chip data coherence. The MESI protocol is named for the four possible states maintained for each block in a particular cache:

- **M, modified:** This cache has a modified version of the data, and no other cache has a copy.
- **E, exclusive:** This cache has a clean copy, and no other cache has a copy.
- **S, shared:** This cache has a clean copy, but other caches may also have copies.
- **I, invalid:** No data is present.

Additionally, in a directory protocol, the home node must keep track of the global state of each line, as well as the set of possible sharers of each line, in order to coordinate writes to shared data.

Our proposed implementation is a variant of this protocol. We now describe the details of our baseline coherence protocol.

4.3.2 Baseline Coherence Protocol

To illustrate the directory coherence protocol, first consider how an L1 read miss traverses the memory hierarchy:

- **Requester:** If the requested location is present in the requester's L2 cache, the cache simply supplies the data and no state change is required at the directory level. If there is an L2 miss, a request is sent to the home node which is associated with the desired memory address.
- **Home node:** The directory controller accesses the node's directory cache, and directory memory if necessary, to examine the coherence state for the

desired cache line. If the home node itself is indicated as a sharer of the desired data, the directory controller forwards the request to the local L2 cache for service. Otherwise, if the coherence state indicates the block is shared (and thus unmodified), a read from the main memory attached to the home node is initiated, and the result subsequently sent to the requester. If the coherence state instead indicates that the block is dirty (thereby held exclusively by one node), the request is forwarded to that remote node's L2 cache for service.

- **Remote node:** The node with the dirty copy replies with the most up-to-date version of the data, which is sent directly to the requester. In addition, a sharing write-back message is sent to the home node to update main memory, and to change the directory state to indicate that the requester and remote nodes now have shared copies of the data.

Next, consider the sequence of operations that occurs when a location is written. We focus here on the case of a write miss.

- **Requester:** A read-exclusive request is sent to the home node to retrieve the cache line and gain ownership.

- **Home node:** The home node can immediately satisfy an ownership request (from its attached memory) for a location that is in the uncached state. If a block is in the shared state, then all cached copies must be invalidated. The entry in the directory cache corresponding to the request address indicates the nodes that have the block cached. Invalidation requests are sent to these nodes. For weakly consistent processors, the home node would concurrently send an exclusive data reply to the requesting node (though data need not be sent for upgrade misses), and then wait for invalidate ACKs from the other potential sharers. For strongly consistent processors, the home node waits until it has received invalidate ACKs from all sharers before replying to the requester and granting ownership of the block. For both types of consistency, a request is considered serviced at the home node only after invalidate ACKs have been received from all previous sharers. If, instead of the shared state,

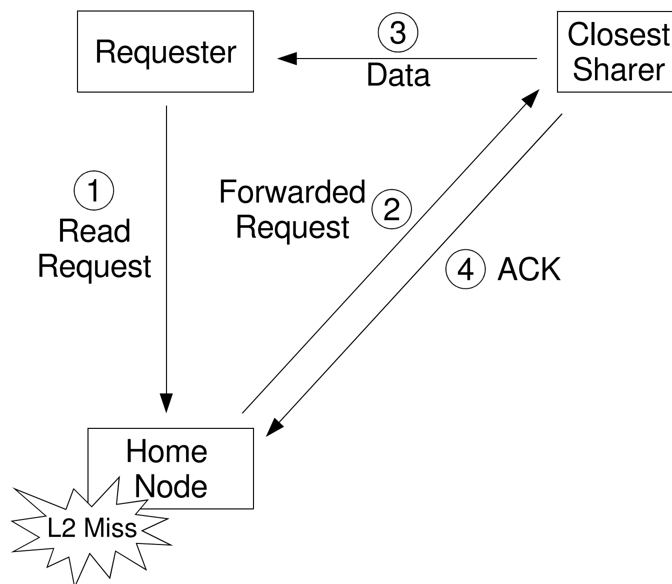
the directory indicates that the desired block is initially dirty, then the read-exclusive request must be forwarded to the sole owner, as in the case of a read.

- **Remote node:** If the directory had initially indicated that the memory block was shared, then the remote nodes are each sent an invalidation request to eliminate their copy. Upon receiving the invalidation, each remote node invalidates the corresponding line, and then replies to the home with an acknowledgement. If the directory had instead indicated the block was initially dirty, then the sole owner is sent a read-exclusive request. As in the case of the read, the remote node responds directly to the requesting node with the data, and sends the home node a message acknowledging transfer of ownership.

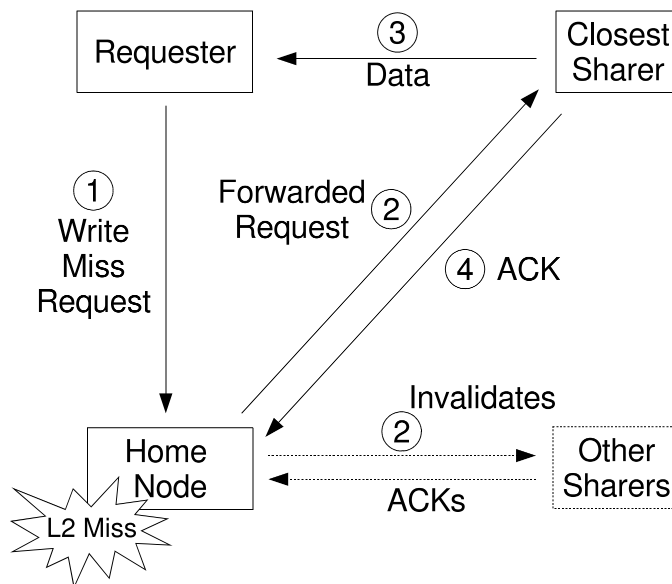
4.4 Accelerating Coherence via Proximity Awareness

Proximity-aware coherence encompasses the recognition of two facts particular to chip multiprocessors. First, that an on-chip cache access – even to a remote node – is always closer than an off-chip memory access. Second, that if there are multiple sharers of the data, selecting the right source to provide the data (one who is *close* to the requester) can reduce both latency and bandwidth utilization. We present a novel directory-based coherence schemes that exploits these properties.

For conventional directory-based coherence, a read or write miss to a line that is in the shared or the uncached state always results in the home node sourcing the data. However, the data may not be in the home node's L2, and accesses to off-chip memory are expensive. Proximity-aware coherence relies on the observation that even if data is not present in the home node's L2, it might still exist in shared state in some other L2 on the chip. This relaxes the constraint of the home node always sourcing the data in such scenarios and instead allows other sharers to source the data.



(a) Read Miss



(b) Write Miss

Figure 4.3: Proximity-aware coherence.

To illustrate the proximity-aware coherence protocol, first consider how a read miss traverses the memory hierarchy. Initial actions at the requester remain the same as in the baseline. The protocol differs at the home node and at the remote node.

- **Home node:** The home node examines the directory state of the memory location. If the block is dirty, the request is forwarded to the exclusive owner, as in the baseline. If the block is uncached, the home node services the request from main memory, as in the baseline. Otherwise, the block is clean; if the home node is indicated as a sharer, the request is forwarded to the home node's L2 cache for service. If the home node is not a sharer, but other nodes are, then the directory controller selects one or more of the potential sharers to ask for the data. The home node sends a message to the closest of these sharers, requesting it forward the data to the original requester. If the protocol supports multiple sharer requests, others are contacted in turn, until one is found to have the data or the maximum number of requests has been made.

  The directory state is not updated until the directory controller either receives an ACK from a remote node indicating that the desired data has been forwarded to the original requester, or every proximity-based request has been NACKed, at which point the controller gives up on future such requests and falls back to requesting the data from main memory.

- **Remote node:** If the remote node is in the dirty state, the same actions take place as in the baseline. However, if the data is shared and the remote node has been asked to forward the data to the requester (because it is the closest sharer), the remote node sources data directly to the requester and sends an ACK back to the home node. If the requested line is not in the remote node's L2 cache when the request from the home node reaches it, the remote node responds with a NACK.

Now consider the sequence of operations that occurs when there is a write miss. Again, initial actions remain identical at the requester.

- **Home node:** If the requested line is in the home node's L2, the same actions take place as in the case of the baseline coherence implementation.

  If the directory indicates that the block is dirty, then the read-exclusive request must be forwarded to the exclusive owner, as in the case of a read.

  If the line is in shared state AND the line is not in the home node's L2, forward-exclusive requests are sent in turn to one or more potential sharers, as in the read-miss case, eventually falling back to reading from main memory if the forwarding requests fail. Any potential sharers which were not sent a forward-exclusive request are then sent invalidate requests, in parallel. Directory state is not updated until replies are received from all sharers.

- **Remote node:** If the directory had indicated a dirty state, then the exclusive owner receives a read-exclusive request. The coherence transactions in that case are identical to the baseline coherence implementation.

  If the directory had indicated that the memory block was shared, and the remote node is the subject of a forward-exclusive request, it forwards a copy of the data (if present) to the original requester, invalidates its own copy, and responds with an ACK to the home node. If the remote node does not have the data, a NACK is sent.

Proximity-aware coherence attempts to ensure that if data is anywhere in the CMP in the appropriate state, a read or write request can be satisfied without the need to do off-chip memory access at the home node. This decreases the latency of coherence. Also, proximity-aware coherence should result in decreased overall bandwidth utilization since the control messages are much smaller than data messages: even though the number of control messages increases, the larger data-carrying responses will travel shorter distances. The latency-bandwidth tradeoffs depend on the spatial location of the nodes and the relative size of the data messages versus control messages.

The implementation of proximity-aware coherence is a straightforward, safe extension of the mechanisms present in the baseline system; no additional storage is required specifically to support it, and the additional state transitions within

the directory and cache controllers do not require significant complexity to handle. Correctness of the underlying protocol is not affected, since 1) the proximity-aware extensions are applied only to clean data (possibly after invalidates), 2) the resulting cache blocks are left clean, and 3) the corresponding directory entry is always updated with a superset of the actual sharers before processing the next request for the subject memory block.

Note that proximity-aware coherence is not applicable for upgrade misses, as no data blocks need to be transmitted in that case. Proximity-aware coherence will work whether the processor supports strong (e.g., sequential) or weak consistency.

While proximity-aware coherence as introduced forwards a single data request to the sharer nearest each requester, a variety of request policies are possible. We explore two additional considerations: how many sharers to send proximity forwarding requests before giving up, and what metric is used to order candidate sharers.

Forwarding requests to multiple potential sharers pits the benefits of avoiding unnecessary off-chip memory accesses when some of the advertised sharers lack copies of the data, against the bandwidth and latency costs of the additional control messages. (It is possible for nodes listed as sharers at the directory to no longer contain copies of the data, because it has been evicted from the cache). We consider forwarding requests to the "nearest" one, two, or three nodes before falling back to an external memory access.

We consider three node selection policies, which are used to order the potential recipients of a proximity forwarding request. The first, *near*, orders candidates by the Manhattan distance from each remote node to the requester. The second, *via*, orders candidates by the sum of the Manhattan distances from the home node to each remote node, and from each remote node to the requester. The final policy, *rand*, simply chooses nodes from the sharer set at random.

In reporting results, we combine the node selection policy and try-count, e.g., a policy which attempts to source data from two remote nodes nearest to the requester is referred to as *near2*. We refer to the policy of consulting a single sharer at random before reverting to main memory simply as *rand*.

4.5 Methodology

We perform all evaluations with a modified version of RSIM [PRA97b], as introduced in Chapter 2. Home nodes are assigned based on a "first-touch" policy [LL97]; this ensures that the "home" designation is assigned to a node that is likely to be an active sharer of the data. We assume 70 nm technology (based on BPTM [CSO+00]) and model a 3 cycle network hop, which includes includes the router latency and an optimally-buffered 5 mm inter-tile copper wire on a high metal layer. The latencies are modeled by assuming a 24 FO4 processor clock cycle [HP04]. Memory channels are assumed to have RDRAM interfaces. Table 4.1 lists the important system parameters used in the experiments.

The workloads we use to evaluate our coherence mechanisms are listed in Table 4.2. These are all parallel workloads and represent a wide variety in their computation-communication ratio. The applications also have varying degrees of sharing and synchronization, and represent diverse application domains.

4.6 Analysis and Results

Next, we present our evaluation of the proposed *proximity-aware coherence* policies. All our evaluations assume a sequentially consistent processor with MESI baseline protocol as described in Section 4.3.2. Evaluations are presented for 256 KiB L2 caches unless otherwise noted.

With proximity-aware coherence we seek to eliminate accesses to memory for any data held in an on-chip cache, and to minimize the distance traveled for any cache-to-cache transfers. The first goal, in particular, exploits the unique property of chip multiprocessors (vs. traditional distributed shared memory multiprocessors) that the latency of communication between compute nodes – and thus the latency of seeking data from a peer core's cache – is significantly lower than the latency of seeking data from one's own local memory.

The effectiveness of this technique, then, will depend in large part on how often a requested line is not present at the queried home node – which would typically result in a memory access – yet is present in some other node's caches.

Table 4.1: Architecture details.

Parameter	*Value*
Processor model	in-order
Issue-width	dual-issue
Instruction window (entries)	16
Load/store queue (entries)	16
Branch predictor	bimodal (2K)
Number of integer ALUs	2
Number of FP ALUs	1
Cache line size	64B
L1 I-cache size/associativity	32KiB/4-way
L1 D-cache size/associativity	32KiB/4-way
L1 Load-to-use latency	1 cycle
L1 replacement policy	Pseudo-LRU
L2 cache size/associativity	256KiB/8-way
L2 load-to-use latency	6 cycles
L2 replacement policy	Pseudo-LRU
Directory cache size/associativity	16KiB/4-way
Directory cache load-to-use latency	1 cycle
Directory cache replacement policy	LRU
Network configuration	4×4 mesh
One-hop latency	3 cycles
Worst-case L2 hit latency (contention-free)	48 cycles
Number of memory channels	16 (1 per L2)
Directory memory latency	30 cycles
External memory latency	256 cycles

Table 4.2: Workloads

Benchmark	*Benchmark Suite*	*Problem Size*
APPBT	NAS	$64 \times 64 \times 64$, 30 iterations
FFT	SPLASH2	64K points
LU	SPLASH2	256×256 matrix, 8×8 blocks
MP3D	SPLASH	48000 nodes, 20 timesteps
Ocean	SPLASH2	130×130 array, 10^{-9} error tolerance
QuickSort	TreadMarks	512K integers
Unstructured	Wisc	Mesh.2K, 5 timesteps

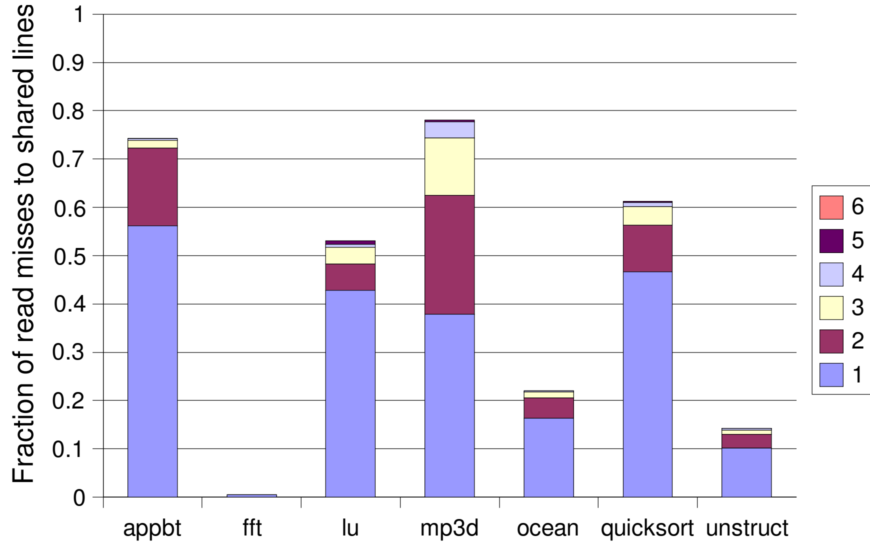


Figure 4.4: Fraction of read misses to shared lines for which the home node is not a sharer, but another node is. The higher the bars, the more potential benefit from proximity-aware coherence. The segments of each bar indicate the minimum number of hops from a sharer to the requester, for each miss.

We characterize this frequency in Figure 4.4, which shows the fraction of read misses to a shared line (i.e. lines for which the corresponding directory entry lists the line in the "shared" state) that do not have the home node listed as a sharer. The higher the bars, the higher the opportunity for initiating proximity-aware cache-to-cache transfers. These results include the effect of the first-touch home node assignment policy, which increases the chance that the home node is an active sharer. In the absence of this policy, the potential for proximity-aware coherence would be even greater.

Note also, in Figure 4.4, that the results for each benchmark are broken down in terms of the distance from the requester of the closest node that is listed in the directory as a sharer. So, if the requester is node 0 (the top left tile of the chip), and the closest sharer for the requested line as listed in the directory is node 15 (the bottom right tile of the chip), it adds to the stacked bar corresponding to 6 (because nodes are six network hops apart).

There are two things to note in this graph. First, we can see that it is quite common for shared data to not be found in the home node, but exist elsewhere on the chip. In fact, this happened for nearly half of read misses to shared lines

(43%). The actual percentage does depend significantly on the data access patterns, however, and therefore we observe a significant variance by benchmark. For example, for *unstruct* and *ocean*, the requester is often the home node as well and thus only 14% and 22% of read misses to shared lines, respectively, have the home node not listed as a sharer. *fft* experiences a very small number of read misses to shared lines, essentially all of which are hosted by the home node. Data migration patterns are more aggressive for *appbt* and *mp3d*, resulting in a high fraction of read misses to shared lines with non-home sharers: 74% and 78%, respectively.

Another observation from the graph is that most of the requests can be satisfied by nodes that are one hop away. While this is not very surprising – the average distance between two nodes in a 16×16 tiled processor is only three hops – it does mean that proximity-aware coherence, done properly, can result in significantly reduced average L2 miss latency.

While Figure 4.4 shows the potential for proximity-aware coherence, the numbers do represent an upper bound, since there is no guarantee we will find the data at the closest apparent sharer. Evictions in the individual caches cause the sharer set in the directory to always be a superset of the actual sharers. Thus, while this result gives an accurate account of how often a sharer exists, there will be times that – depending on evictions in progress – finding a sharer takes multiple queries.

A more direct measure of success for this technique coherence is the reduction in average L2 miss latency, when proximity-aware coherence is applied. Figure 4.5 shows the average latency of an L2 miss for a multi-core processor enhanced with proximity-aware coherence. The results are normalized against L2 miss latencies for the processor with baseline coherence. As can be seen, proximity-aware coherence can often result in significant reduction in latency of coherence operations. Latency reductions of up to 79% (*quicksort*) were observed. Average latency reduction was 24.6%.

Proximity-aware coherence has two distinct enhancements: elimination of unnecessary memory accesses, and the minimization of distance traveled by shared data. The similarity of the three bars in each group indicates that the former is

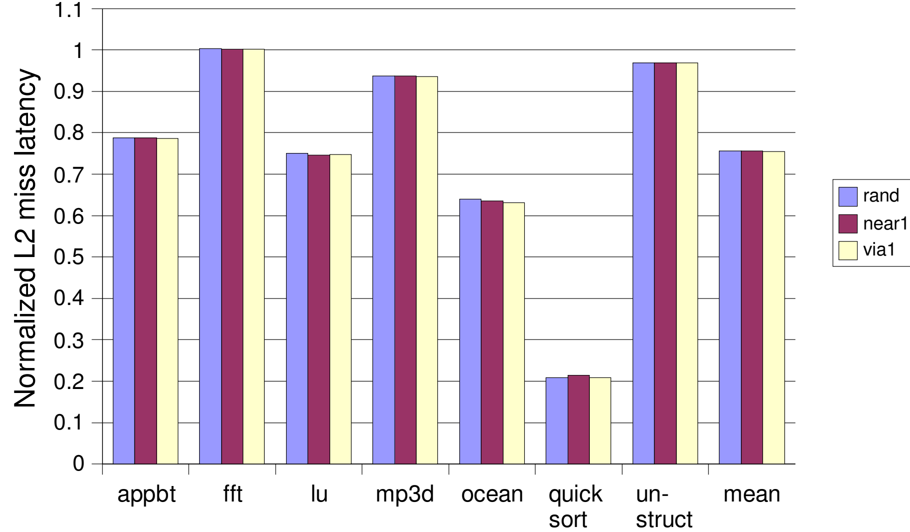


Figure 4.5: Mean L2 miss-service latency with proximity-aware coherence, normalized (*base* = 1.0). The *rand* policy queries a random on-chip sharer, *near1* queries one sharer closest to the requester, and *via1* queries one sharer with a minimum distance along the home-requester path.

clearly the more important factor driving performance in these experiments; all three distance protocols achieve strong gains, but the difference between them is slight.

However, it is still worth noting that *via* performs slightly better than *random* and *near*. *Random (rand)* represents a baseline distance-oblivious heuristic. *Near* minimizes the distance from the sharer to the requester, but in many cases the total hops traveled from the home node to the requester (via the sharer) is greater than the minimal number of hops from the home node to the requester. This is because the control message must travel from the home node to the sharer which, although presumably close to the requester, may be distant from the home node. So while the distance traveled by the data is minimized, the total distance is not. The *via* enhancement greatly increases the likelihood that the combined distance traveled by the control message and the data is no greater than the minimal distance. This results in reduced overall L2 miss latency.

An associated effect of performing a spatial optimization (e.g., *near* and *via*) is that the average bandwidth pressure on the interconnect is reduced. This is because the sourced data now traverses a shorter distance, on average, from the

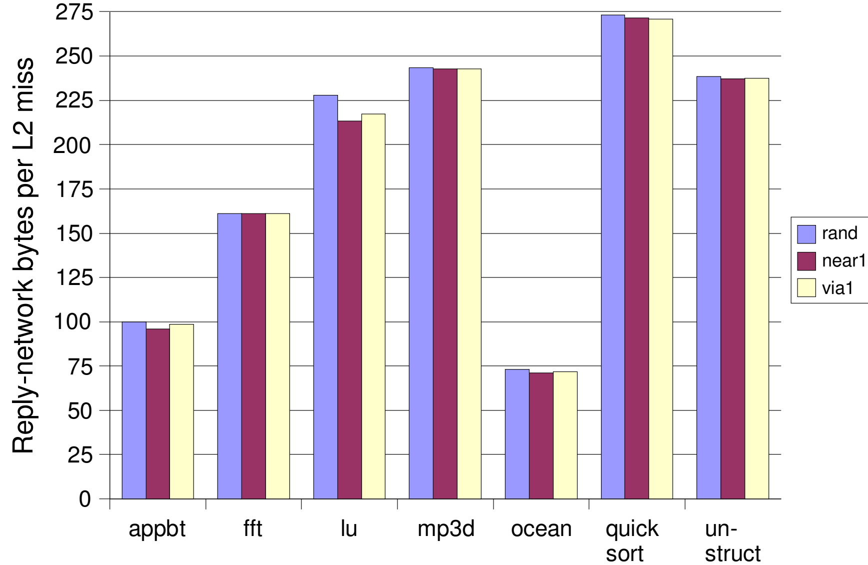


Figure 4.6: Reply-network utilization, in terms of *total* network traffic generated for each miss. (Each individual link transit counts toward the total.)

sharer to the requester. Figure 4.6 shows the average number of bytes of data transferred per L2 miss for the three proximity-aware policies. We observe up to 6% reduction in bandwidth requirements for *via* and *near* over *random*. In a system where contention for some links was high, we would expect that reduced bandwidth to translate more directly into latency reductions (due to reduced queueing). However, the SPLASH2 benchmarks put very little overall pressure on our assumed interconnect.

In fact, there are several reasons why we believe these results understate the potential gains of the proximity-aware coherence, and the spatial optimizations (e.g., *via*) especially. The SPLASH2 benchmarks have small working sets relative to our cache sizes. This has two effects: contention for links is low, and the number of L2 misses per instruction is generally quite small. This means that the sensitivity of these results to the actual L2 miss latency is much lower than most realistic parallel commercial workloads. Additionally, we will see even greater gains as the number of cores – and thus the maximum and average distance between cores – increases. With future many-core systems composed of tens to hundred of cores, the impact of proximity-aware coherence overall as well as of the node-selection

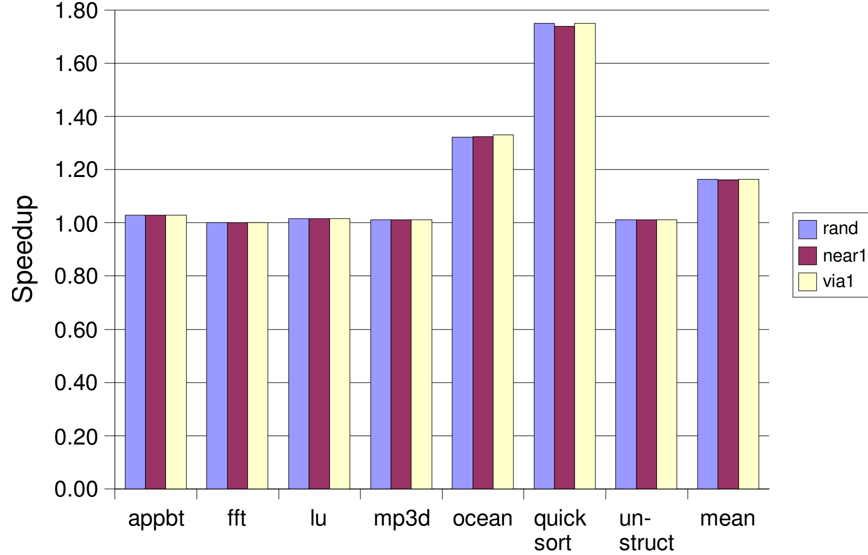


Figure 4.7: Speedup from using proximity-aware coherence, on a processor with sequential consistency.

policy will only increase.

In spite of the benchmark limitations discussed above, the significant reduction in average L2 miss latency through proximity-aware coherence does translate into improved system performance. Figure 4.7 shows the performance of the three proximity-aware policies normalized against the baseline coherence protocol. As can be seen, proximity-aware coherence can result in speedups up to 75% with an average speedup of 16%. The *via* policy results in the highest system performance.

We also evaluate the impact of the *near* and *via* policies with try-counts of two and three, i.e., retrying one or two additional potential sharers, after the failure of the first proximity-read request, before falling back to the use of main memory. The use of additional requests increases the success rate of proximity-read sequences by a mean of 7% for *near2* and *via2*, and 9% for *near3* and *via3*. Unfortunately, these gains are offset by increases in bandwidth utilization, as well as some increases in the L2 miss-service latency (since retries are handled serially). While *lu* benefits from retries with an additional 1% decrease in mean L2 miss-service time and an additional 0.2% increase in speedup, the suite-wide impact of retries is negligible.

4.7 Summary

In this chapter, we have taken a step in exploring the design of directory coherence protocols for chip multiprocessors. Future multi-core designs will continue to feature multiple L2 caches and scalable interconnects. We have seen here that simply implementing traditional directory protocols on a multi-core architecture does not provide the best design.

In particular, we have shown a multi-core specific customization of directory coherence – *proximity-aware coherence* – which resulted in speedups up to 75% over a traditional directory coherence protocol applied directly to a multi-core processor, with average speedup of 16%. Reduction in average L2 miss latency for coherence misses was even greater: per-benchmark average miss latency was reduced up to 79%, with a suite-wide average reduction of 25%. More importantly, the results suggest that as multi-core designs are scaled up, both in terms of number of on-die cores as well as the sizes of the data and working sets of applications, the potential for proximity-aware coherence will increase.

Acknowledgements

The work in this chapter was supported in part by NSF grant CCF-0541434 and Semiconductor Research Corporation grant 2005-HJ-1313.

This chapter contains material from "Proximity-Aware Directory-based Coherence for Multi-core Processor Architectures", by Jeffery A. Brown, Rakesh Kumar, and Dean Tullsen, which appears in *Proceedings of the Nineteenth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA).* The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2007 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

The Shared-Thread Multiprocessor

5.1 Introduction

As processor manufacturers further develop multi-core designs, the successful exploitation of thread-level parallelism is central to sustaining the scaling of system-wide performance. As introduced in Chapter 1, while the details of expressing TLP in software may vary, the efficiency of the underlying hardware support for thread manipulation and scheduling is inescapably critical. For general-purpose computing workloads – wherein the amount of parallelism is not constant – we desire an architecture which achieves good performance on a single thread, with performance improving as more threads are added, up to a relatively high number of threads.

Designs with many relatively small cores offer high peak throughput, but provide high per-thread latencies and perform poorly when thread-level parallelism is low. A processor with fewer, more powerful cores will provide lower per-thread latencies and good few-thread performance, but will be comparatively inefficient when running many threads. If we add multithreading [TEL95, TEE+96] to the latter design, we achieve a better tradeoff: providing both low latency when offered few threads, and high throughput when running many threads. However, there is

a limit to how far throughput gains will scale, since very large cores do not scale even their peak throughput linearly with area.

In this chapter, we describe an architecture which features relatively modest cores with only minimal support for on-core multithreading (i.e., simultaneous multithreading, or SMT), augmented with fast off-core thread storage, thus supporting more threads chip-wide than allowed by the SMT cores themselves. Peak performance on such an architecture will scale roughly linearly until the number of threads reaches the number of cores, will continue to scale well as the number of threads increases to the total number of SMT contexts, and continue to climb modestly as even more threads are added. We call this architecture the *Shared-Thread Multiprocessor* (STMP).

The Shared-Thread Multiprocessor enables distinct cores on a chip multiprocessor to share thread state between them. This shared thread state allows the system to not only mix on-core and off-core support for multithreading – providing high ILP with minimal design overhead – but, by scheduling threads from a shared pool onto individual cores, also allows for rapid movement of threads between cores. This approach enables, for the first time, low-latency "multithreading-type" context switches between distinct cores. In this way, it combines the simplicity of multi-core designs with the flexibility and high utilization of aggressively multithreaded designs.

This architecture offers several benefits, compared to a conventional chip multiprocessor:

- By providing more thread state storage than available in the cores themselves, the architecture enjoys the ILP benefits of many threads, but carries the in-core complexity of supporting just a few.

- Threads can move between cores fast enough to hide long-latency events such as memory accesses. This enables very-short-term load balancing in response to such events.

- The system can redistribute threads to maximize symbiotic behavior and balance load much more often than traditional operating system thread schedul-

ing and context switching.

5.2 Related Work

Tune et al. describe Balanced Multithreading, (BMT) [TKTC04], which allows a single processor core to combine the benefits of simultaneous multithreading and a form of coarse-grain multithreading. In their preferred design, the SMT core only supports two hardware threads, keeping complexity low and the register file small. However, this is supplemented with off-core thread storage, which allowed thread state to be brought into the core quickly when an SMT context became idle due to a long-latency event. This adds little or no complexity to the core because it relies on injected instructions to transfer context state in and out.

In this way, more than two threads time-share the two SMT contexts. BMT achieves the instruction throughput of an SMT core with additional hardware threads, without the full hardware costs: two SMT-scheduled threads augmented with two coarsely-scheduled threads can exceed the IPC of three SMT threads.

The Shared-Thread Multiprocessor is an extension of the BMT architecture. In STMP, the off-core thread storage is shared among a pool of cores. This brings several new capabilities not available to BMT: the ability to dynamically partition the extra threads among cores, the ability to share threads to hide latencies on multiple cores, and the opportunity to use the shared-thread mechanism to accelerate thread context switching between cores (enabling fast rebalancing of the workload when conditions change).

Other work has examined thread scheduling policies to maximize the combined performance of threads on a multithreaded processor [ST00, PELL00]. In Chapter 3, we identified the importance of accounting for long-latency loads and minimizing the negative interference between stalled threads and others on the same core. Both BMT and STMP address those loads, by first removing stalled threads and then injecting new threads which are not stalled.

Constantinou et al. [CSM+05] examine several implications of thread migration policies on a multi-core processor migration. However, they do not examine

these issues in the context of the type of hardware support for fast switching that our architecture provides.

Torrellas et al. [TTG95] is one of several papers that examine the importance of considering processor affinity when scheduling threads on a multiprocessor. In their work, they endeavor to reschedule threads where they last executed, to minimize cold cache effects. We find that processor affinity effects are also extremely important in the STMP.

Spracklen et al. [SA05] argue for the combination of multiple SMT cores within a CMP for efficient resource utilization in the face of high-TLP, low-ILP server workloads. Their focus is primarily on such workloads, sacrificing single-thread performance for a design tailored to maximize throughput, without paying particular attention to the movement of threads. The STMP exploits high ILP when it is available, and offers fast context-switching to other threads in order to boost system throughput when ILP is lacking.

Dataflow architectures [Arv81, AN90] offer an alternative to traditional ILP-based designs. Stavrou et al. [SKET07] consider the implementation of hardware support for data-driven multithreaded computation on top of a conventional CMP. Their design uses additional per-core hardware to coordinate data movement and the scheduling of threads as live-in data become ready; threads are explicitly compiled as slices of decomposed dataflow and synchronization graphs. Prefetching and cache conflict tracking are used to avoid long-latency memory stalls during a thread's execution. The STMP, in contrast, uses traditional functional unit scheduling and control-driven instruction streams, detecting and reacting to long-latency memory events as they occur.

5.3 The Baseline Multi-threaded Multi-core Architecture

The next two sections describe the processor architecture that we use for this chapter. This section describes a conventional architecture which is the base upon which we build the Shared-Thread Multiprocessor described in Section 5.4.

Our baseline is a multithreaded, multi-core architecture (referred to sometimes as chip multithreading [SA05]). Several examples of this architecture already exist in industry [CFS+04, FMJ+07, KAO05, JN07].

5.3.1 Chip multiprocessor

We study a chip multiprocessor (CMP) design consisting of four homogeneous cores. Each core has an out-of-order execution engine and contains private first-level instruction and data caches; the four cores share a common second-level unified cache.

The cores communicate with each other and with the L2 cache over a shared bus; data caches are kept coherent with a snoop-based coherence protocol [KEW+85]. Off-chip main memory is accessed via a memory controller that is shared among cores. While this relatively simple interconnect does not scale up to many cores, it suffices for four cores, and still features the essential property that inter-core communication is much faster than memory access, with latency comparable to that of an L2 cache hit. Our STMP extensions can apply to more scalable interconnects, such as point-to-point mesh networks, but we utilize this simpler symmetric interconnect in order to focus on the STMP implementation itself.

5.3.2 Simultaneous-Multithreaded cores

Each core of our chip multiprocessor features Simultaneous Multithreading [TEL95, TEE+96] (SMT) execution, with two hardware execution contexts per core, similar to several recent processor designs [CFS+04, KM03].

Two-way SMT allows for two threads to share a core's execution resources within any cycle, enabling efficient resource utilization in the face of stalls. SMT has been shown to effectively hide short-term latencies in a thread by executing instructions from other threads. It provides significant gains in instruction throughput with small increases in hardware complexity. We evaluate our Shared-Thread Multiprocessor designs on top of SMT in order to demonstrate that there is addi-

tional potential for performance gain beyond what SMT is able to capture. SMT is less successful at hiding very long latencies, where at best the stalled thread becomes unavailable to contribute to available ILP, and at worst stalls other threads by occupying resources.

The inclusion of SMT support is not essential in the design of the STMP; single-threaded cores could apply coarse-grain multithreading, switching in threads from the shared off-core thread pool. Such an architecture would still offer significant advantages over a single-threaded CMP without inactive-thread storage, but we find the SMT-based architecture more attractive, because the second thread on each core allows the processor to hide the latency of the thread swap operations themselves.

5.3.3 Long-latency memory operations

A typical memory hierarchy features successively larger and slower memory units (caches) at each level. In such a system, individual memory instructions may vary widely in response times, ranging from a few cycles for a first-level cache hit, to hundreds of cycles for a main-memory access, to essentially unbounded latencies in the event of a page fault.

While out-of-order execution is able to make progress in the face of memory operations taking tens of cycles, those which take longer rapidly starve the processor of useful work: once the memory instruction becomes the oldest from that thread, there is a finite dynamic-execution distance following it beyond which the processor is unable to operate, due to resource limitations. In Chapter 3 we explored this phenomenon, in particular the impact of long-latency loads on SMT processors, along with several mechanisms for detecting these situations and mitigating the impact on co-scheduled threads.

Our baseline SMT processors include a similar long-latency-load detection and flushing mechanism: when a hardware execution context is unable to commit any instructions in a given cycle, and the next instruction to commit from the resident thread is an outstanding memory operation which is older than a time threshold, it is classified as "long-latency", and action is taken. In the baseline

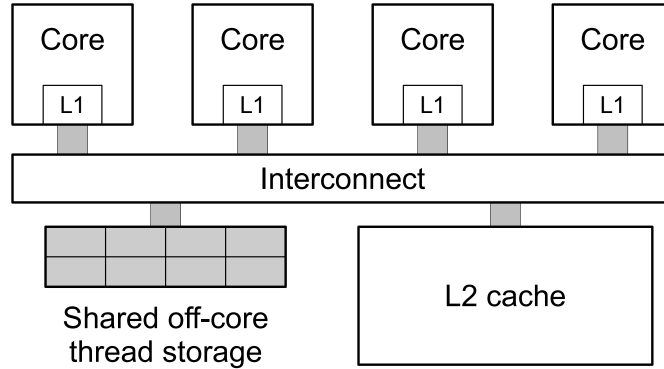


Figure 5.1: The Shared-Thread Multiprocessor.

system, instructions younger than the load are flushed while the load remains; the experimental STMP systems described below also use this signal as an input to scheduling decisions.

5.4 Shared-Thread Storage: Mechanisms & Policies

In the Shared-Thread Multiprocessor architecture, we augment the CMP-of-SMTs described in Section 5.3 with several relatively inexpensive storage and control elements.

5.4.1 Inactive-thread store

We add an off-core *inactive-thread store*: storage for the architected state of numerous threads. This storage is used to hold additional threads beyond those supported by the hardware execution contexts. Per-thread state consists primarily of logical register values, the program counter, and a system-wide unique thread ID. This state occupies a fixed-size store of a few hundred bytes per thread, which is accessed non-speculatively in a regular fashion; it can be implemented efficiently with a small amount of SRAM. This is shown in Figure 5.1.

5.4.2 Shared-thread control unit

We introduce additional control logic which coordinates the movement, activation, and deactivation of threads. The *shared-thread control unit* resides with the inactive-thread store and implements the various scheduling policies we explore. This control unit communicates with execution cores over the processor interconnect, receiving occasional notification messages from cores containing recent thread performance or status-change notifications, and sending messages which signal cores to trigger thread migration. This unit may be implemented in various ways, from a simple automaton to perhaps a small in-order processor (with some associated working memory), depending on the complexity of the scheduling policy desired.

As an alternative to using a discrete shared-thread control unit, its functionality may be implemented within the cores themselves in a scalable peer-to-peer fashion, which would be particularly beneficial as the number of cores is increased. We focus on a discrete implementation for this study.

5.4.3 Thread-transfer support

To each execution core, we add a mechanism to rapidly transfer thread state between the execution core with its internal data structures, and the simpler *inactive-thread store*.

Communication with the shared-thread control unit and its associated storage takes place across the processor interconnect, which is optimized for the transfer of cache blocks. To maximize communication efficiency, we add a small cache-block sized *spill/fill buffer* to each core, which is used for assembling thread-transfer data into cache-block sized messages. The spill/fill buffers are fixed-sized SRAM structures, which are accessed sequentially from beginning to end.

The primary concerns when removing a thread from a core are stabilizing a consistent, non-speculative view of the thread state, and then extracting that state from the on-core data structures into a concise form suitable for bulk transfers. We re-use mechanisms present on typical out-of-order processors to perform most of the detailed work involved with thread movement. We introduce two new micro-

instructions, *spill* and *fill*, which each transfer a single register value to or from the next location in the spill/fill buffer. These new micro-instructions each specify a single logical register operand; they are injected into an execution core in a pre-decoded form just before the register-rename stage, where they re-use the renaming and dependence logic to locate the appropriate values.

Each *spill* instruction extracts one architected register value, utilizing existing register-read ports and honoring any in-flight data dependences. When the *spill* completes execution, it appends the value to the spill/fill buffer. Each *fill* instruction reads one value from the spill/fill buffer as it executes, and writes to an architected register when it completes execution. When the buffer is filled by a write from a *spill* instruction, the entire buffer is bulk-transferred to the inactive-thread store, much like the delivery of a cache write-back. When the buffer is emptied by a read from a *fill* instruction, a flow-control indication is sent indicating that the core is ready for another block. (There is a small amount of ancillary data in these messages.)

In addition to the existing logic used to access registers, a new data pathway is necessary between the *spill/fill buffer* and whichever functional units it is most convenient to map *spill* and *fill* instructions to. Since *spill* and *fill* instructions are never introduced speculatively, are never used with overlapping register numbers, and always access the *spill/fill buffer* in order, adding this new data pathway is unlikely to introduce significant complexity to the processor.

Finally, each processor core is augmented with a small amount of *spill/fill control* logic, responsible for sending periodic performance counter samples to the shared-thread control unit, and for reacting to thread-swap requests. When a thread-swap request is received from the shared-thread control unit, the spill/fill control logic initiates a flush of the indicated hardware thread context – effectively causing an asynchronous trap – and then begins injecting *spill* or *fill* instructions (as appropriate), instead of vectoring to a conventional operating-system trap handler. This logic is also responsible for stalling fetch on the corresponding hardware context until the thread swap is complete. Note that, since each core supports two-way SMT execution, normal execution of a separate workload may proceed in

one context while another is performing a swap.

Processors without explicit register renaming may still implement these *spill* and *fill* instructions, through whatever mechanism the processor uses to map between architected register numbers and run-time storage. We expect these new instructions to be trivial to represent in the existing internal instruction representation of most microarchitectures.

In summary, then, the bulk of the support for the STMP is in the thread storage and control, which is at about the same level (e.g., in distance and latency) from the cores as the L2 cache, but is much smaller. Other support that needs to be added to each core is relatively minor, and none of this functionality affects potential critical timing paths (such as renaming, instruction scheduling, register access, etc.) within a core.

5.4.4 Scaling the Shared-Thread Multiprocessor

In this chapter, we focus on a single implementation featuring four cores sharing thread state. As the number of cores increases, we could scale this architecture in two ways. We could increase the number of cores sharing a single centralized thread store, or we could increase the number of STMP instances. It is unclear that much would be gained by sharing thread storage directly among more than four cores, due to increases in latency and contention. Instead, we envision "islands" of STMPs, each a small set of cores clustered around their own inactive-thread store. A 64-core CMP, then, might have 16 thread storage units, each servicing four cores. Movement between thread stores would be supported, but with much less frequent movement, and need not be as fast. Future work is necessary to fully analyze these options. However, if we assume the approach which clusters STMP cores, then our results, which focus on four cores, are indicative of not just that configuration, but also the characteristics of each cluster of cores in a larger configuration.

The mechanisms we present here are focused on the migration of registers and associated state; they assume a fairly generic memory and interconnect layout. The underlying system described in Section 5.3.1 offers a shared L2 cache, with

relatively low access latency. An important alternative case is that of an architecture with private L2 caches, which would still support efficient register migration, but which would experience more performance degradation due to cache effects subsequent to migration. While we focus on the shared-L2 case here, we will return to the private-L2 case in Chapter 6, and address post-migration cache effects in detail.

5.4.5 Thread control policies — Hiding long latencies

We consider a variety of policies for coordinating the motion of threads within the processor. The shared-thread control unit implements these policies, relying on status messages from individual cores for samples of performance data as well as notification of long-memory stall detection. For the sake of simplicity, these policies are centrally managed; more advanced implementations could coordinate peer-to-peer directly between cores.

The policies we evaluate fall into two distinct classes of policies, intended to exploit different opportunities enabled by the STMP: first, the ability to move threads quickly enough to react to very short-term thread imbalances resulting from long-latency operations; second, the ability to perform coarser-grained load-balancing, but at time scales much shorter than possible in an operating system. In this section we consider policies targeting the former opportunity, while we target the latter in Section 5.4.6.

Here, we focus on cases where threads are more numerous than cores, wherein some of the (multithreaded) cores are be more heavily loaded than others. When a thread on a less-loaded core stalls for a long-latency memory access – as characterized in Section 5.3.3 – it leaves the execution resources on that core under-utilized relative to the more heavily-loaded cores; in effect, hardware contexts on a core experiencing a stall will become relatively "faster" while the stall is outstanding. These policies seek to exploit the decreased contention on cores experiencing memory stalls. While this is possible when thread load is evenly balanced, it is less likely: it requires two threads to be stalled on the same core to create a significant opportunity for thread movement. Therefore, the load-imbalance phe-

nomenon exploited in this section is best illustrated when there are 5–7 threads on the (eight-thread) processor. In such scenarios, some cores will have one thread scheduled, and others two threads. If a thread scheduled by itself experiences a full-memory-latency stall, its parent core essentially sits idle for hundreds of cycles, while other cores are executing two threads. Because a four-wide superscalar processor running two threads typically achieves less than twice the throughput as when running a single thread, this imbalanced load is likely inefficient. If we can quickly move a co-scheduled thread to the temporarily-idle core, these two threads executing on separate cores may execute faster than they would when co-scheduled on a single core.

It should be noted in this section and the next, that we only show a subset of the policies considered. Typically, the shown policy is a tuned, effective representative of a class of similar policies.

The specific policies evaluated in this study are:

- **Stall-chase:** The *stall-chase* policy aggressively targets individual memory stalls. When a thread is detected as having stalled for a miss to memory, active threads on other cores are considered for immediate migration to the site of the stall. The results presented use a policy which selects the thread with the lowest recent sampled IPC for migration; other selection policies we have evaluated include choosing the thread with the smallest L1 cache footprint, the thread which has has been running the longest since its last migration, and choosing a thread at random.

- **Runner:** Under the *runner* policy, one or a subset of threads are designated as "runners", which will be opportunistically migrated toward stall-heavy cores; non-runner threads do not move, except in and out of inactive-thread storage, and back to the same core. This policy recognizes the high importance of processor affinity: most threads will run best when returning to the same core and finding a warm cache. The "runner" designation is used in the hope that some applications will inherently suffer less performance loss from frequent migrations than others. (The *runner* policy is similar to the

stall-chase policy in that long-memory stalls "attract" other threads; they differ in the selection of the thread to be migrated.)

This policy attempts to negotiate some of the more difficult tradeoffs facing these policies. If we seldom move threads between cores, then when we do move them, they tend to experience many L1 cache misses immediately as they enter a cold cache. If we move threads often enough that cold caches are not a problem, we are then asking all L1 caches to hold the working set of all threads, which puts excessive pressure on the L1 caches. By designating only a few threads as runners, we allow those threads to build up sufficient L1 cache state on the cores they visit frequently, decreasing the cost of individual migrations; additionally, each core's L1 cache now need hold the working set of only a few threads.

When a non-runner thread is detected as having stalled for a long-memory operation, the shared-thread scheduler decides whether to move the runner from its current location to the stalling core. Experimentally, interrupting the runner in order to migrate it whenever a long stall is detected on another core proved too disruptive to its forward progress; the results we present use a more relaxed policy, wherein the shared-thread scheduler records long-stall detection events, and when the runner itself stalls for a memory access, it is opportunistically migrated to the core with the most recent non-runner stalls. This decreases the amount of idle time which a runner may exploit for any one memory stall, but mitigates the performance impact on the runner itself. Over time, the runner gravitates towards those applications which stall for main memory most often.

While this policy is unfair over the short term to the runner threads – which suffer more than the threads pinned to particular cores – this unfairness can be mitigated by rotating which threads serve as runners over larger time intervals.

- **Conflict:** The *conflict* policy migrates threads away from cores with execution resource contention, towards those which are experiencing many mem-

ory stalls. It conservatively prefers to leave threads in-place, minimizing interference with normal execution. When the system detects that a thread has stalled for a long-latency memory access, it considers the amount of execution-resource contention that the corresponding core has experienced recently (within the last 10,000 cycles; this is a parameter). If the conflict rate – the mean number of ready instructions unable to issue per cycle due to resource exhaustion – exceeds a threshold, the already-stalled thread is evicted and sent to wait at the inactive-thread store. When the corresponding memory access completes, the thread is sent for execution on the core whose threads have experienced the highest rate of long-memory stalls in recent history.

In order to decrease spurious thread movement, additional conditions apply: threads are returned to their previous core if the stall-rate of the candidate core does not exceed that of the previous core by a threshold (5%); the initial thread swap-out is suppressed if a pre-evaluation of the core selection criteria indicates that it is already scheduled on the "best" core; and, stalled threads are never moved away from an otherwise-empty core.

For this set of policies we do not demonstrate the cases where the number of threads is less than the number of cores (this becomes simply multi-core execution) or where the number of threads is greater than the number of SMT contexts. In the latter case, the STMP architecture clearly provides gains over a conventional architecture, but we do not show those results because they mirror prior results obtained by BMT [TKTC04]. Consider the case where we have 12 threads; a reasonable setup would have each core executing a set of three threads over time, in round-robin fashion, using the BMT mechanisms to swap an active thread for an off-core thread when an L2 miss is encountered. There would be no immediate advantage in re-assigning threads to different cores when a thread stalls for a miss, because each core already has threads available to tolerate that miss.

On the other hand, we may want to re-balance those threads occasionally, to ensure that we are running a complementary set of applications on each core; those rebalancings are best done at a much coarser granularity than the polices

described above, which target the covering of individual misses. Such rebalancing mechanisms are described in the next section.

5.4.6 Thread control policies — Rapid rebalancing

One advantage of the STMP over prior architectures, including BMT, is its ability to redistribute threads quickly between cores. Other architectures would rely on the operating system to make such changes; however, the operating system is not necessarily the best place to make such load balancing decisions. We advocate making some of these decisions in hardware, because in this architecture the hardware (1) has performance data indicating the progress of each thread, (2) has the mechanisms to context switch without software intervention, and (3) the cost of context switches is so low that it is economical to make them far more often than system software can consider. This rapid rescheduling ability allows the system both to find good initial schedules more quickly than unassisted software, and to react very quickly to later phase changes.

This group of policies considers, in particular, cases where there are more overall software threads than there are hardware execution contexts. These policies would also be effective when there are fewer threads than total contexts, as there would still be a need to consider which threads are co-scheduled and which run with a core all to themselves. We focus on the over-subscribed case, which combines stall-by-stall round-robin movement of threads to and from a core, with occasional re-shuffling among cores at a higher level. We evaluate several thread re-scheduling policies for this over-subscribed execution mode.

Our baseline for performance comparison is a model where the cores still share thread state, enabling hardware-assisted thread swapping, but the threads are partitioned exclusively among cores, and this partitioning rarely changes (i.e., at the scale of OS time-slice intervals).

The re-balancing policies considered are:

- **Symbiotic scheduler:** This policy alternates through two phases, a "sampling" phase and a "run" phase, evaluating random schedules for short intervals and then using the best of the group for much longer execution intervals.

Prior work [ST00] has found this style of policy effective for single-core SMT execution.

In a sampling phase, a sequence of random schedules – in our experiments, 19 random schedules as well as an instance of the best schedule from the previous run phase – are evaluated for performance. Each schedule is applied in turn; first, threads are migrated to the cores indicated by the schedule. Next, the threads are left undisturbed (i.e. no cross-core migrations are scheduled) for one sampling period in order to “warm up” execution resources. (Threads may still be rotated through a single core, as in Balanced Multithreading [TKTC04], during this time.) After the warm-up time has expired, the threads are left undisturbed for another sampling period, during which performance counters are collected. The counters are evaluated, then sampling continues at the next candidate schedule. The entire sequence of scheduling, warming up, and measuring each candidate in turn, makes up one sampling phase.

After the sequence of candidate schedules is exhausted, a “run” phase begins. The shared-thread control unit chooses the schedule from the sampling phase with the best observed performance, applies it, and then leaves that schedule to run undisturbed for much longer than the duration of an entire sampling phase (20 times longer, in our experiments). Afterward, the next sampling phase begins.

- **Medium-range predictor:** This policy incorporates performance observations collected over many sampling periods, considering the correlation of overall performance with the pairs of applications that are co-scheduled in each. These observations are summarized in a compact data structure, a table from which predictions can be generated about arbitrary future schedules.

  Simplified, the schedule performance predictor operates by maintaining a matrix that, with application IDs as coordinates, indicates the mean performance measured across all samples when those applications were resident on

the same core. The predictor takes as input (*schedule*, *performance*) tuples of performance observations; the overall performance is added to the matrix cells corresponding to each pair of co-scheduled applications. A second matrix is maintained with sample counts to allow for proper scaling. Any candidate schedule can be evaluated against these matrices to yield a performance estimate. The matrices are continuously aged over time, so that more recent observations carry more significance than older ones.

Additional details of the *medium-range predictor* merit discussion. To make use of its collected performance samples, we have developed a straightforward greedy algorithm which synthesizes new "good" schedules. These schedules are good in the sense that, when evaluated in the context of past measurements, their forecast performance tends toward optimal. That is, given collected performance and count observation matrices P and C, and a fixed performance forecasting function fc, the algorithm outputs a schedule s which tends to maximize the value of $fc(P, C, s)$. This is a heuristic, however, and optimality bounds are not established.

Note that this scheduler does not guarantee optimal future performance; in a sense, it runs the performance predictor in reverse, directly constructing a new schedule which the predictor considers to be good, tying the overall resulting performance to the accuracy of the predictor. The greedy schedule synthesizer constructs a schedule from the matrix of aggregated performance measurements by starting with an empty schedule, then successively co-scheduling the pair of applications which corresponds to the next-highest performance point in the matrix. (Several additional conditions apply, to ensure reasonable schedules result, to account for applications scheduled solo, etc.)

One additional use for the greedy schedule synthesizer is to generate schedules which represent the least-sampled portions of the schedule space. This is accomplished by creating a performance-sample matrix with each element set to the negative of the corresponding element in the sample-count matrix, and running the algorithm on that; since the greedy algorithm attempts to maximize the resulting expected-performance sum, it selects the near-minimal (negated) sample

counts.

The *medium-range predictor* operates by using the greedy schedule synthesizer to construct a good schedule. This schedule is run for several sampling periods – 20, in our experiments – followed by an alternate schedule to ensure diversity in the measurement space. We generate alternate schedules either at random (*mrp-random*), or specifically targeting the least-sampled portions of the schedule space (*mrp-balance*).

5.5 Methodology

We evaluate the Shared-Thread Multiprocessor and a variety of scheduling policies through simulation, using a modified version of SMTSIM [Tul96], as introduced in Chapter 2. Starting with the multi-core baseline described in Section 5.3, we implement the STMP mechanisms and policies described in Section 5.4. We do not simulate the computation needed to implement the on-line thread control policies; however, these policies utilize very simple operations on small numbers of performance samples, which we expect could be performed with negligible overhead.

5.5.1 Simulator configuration

We assume a four-core multiprocessor, with the execution cores clocked at 2.0 GHz; for ease of accounting, all timing is calculated in terms of this clock rate. Table 5.1 lists the most significant of the baselines system parameters used in our experiments. While the 500 cycle main-memory access latency we specify would be high for a single-core system with two levels of caching, it is not unreasonable for a more complex system: in separate work, we have measured best-case memory access latencies of over 300 cycles on four-way multiprocessor hardware, with cores clocked at 1.8 GHz.

Table 5.1: Architecture details.

Parameter	*Value*
Pipeline length	8 stages minimum
Fetch width	4
Fetch threads	2
Fetch policy	ICOUNT
Scheduling	out-of-order
Reorder buffer	128 entries
Integer window	64 insts
FP window	64 insts
Max issue width	4
Integer ALUs	4
FP ALUs	2
Load/store units	2
Branch predictor	8 KiB gshare
BTB	256-entry, 4-way
Cache block size	64B
Page size	8 KiB
Cache replacement	LRU, write-back
L1 I-cache size/assoc.	64 KiB/4-way
L1 D-cache size/assoc.	64 KiB/4-way
L1 D-cache ports	2 read/write
L2 cache size/assoc.	8 MiB/8-way
L2 cache ports	8 banks, 1 port each
ITLB entries	48
DTLB entries	128
Load-use latency, L1 hit	2 cycles
Load-use latency, L2 hit	13 cycles
Load-use latency, memory	500 cycles
TLB miss penalty	+500 cycles

Table 5.2: Component benchmarks.

Benchmark	*Input*	*Fast-forward* ($\times 10^6$)
ammp		1700
art	c756hel, a10, hc	200
crafty		1000
eon	rushmeier	1000
galgel		391
gap		200
gcc	166	500
gzip	graphic	100
mcf		1300
mesa	-frames 1000	763
mgrid		375
parser		650
perl	perfect	100
twolf		1000
vortex	2	500
vpr	route	500

5.5.2 Workloads

We construct multithreaded workloads of competing threads by selecting subsets of the SPEC2000 benchmark suite. Starting with a selection of the 16 benchmarks as detailed in Table 5.2, we construct workloads for a given thread count by selecting subsets of the suite such that each subset contains the desired number of threads, and so that all subsets with a given thread count, when taken together, yield roughly the same number of instances of each benchmark. The 16 benchmarks were chosen arbitrarily to allow for this convenient partitioning, without requiring an excessive number of simulations to evenly represent individual benchmarks.

Table 5.3 details the composition of the workloads used in this study. We simulate each multithreaded suite until ($\# \, threads \times 200 \times 10^6$) overall instructions have been committed.

This evaluation of our thread-migration system does not make special accommodations for different classes of workloads – e.g. soft real-time, co-scheduled shared-memory – though such workloads are not intrinsically excluded. We focus

Table 5.3: Composite workloads.

ID	*Component Benchmarks*
5a	ammp, art, crafty, eon, galgel
5b	gap, gcc, gzip, mcf, mesa
5c	mgrid, parser, perl, twolf, vortex
5d	ammp, crafty, galgel, gcc, mcf
5e	mgrid, perl, twolf, vortex, vpr
5f	art, eon, gap, gzip, mesa
6a	ammp, art, crafty, eon, galgel, gap
6b	gcc, gzip, mcf, mesa, mgrid, parser
6c	mgrid, parser, perl, twolf, vortex, vpr
6d	ammp, eon, gcc, mesa, vortex, vpr
6e	art, crafty, galgel, gzip, mcf, perl
7a	ammp, art, crafty, eon, galgel, gap, gcc
7b	gzip, mcf, mesa, mgrid, parser, perl, twolf
7c	art, eon, gap, gzip, mesa, vortex, vpr
7d	ammp, crafty, galgel, gcc, parser, twolf, vpr
7e	gap, mcf, mgrid, perl, twolf, vortex, vpr
8a	ammp, art, crafty, eon, galgel, gap, gcc, gzip
8b	mcf, mesa, mgrid, parser, perl, twolf, vortex, vpr
8c	ammp, crafty, galgel, gcc, mcf, mgrid, perl, vortex
8d	art, eon, gap, gzip, mesa, parser, twolf, vpr
10a	ammp, art, crafty, eon, galgel, gap, gcc, gzip, mcf, mesa

on the SPEC2000 suite as a source of diverse execution behavior largely to streamline experimental evaluation; extending the system to consider explicit information about alternate workload types is an interesting avenue for future research.

5.5.3 Metrics

As discussed in Chapter 2, evaluating the performance of concurrent execution of disparate workloads presents some challenges; using traditional metrics such as total instructions-per-cycle (IPC) tends to favor any architecture that prefers individual benchmarks which exhibit higher IPC. We use a metric with a built-in fairness criterion, *weighted speedup*. See Section 2.2 for a discussion of weighted speedup, specifically Section 2.2.1 and Equation 2.1 for details of primary performance metric used in this chapter. Several dynamic schedulers evaluated in this

chapter utilize a modified version of weighted speedup, named *interval weighted speedup*, which is adapted to provide on-line results during simulation. See Section 2.2.3 for discussion, and Equations 2.3 and 2.4 for the relevant equations.

To underscore the difference between performance metrics used in this work: *Weighted speedup* is the metric we use to evaluate and performance. *Interval weighted speedup* is an online metric used by our architecture to evaluate or estimate the effectiveness of schedules and make scheduling decisions, but it does not appear in any results.

5.6 Results and Analysis

In this section, we evaluate the performance of a *Shared-Thread Multiprocessor*, considering several distinct modes of operation. All results are reported for the hardware configuration as described in Section 5.5.1, unless otherwise noted.

5.6.1 Potential gains from memory stalls

This section demonstrates one axis of the opportunity for the STMP. Even a moderately-sized core with SMT support frequently experiences idle cycles where no execution progress is being made. We see in Figure 5.2 the total amount of time in which cores are effectively idle – with all active execution contexts unable to be used – due to long-latency memory operations, as described in Section 5.3.3. The values shown are the fraction of the overall execution time that each core spends completely idle, summed across all cores.

From this result we see that even when the full SMT capacity of the cores is used (8T), there remains significant idle execution capacity. Simply applying STMP, even without dynamic thread movement policies, should significantly reduce the idle cycles. Additionally, we see that the problem is more acute when not all of the cores are able to exploit SMT. We see that in many instances, the equivalent of an entire additional "virtual core" is available, if we can find a way to effectively utilize the combined lost cycles.

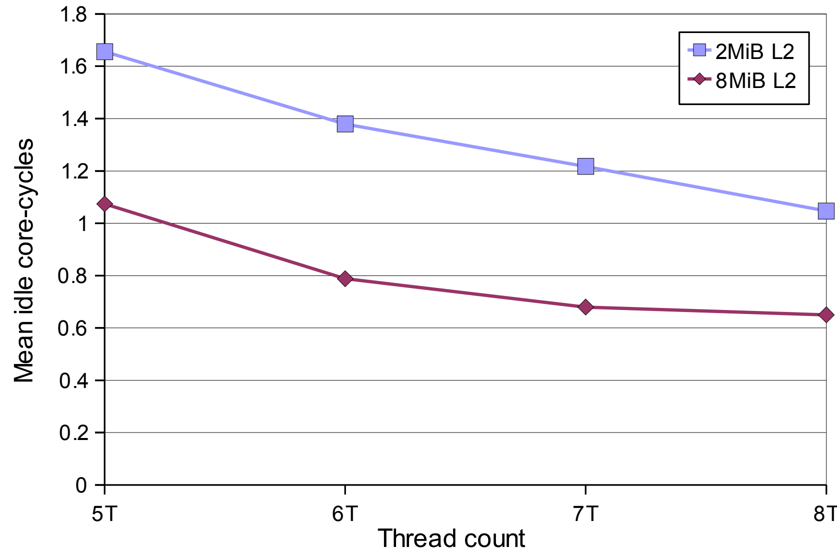


Figure 5.2: Total core-cycles (per processor cycle) where a fully-occupied core is effectively idle due to stalled memory instructions. Means are taken across all workloads for a given thread-count. Values above 0 indicate potential gains.

5.6.2 Rapid migration to cover memory latency

This section evaluates the use of the STMP to rapidly migrate individual threads between cores in response to stalls for memory access.

Figure 5.3 shows the weighted speedups achieved by several different thread-migration policies, with the five-threaded workloads described in Section 5.5.2. (Recall that our processor has four two-way SMT cores, for a total of eight execution contexts.) We use the five-thread case to exercise our policies because it is a region where we expect to see frequent imbalance.

For the sake of comparison, we also evaluate each workload under all possible static schedules, wherein threads do not migrate during execution. The *best-static* result shows the highest weighted speedup achieved by any static schedule for each workload. *mean-static* shows the arithmetic mean weighted speedup for each workload over all static schedules; it reflects the expected performance from assigning threads to cores at random, then not moving them during execution.

The *best-static* result represents a reasonable goal for our policies, as it is unattainable without oracle knowledge. In theory, we could beat *best-static* by ex-

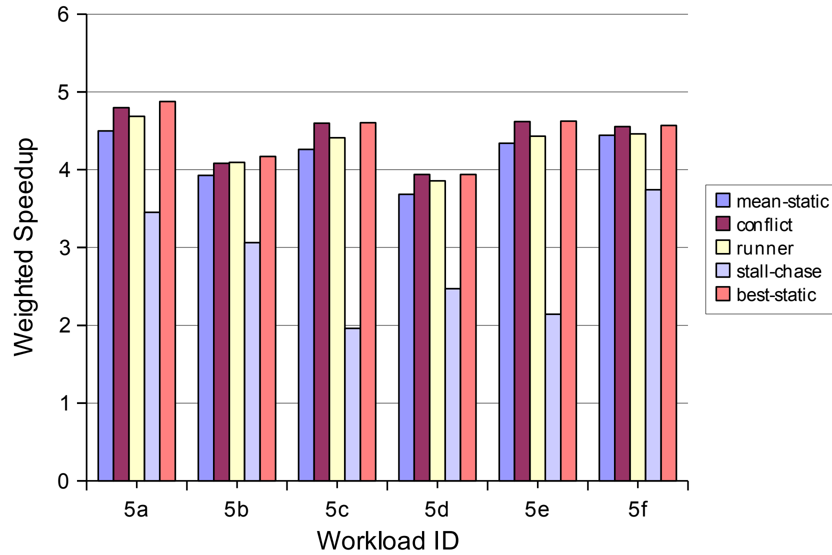


Figure 5.3: Performance of several stall-covering schemes on five-thread workloads.

ploiting dynamic changes in the workload, but none of these results achieve that. *mean-static* represents a reasonable baseline, as it represents what an OS scheduler might do, given the STMP hardware but no useful information about thread grouping. In these results, weighted speedup is computed relative to single-thread execution. Thus, for a five-thread, four-core configuration, a weighted speedup (WSU) of 4.0 is the minimum expected, and a WSU of 5.0 would indicate we are achieving the performance of five separate cores. Thus, the range of improvements we are seeing in these results is definitely constrained by the limited range of possible WSU values.

The *conflict* policy performs very close to the oracle static scheduler. This policy succeeds because it inhibits switching too frequently and it targets threads known to be on oversubscribed cores for movement. Because it only moves these threads, the instances where movement significantly degrades progress of a thread are lessened: the selected thread was struggling anyway.

We see that the *runner* policy performs significantly better than *stall-chase*. The two schemes are similar, with the primary difference that the former requires no core's L1 cache to support the working set of more than two threads at a time, significantly mitigating the lost L1 cache performance observed for the latter policy.

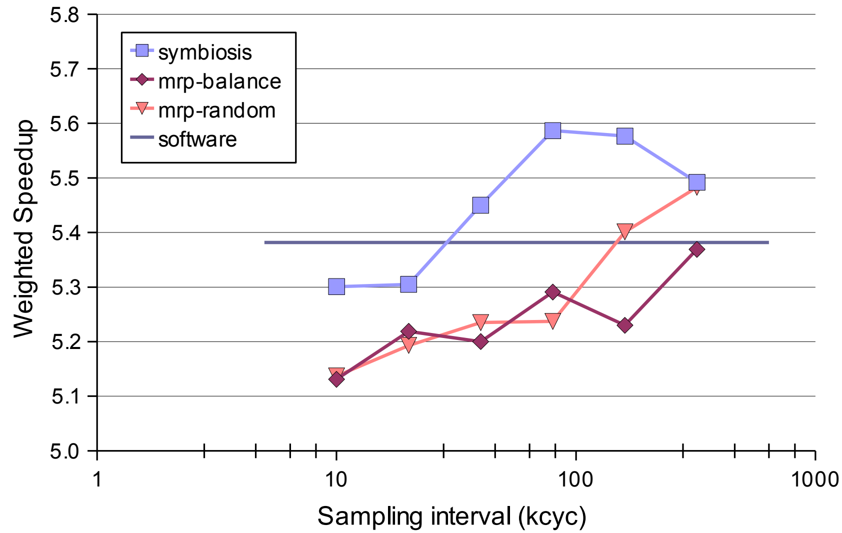


Figure 5.4: Performance impact of several dynamic-scheduling algorithms on a ten-thread workload, versus sampling interval.

Although *runner* provides performance gains in all cases over the baseline, it always suffers slightly compared to the less aggressive *conflict* because the frequency of ineffective migrations is still high.

Thus, we demonstrate two schemes that show improvement over a conventionally multithreaded chip multiprocessor. *conflict*, in particular, allows the processor to match the performance of an ideal oracle schedule: one which perfectly places threads in the optimal core assignment, with no overheads associated with identifying the optimal assignment at runtime. This result is very significant, because it shows that it is indeed possible to support multithreaded-style thread context-sharing between distinct CMP cores; in fact, they can share contexts quickly enough to even hide frequent memory latencies.

5.6.3 Rapid migration for improved scheduling

This section explores the second expected benefit of the STMP architecture, the ability to do frequent re-evaluation of the thread-to-core assignment. These results focus on policies which migrate threads between cores much less frequently than those of the previous section, but potentially far more frequently than could

be done by system software alone. Figure 5.4 shows the performance of some of the policies discussed in Section 5.4.6.

The baseline software scheduler – which uses our best symbiotic scheduling mechanism, but at a time scale possible in software – is shown as the horizontal line. It should be noted that the baseline makes use of our STMP architecture, e.g. allowing three threads to share two SMT contexts an a core, and it uses our best mechanism to re-evaluate schedules; the baseline only lacks the ability to re-evaluate schedules at a rate faster than OS time-slice intervals.

We see that the symbiotic scheduler is able to achieve some gains over the software scheduler for intermediate levels of rescheduling. The more often we re-sample and re-schedule the threads, the quicker we are able to react to changes in the workload.

However, the reason that the results peak in the middle is that there are offsetting costs to rescheduling too frequently. The first reason, which our results have shown to be less of a factor, is the cost of migrating threads (primarily the cold cache effects).

The second, more important but less obvious factor, is our reliance on performance predictions, which become inherently less reliable as the sampling intervals shrink. All of our schemes depend on some kind of predictor to estimate the schedule quality. For the symbiotic scheduler, the predictor is a direct sample of each schedule; for the others, they use predictors based on some kind of history. The inherent advantages of rescheduling more quickly are offset by the inaccuracy of the predictor. For example, the shorter our sampling intervals in the symbiosis scheduler, the more noisy and less reliable the sample is as a predictor of the long-term behavior of that schedule.

The *medium-range predictor* provided more accurate predictions of future performance. While predictions proved relatively accurate, it did not outperform the much simpler *symbiotic scheduling* policy; the predictor-based scheduler still tended to schedule more thread movement than the symbiosis policy, losing performance to scheduling overhead. Of the two predictor-based policies shown, *mrp-random* tended to marginally outperform *mrp-balance*. This is counterintuitive:

both policies explicitly deviate from the current "expected best schedule" in order to explore new ground, with the latter explicitly targeting the least-sampled regions of the schedule space; one would suspect that *mrp-balance* would thus be more successful at introducing useful diversity at each step. However, as the predictor-based scheduler learns more about the performance correlation between threads, it naturally begins to avoid co-scheduling threads which do not perform well together. Over time, then, some of the the under-sampled regions of the sample space are avoided specifically due to bad performance, and the *mrp-balance* scheduler can force execution back into these regions. The *mrp-random* scheduler has no such bias.

These results demonstrate that with a very simple hardware re-scheduler, there is clear value to load-balancing and re-grouping threads at rates significantly faster than possible in a software scheduler. These results indicate that in this type of aggressive multithreaded, multi-core processor, an operating system is no longer the best place to make all scheduling decisions.

5.7 Summary

The Shared Thread Multiprocessor extends multithreading-style thread interaction to operate between the cores of a chip multiprocessor. In this chapter we have demonstrated that, beyond the gains of previous work – where external thread storage is used to improved the performance of a single SMT processor – the STMP can be exploited in two unique ways:

- At fine time scales, when long-latency memory stall events create transient load imbalances between cores, we can improve overall throughput by quickly moving a thread between cores to exploit otherwise idle resources; hardware thread-sharing support is fast enough that we can move threads at the level of individual stalls.

- At coarser time scales, we are able to repartition the thread-to-core mapping quickly and efficiently, allowing us to re-evaluate scheduling decisions and

react to emerging application behavior. While sampling too frequently rendered scheduling decisions unreliable, we identified a middle ground in which it was profitable to make decisions more quickly than possible with software mechanisms alone.

In both scenarios, the efficient implementation of scheduling primitives in hardware has enabled resource-aware scheduling of parallel workloads – typically performed by operating systems or run-time libraries – to be performed at much finer granularity, improving system-wide performance.

Acknowledgements

The work in this chapter was supported in part by NSF grant CCF-0702349 and Semiconductor Research Corporation grant 2005-HJ-1313.

This chapter contains material from "The Shared-Thread Multiprocessor", by Jeffery A. Brown and Dean M. Tullsen, which appears in *Proceedings of the 2008 ACM International Conference on Supercomputing (ICS)*. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2008 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 6

Fast Thread Migration via Working Set Prediction

6.1 Introduction

Increasing processor core count has become the preferred avenue for achieving performance growth. Continued proliferation of largely independent cores within systems will leave us increasingly dependent on high levels of workload thread-level parallelism (TLP) to sustain the desired scaling of system performance. This will require the development of new language, compilation, and execution models that offer enough TLP to sustain high throughput on future many-core systems. Continuing along on the trajectory introduced in Chapter 1, in this chapter we focus on hardware support necessary to *enable* that development.

An important barrier to the viability of such development is the inability of processors to execute short threads efficiently: while efficiently initializing the control and register state of a new thread is straightforward – for example, via the mechanisms discussed in Chapter 5 – the overhead of populating a core's caches with a suitable working set dominates, dooming young threads to perform poorly. Aggressive threading schemes will be most effective if they have the freedom to exploit parallelism in threads which are tens or hundreds of instructions long. However, current machines cannot move or fork execution between cores profitably

at ranges below tens or hundreds of *thousands* of instructions; a great deal of potential parallelism is unavailable due to the cost of moving or starting a thread.

Beyond the untapped potential of parallelism available to short threads, we also see frequent demand for efficient working-set migration in traditional parallelizing schemes, both at loop-level – where threads spawned for a loop iteration inherit the state of the serial code leading to the loop – and at task-level, where a parallel task inherits the state of its caller.

Several recently proposed applications of parallel hardware also benefit from efficient working-set migration. Speculative multithreading [SBV95, HWO98, QMS+05, SM98] breaks serial execution into potentially parallel threads, with each thread inheriting the execution context of the previous thread. Helper threads [CSK+99, CWT+01, ZS01] also utilize parallel hardware for speedup, but without actually offloading computation; even in these cases each new helper thread executes within the same address space as the main thread, inheriting its memory state.

Heterogeneous multi-core proposals [KFJ+03, KTR+04] move threads between cores of different capabilities in order to optimize power-performance-area trade-offs. These proposals use frequent sampling, via heavy thread migration, to discover beneficial thread-to-core mappings. The initial studies migrated threads conservatively, due to the high cost of migration; presumably, the lower that cost, the more quickly the architecture can adapt, and the higher the potential gains.

Other research migrates threads at points when thread-level parallelism changes [AGS05], and at system calls [MMB+08, CWS06]. Software data spreading [KST10] migrates threads at compiler-determined points in order to effectively utilize the aggregate capacity of multiple private caches. Even when multi-cores are not being exploited for performance, thread migration can be frequent; e.g. some schemes advocate core-hopping for thermal management [CGG04].

These techniques all share the property that a thread begins or resumes execution on one core after the working set it needs to execute at full speed has been built up on another core (primarily in the caches). As a result, the "working set" of the thread must migrate from one core's caches to another's.

In conventional systems, the primary mechanism for working set migration is executing code on the target core. Execution generates demand misses which retrieve the necessary data, from either another core or from a shared level in the cache hierarchy. This is a particularly inefficient mechanism for building a working set after migration, since the speed at which data migrates is limited by the rate at which this "migration engine" – the executing code – can generate additional memory requests, but that execution itself is severely hamstrung by cold cache effects. This vicious cycle limits migration speed to well below what the interconnect and caches themselves are capable of sustaining.

In this chapter, we explore several mechanisms for predicting and migrating the working set of a thread from one core to another. We introduce a three-step approach to enable working set migration: first, we augment each core with simple hardware to *capture* the access behavior of threads as they execute. Next, when deactivating an executing thread, we *summarize* the captured behavior into a compact representation of likely future instruction and data accesses, and transfer that summary along with other thread state. Finally, we *apply* the summary data at the new core, using it to rapidly prefetch upcoming memory blocks.

Our primary purpose in this chapter is to evaluate a diverse set of potential mechanisms which strive to capture access behavior, and measure how effectively each of these mechanisms predict future accesses. Though we present a framework with which we evaluate schemes of varying complexity, we achieve our best performance results using a combination of the most inexpensive capture-tables. This is encouraging: useful performance gains are realized at low cost using small, low-complexity tables, maintained using only the address stream of an executing thread.

We make the following important contributions: We demonstrate that demand-fetching is not a reasonable mechanism to fill the caches post-migration, resulting in highly serial accesses. We show that conventional hardware prefetchers are not useful over the time intervals in which performance degradation is the most dire. We show that, unlike the common case of execution without migration, instruction-stream misses are more common and significantly more critical to

post-migration performance than data stream misses. We demonstrate that transferring the actual contents of the private caches is surprisingly ineffective, and over several sampling intervals, is worse than doing nothing. With the addition of just a few small, simple tables to monitor access activity, along with a prefetcher that exploits the contents of those tables, we achieve as much as a 2X performance boost for short threads.

6.2 Related Work

Chapter 5, along with previous work [TKTC04], describes support mechanisms for migrating register state in order to decrease the latency of thread activation and deactivation; however, performance subsequent to migration suffers due to cold-cache effects. This chapter complements those studies, as it specifically addresses post-migration cache misses which limit the gains of those techniques. Choi, et al., explore the complementary problem of effective branch prediction for short-lived threads [CPT08].

Stream buffers [Jou90, PK94] introduce small, associative structures which track ongoing data access patterns. More advanced stream buffers [SSC00] extend this idea, allowing for an advanced predictor to be shared among many streams. Our scheme is built similarly, with the added ability to extract and transfer summaries. Our techniques can leverage much of the hardware that is already present in such stream buffers. We model a discrete hardware predictor-directed stream buffer in the style of [SSC00], omitting the shared Markov predictor, as part of our baseline. We also add the ability to transfer stream-buffer state as a potential working set predictor.

Sair et al. [SSC02] survey several successful prefetching approaches, and demonstrate an approach to classifying memory access behaviors in hardware: the memory access stream is matched against behavior-specific tables operating in parallel. They use this information to quantify the miss patterns of several benchmark suites. We utilize some similar structures in the *capture* stage of our working-set migration.

Speculative Precomputation [ZS01, CWT+01] targets specific memory instructions which are observed to degrade performance due to poor cache behavior. They employ helper threads, distilled from the original code, which prefetch future data into the cache, on multithreaded or CMP [BWC+02] architectures. Focusing on misses, these schemes target the subset of the future working set which is not currently cached.

Runahead Execution [MSWP03] prefetches by speculatively executing the same application, but ignoring dependences on long-latency misses. This seems well-suited to our need to prefetch what would normally be cache hits in addition to misses, and able to cover a substantial amount of working set with little metadata overhead. However, this scheme is hamstrung in the post-migration environment by the lack of instruction cache state at the target core; stalling to service I-cache misses serializes short-term prefetching.

Several dependence-following schemes [APD01, CSCT02] enable prefetching dependence chains through memory. While valuable, these techniques alone are of limited utility immediately after a migration, due to their serial progression. (We incorporate this style of prefetching with our *pointer-chase* sub-table.)

Dead-block prediction [LFF01] introduces a predictor which tracks the lifetimes of L1-resident cache blocks, predicting when many cache blocks are no longer needed and may be evicted before replacement demands it. An additional table is then used to predict likely successors for early-evicted blocks. They use the dead-block predictor and the correlations it exposes to prefetch likely misses, and use the freed L1 storage as a prefetch buffer. That work motivates ours in the sense that we also find that caches often hold much more than the current working set; put another way, the current cache contents are not the best predictor of future working sets.

6.3 Baseline Multi-core Architecture

In this chapter, we study a chip multiprocessor (CMP) design consisting of four identical cores. Each core has a four-way superscalar, out-of-order execution

engine. Because of its ability to exploit memory level parallelism, this core will be *less* sensitive to cache migration effects than a less aggressive design. Cores have private first-level instruction and data caches, and a private second-level unified cache; see Figure 6.1. Off-chip memory is accessed via a shared four-channel off-chip memory controller. Specific parameters of the core and memory subsystems are detailed in Section 6.6.1. The four cores communicate over a shared bus. Caches are kept coherent with a MESI coherence protocol [PP84] and snooping; our technique would readily apply to systems with more scalable interconnects and more numerous cores.

In this study, virtually all misses (over the post-migration regions of interest) are serviced core-to-core rather than off-chip; consequently, we expect our results to be nearly identical to what would be observed in a system with a shared on-chip L3 cache, e.g. Intel Nehalem [Int08]. We confirm this is in Section 6.7.8.

We assume the cores of our CMP feature hardware support for thread activation and deactivation, as discussed in detail in Chapter 5 and in prior studies of thread scheduling [TKTC04]. While those works used hardware support to implement complex scheduling and time-sharing policies, in this chapter we use it as a simple and explicit mechanism for adding and removing executing threads from cores. Most speculative multithreading proposals also assume hardware support for thread movement and spawning. Traditional system-software-driven migration would have much higher overhead, and expose the cache migration costs less; however, even in that scenario our experiments confirm that the cache migration costs are still the dominant cost of migration in most cases. We believe that direct OS involvement in all thread movement is quickly ceasing to be a viable model, but even the OS overhead for migration can be significantly reduced from current levels [SMM+09].

6.4 Motivation: Performance Cost of Migration

There are many execution scenarios which require the working set of a thread on one core to migrate to another: system load balancing, thread spawn-

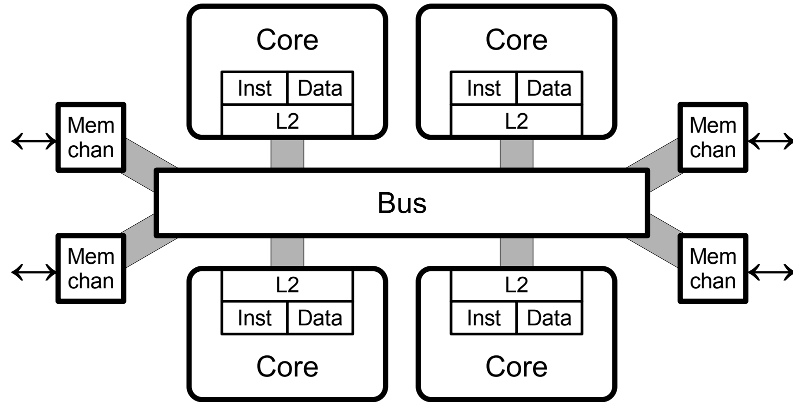


Figure 6.1: Baseline multi-core processor.

ing, loop-level parallelism, task-level parallelism, helper threads, speculative multithreading, single-ISA heterogeneous multi-core adaptation, thermal management, etc. Virtually all of these will become more common as core counts increase, given the already lower costs of communication; in nearly every case, these mechanisms will be even more effective if the cost of migrating thread state is decreased.

Although the principles from this research apply in all of these cases, for clarity of evaluation, we specifically target the case of single-thread migration at arbitrary points in the program. This most directly reflects migration for load-balancing, migration for thermal management, and migration-based sampling of schedules on heterogeneous multi-cores. Our results would also directly apply to speculative multithreading, transaction-based parallel code, etc., where migration instead tends to occur at particular points in the program. Our evaluation migrates at arbitrary program points; this considers – in an informal sense – the expected behavior over a large number of possible thread trigger points.

To evaluate various migration support mechanisms, we repeatedly move a single executing thread among a set of cores, at arbitrary points in the program. There are several costs incurred in migrating a thread: transferring thread register state, transferring TLB state, recreating branch predictor state, etc.; however, the largest amount of program state on a core resides in the caches. As a result, the cost of transferring cached state dominates thread re-start performance. In

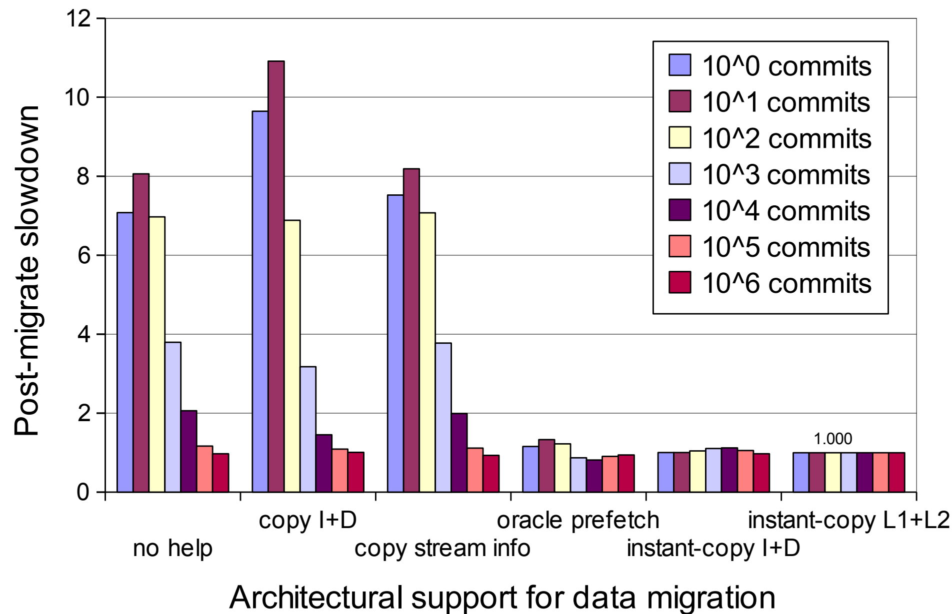


Figure 6.2: The cost of migration, in reduced instruction throughput, for various assumptions about the migration of data. The baseline is instant replication of all private caches.

addition, that state is moved very slowly, because it is only demand-fetched as the thread executes on the new core, and after migration that thread is executing (and demand-fetching) extremely slowly.

Figure 6.2 gives the result of an experiment that illustrates the cost of migration and the potential to reduce the cache-related portion of that cost. We force a single thread to migrate round-robin among four cores, moving every 1 million commits, and record the time it takes to start-up and commit the next $1, 10, 100 \ldots 10^6$ instructions after each migration. We perform this experiment across our benchmark suite, with varying amounts of architectural (and oracle) support for migration. We show slowdown relative to the ideal case where all cache contents (instruction, data, L2) are instantly transported to the new core for free. It takes, on average, 7 times as long to commit the 100th instruction in the default migration case – "no help" – compared to cost-free cache migration. (We assume there is a background load that causes cache state to get evicted before the thread returns to the original core, but which does not otherwise impair the thread.)

The impact of several migration-support schemes is shown. For the feasible schemes, "copy I+D" bulk-copies both first-level caches by transferring a list of their tags to the new core and then fetching them via core-to-core transfers; "copy stream info" transfers just the metadata from the first core's hardware prefetch stream buffer to the second, allowing it to resume fetching any streams it was following. For the idealized schemes, "oracle prefetch" uses perfect knowledge of near-future accesses to fetch the required blocks at the new core, requesting them via the memory hierarchy as the thread restarts and modeling the costs of these requests; "instant-copy I+D" instantly transfers the contents of the first-level instruction and data caches to the new core, cost-free; finally, "instant-copy L1+L2" instantly transfers all cached data, cost-free.

This graph provides several key insights. First, we see that unless a thread executes on a core for many instructions before being migrated again, the cost of migration is not amortized in the realistic schemes. At 10,000 commits, the cost is still very high (2X slowdown); for shorter threads, migration cost is extreme. Note that several speculative multithreading proposals routinely execute threads under 100 instructions [MG02], as do helper-thread proposals [ZS00]. Several transactional memory programs (for Transactional Coherency and Consistency) showed average transaction lengths in the low hundreds [CCM+06].

We also see that transporting whole caches proactively – the "copy I+D" case – is not particularly effective: there is too much data, and not all of it will be accessed, either soon or at all. It is worse than doing nothing over short intervals, and takes about 10,000 instructions to be amortized enough to approach break-even. While not shown, the cost of copying the larger L2-resident state is even worse.

As an alternative, we could copy prefetcher state; this is the "copy stream info" case. Over the short term, we see this is more effective than moving the entire L1 cache state, since prefetcher state is small and directly targets future accesses. It is still not particularly effective, however, since it represents only a small fraction of the future working set: the stream buffer is built to target future *misses* in the data stream, data which is expected to be absent from the previous cache; what

we really want is something similar to the stream buffer, but trained on the entire access stream, not just the misses. Furthermore, this scheme does not prefetch the future instruction stream. Immediately after migration, there is a large demand for instructions. The lack of I-stream prefetching is greatly exacerbated by the inability to overlap multiple demand-fetched I-misses.

At the short time scales we are most interested in for migration support, conventional hardware prefetchers are unable to contribute much: they do not have enough time to learn about an incoming thread's behavior, since the thread itself is struggling to execute.

6.5 Architectural Support for Working Set Migration

In the previous section we demonstrated that observing and characterizing the miss stream is not sufficient to cover many migration-related misses; we need to characterize the *access* stream instead. We construct a working set predictor which works in three stages. First, we observe the access stream of an executing thread, and *capture* patterns and behaviors. Second, we *summarize* this behavior and transfer the summary data to the new core (we try to minimize the size of this summary, because this transfer can interfere with the transfer of register state and other data). Third, we *apply* the summary via a prefetch engine on the target core, to rapidly fill the caches in advance of the execution of the migrated thread.

First, we will discuss the architecture of a set of potential capture engines. In this discussion, we consider a number of hardware tables that capture a wide variety of access patterns to better evaluate the design space, and determine which mechanisms are most effective in this scenario. This system operates on information about committed instructions, and does not directly supply data to any part of the pipeline, allowing for a conservative implementation outside of critical timing paths.

Table 6.1: Activity record field superset.

Field	*Description*
time	Cycle of corresponding commit
addr	Virtual address of access
width	Width of access
op-type	I-fetch, Load, or Store
app-id	Address-space ID
pc	Taken-branch PC (only for fetch after a taken branch)
addr-regnum	Logical register used for address
offset	Address offset immediate value
data-regnum	Logical source/destination register
addr-regval	Input value from address register
data-regval	Input value from data register
service-level	Most distant level of memory hierarchy contacted
mem-delay	Total time from address generation to memory completion

6.5.1 Memory logger

Within each core, we add a *memory logger*, a specialized unit which records selected details of recent memory activity. This unit passively observes the current thread as it executes, but does not affect execution.

From the existing structures, details of each committed memory instruction and instruction cache access are collected into an *activity record*, which is sent to the memory logger for analysis. This can be implemented conservatively outside of the main pipeline, incurring extra communication latency if need be. If the memory logger becomes swamped with data and is unable to accept new input temporarily, records may safely be discarded; only the fidelity of summary information will suffer. Table 6.1 shows the fields collected in each activity record. This is a union of fields used by all the potential capture mechanisms; the set can later be pared down (*significantly*) to match the particular subset of capture tables actually implemented.

The memory logger can be implemented as one or more small content-addressable memory (CAM) tables, with associated control logic. These tables are indexed using various portions of the information in an activity record. Each of the individual table structures is tailored to capture a specific class of memory access

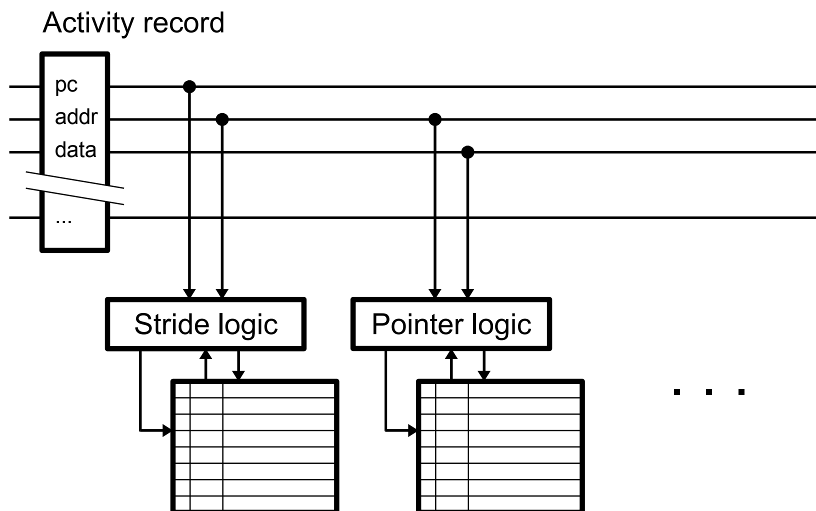


Figure 6.3: Memory logger overview.

pattern: one table tracks striding accesses, another tracks pointer traversals, etc. Figure 6.3 shows an overview of the memory logger architecture.

As we begin this study, we start out concerned less with the cost of the memory logger or component tables, and more with exploring a variety of possibilities. After considering schemes of varying complexity with all of the logger fields available, we will see that we can drastically pare down the set of fields, achieving our best results relying only on the *addr* field for both I-stream and D-stream accesses.

Lookup, replacement, and aging

Each potential memory logger table uses fully-associative lookup and a simple gated-round-robin replacement policy, similar to "second-chance" page-replacement in operating systems. (These lookup and replacement policies are not intrinsic to the design of the system; more typical set-associative CAMs may be used instead.) Each table maintains per-entry match statistics.

To track the expected accuracy of individual tables, we use a simple scheme to obtain an age-adjusted hit metric. Every epoch (a certain number of memory operations executed), the cumulative number of hits is halved (shifted right), and

the number of hits for the current epoch is added in. The choice of epoch size also conveniently establishes the maximum value of the hit counter.

Table types

Each candidate table within the memory logger targets a specific type of access pattern. (Note that we are not proposing these tables themselves as novel prefetching schemes; several of the underlying ideas are discussed in Section 6.2, and [SSC02] surveys several more.) Our tables are:

- **NextBlock{Inst,Data}:** These tables detect sequential block accesses, for both instruction and data accesses. Lookup is performed with the block-aligned memory address; beyond the common hit and usage counters, the only information associated with each address is its presence. On a hit, the block address is incremented to the next block. Similarly, on a miss the next block address is written into the address field. We use two of these tables: one for instructions, and one for data.

- **StridePC:** This table tracks individual instructions which walk through memory in fixed-sized steps. Lookup is performed using the program counter (PC). For each PC, the stride table tracks the previous memory address and the difference between the previous two memory addresses. On a PC match, the new stride is compared with the stored stride; if they match, this is a hit. If the PC or stride do not match, a new entry is allocated.

- **Pointer, Pointer-chase:** With these, we seek to capture the set of active pointers: addresses which are used to store other addresses. Table lookup is performed using the effective address of a memory reference, while replacement is performed using the data value read by a load instruction. Matching in this manner, we detect values output by previous loads which are used as inputs to subsequent memory operations (including instruction fetches), similar to pointer-cache [CSCT02] and dependence-based [RMS98] prefetching. Replacement is only performed when the memory operation being considered is an aligned, pointer-width load to a writable register. We consider

two variants: *Pointer* just tracks recently loaded values which are later used as addresses, without following them, while *Pointer-chase* replaces an entry with the returned value when a load match is observed. In this manner, *Pointer-chase* detects and follows the latest node in list traversals, and is capable of following those lists further at prefetch-time.

- **Same-object:** This table captures accesses to ranges of memory which use a common base address, as is common for structure access and object-oriented code. Table lookup takes place using the "base address" associated with a memory operation, rather than the effective address. This takes advantage of the "base+offset" addressing mode available in many instruction sets. The same-object table tracks the minimum and maximum offset values for matching loads, learning the extents of currently-active objects. This table ignores operations with negative offset values, or with the base address sourced from the stack pointer or global pointer registers.

- **SPWindow:** In order to prefetch relevant data near the top of the stack, we simply record the value of the stack pointer register. This does not require actual table storage; the prefetcher fetches a window of data near the recorded stack pointer value, hoping to get the current stack frame of local variables.

- **{Inst,Data}MRU:** These tables simply record the most recent addresses accessed. If the current access is not a hit in the table, it replaces an existing entry. We have one table for instructions and one for data. These track addresses at a coarse granularity of four-cache-block alignment, allowing them to cheaply account for a larger number of blocks without increasing the amount of tag maintenance.

- **BTB:** This table captures taken branches and their targets. Table lookup and replacement use the PCs from I-fetch accesses which immediately follow a taken branch in the correct-path instruction stream, i.e. branch target PCs. Each table entry records the most recent inbound branch PC for that target.

- **BlockBTB:** This is a cache-block-aligned variant of the *BTB* table above. The table operation is the same, except that both taken branch PCs and target addresses are block-aligned. This variant loses detail, but allows a greater amount of the instruction working set to be characterized with a given table size.

- **RetStack:** The return stack is a hardware structure in modern processors that predicts (very accurately) the target of return instructions corresponding to the current call stack. This table uses activity record fields to maintain a shadow copy of the return stack, and prefetches blocks of instructions near the top few frames on the return stack.

- **PCWindow:** As with *SPWindow* above, this also does not require an actual table; we simply use the PC of the first post-migrate instruction, and prefetch a window of instructions near that PC.

6.5.2 Summary generator

The *summary generator* activates when a core is signaled to migrate a thread. As introduced in Section 6.3, our baseline core design assumes some hardware support for thread swapping; at halt-time, the core collects and stores the register state of the thread being halted. (These techniques will still work absent hardware thread-swapping support; in such systems, it is important that our tables gather enough history to ensure we prefetch user state, not just recently accessed kernel state.)

While register state is being transferred, the summary generator reads through the data collected by the memory logger, and prepares a compact summary of the thread's likely future working set. This summary is transmitted after the architected thread state, and is used to prefetch the thread's working set when it resumes on the new core. During summarization, each table entry is inspected to determine its likely usefulness, by observing whether its age-adjusted hit counter exceeds a configurable threshold. Output data from the various tables are packed into cache-line size blocks for efficient transfer.

Summarizing the logged data serves several purposes: (1) Size reduction: summaries are considerably smaller than the total table storage in the memory logger. (2) Culling: at any given time, some table entries are unused, or contain relatively unimportant or redundant entries. Short summaries are critical to getting the migrated thread restarted quickly.

Selected table entries are summarized for transfer by generating a sequence of block addresses from each, and encoding them for transfer with a simple encoding. For contiguous or striding sequences, we use a simple linear-range encoding; for example, the summary of an entry in the stride table would be <*start_address, stride, length*>, which tells the prefetcher to fetch *length* cache blocks, starting at *start_address*, with stride *stride*. For arbitrary address sequences where this simple linear encoding is not appropriate, we resort to encoding a sequence as an initial cache block address followed by a sequence of narrow, signed delta-values.

The *length* parameter for strided accesses is necessary because we prefetch directly into the caches, as we are trying to fill them as quickly as possible. Therefore, we do not have the natural throttling effect present in, for example, stream buffers with dedicated storage (though the presence of finite MSHR resources limits overall prefetcher throughput, judicious *length* restrictions help individual summaries share those MSHRs). Overall, we have tuned lengths to roughly transfer enough data to cover the first 1,000 instructions, based on measurements of the average number of unique blocks touched over all benchmarks.

If we expect to migrate again in short order (e.g., speculative multithreading, transactional codes, etc., which repeatedly execute short threads), we can use the summary data to pre-fill the memory logger in the new core, since starting the new memory logger from scratch may not give it time to ramp up before the next migration.

6.5.3 Summary-driven prefetcher

Rounding out our working-set migration support hardware is the summary-driven prefetcher itself, depicted in Figure 6.4. When a previously-suspended thread is re-activated on a core, its summary records are read by the prefetcher.

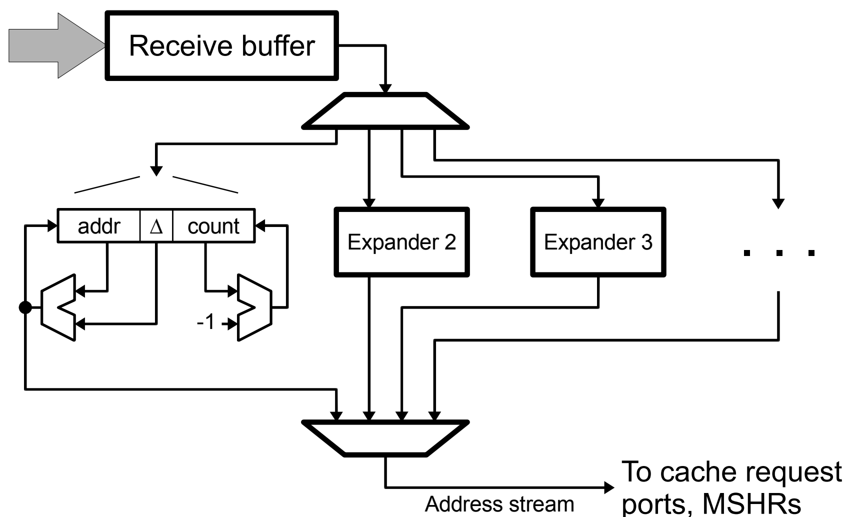


Figure 6.4: Summary-driven prefetcher.

Each record is expanded to a sequence of cache block addresses, which are submitted for prefetching as bandwidth allows. While the main execution pipeline reloads register values and resumes execution, the prefetcher independently begins to prefetch a likely working set. Prefetches search the entire memory hierarchy, and contend for the same resources as demand requests. Once the register state has been loaded, the thread's execution begins to compete with the prefetchers for memory bandwidth; this is modeled.

Unlike the memory logger, it may be more difficult to completely decouple the prefetch engine from the performance-sensitive parts of the processor. We model the prefetch engine as submitting memory requests as virtually-addressed requests at ports of the instruction and data caches, utilizing the existing memory hierarchy (including MSHRs) as much as possible. While these requests compete with the thread itself for cache ports, we expect most prefetching to occur while the thread would otherwise be stalled for memory access. This design does not require additional ports.

6.6 Methodology

We evaluate the prefetching coverage and overall performance of our working-set migration system using SMTSIM [Tul96], an out-of-order processor and memory system simulator introduced in Chapter 2. In this section, we present our processor configuration, our parallel workload composition, and our performance metric.

6.6.1 Simulator configuration

We configure SMTSIM for multi-core simulation with single-threaded cores (SMT is not used in this chapter). The system consists of a four-core multiprocessor, with cores clocked at 2.0 GHz; timing for other structures is also reported in terms of this clock rate. Table 6.2 lists the most significant parameters of the baseline system used in our experiments. Our memory subsystem layout and modeled latencies are based on that of recent Intel Nehalem-based processors [Int08]; bandwidth constraints are based on benchmarking of Nehalem-based systems along with the specifications of DDR3-1600, to the extent we could adapt our simulator to model them.

Atop this baseline, we implement and evaluate the working-set migration architecture described in Section 6.5. All memory logger CAMs are 32 entries, and we use an easily-implemented replacement policy that is round-robin, but skips entries accessed recently (within the last 500 lookups).

6.6.2 Workloads

Evaluating the speed of migration is not straightforward. We could examine a particular environment which benefits from fast migration (e.g. speculative multithreading, the shared-thread multiprocessor), but the results would be very specific to those execution models; instead, we strive to model a more generic environment and produce techniques that are useful in the most general case.

We start with all individual benchmarks from the SPEC2000 suite, running standalone on a four-core processor. Each benchmark is simulated for 200 million

Table 6.2: Baseline processor parameters.

Parameter	*Value*
Clock rate	2.0 GHz
Fetch width	4
Reorder buffer	128 entries
Integer window	64 insts
FP window	64 insts
Max issue width	4
Integer ALUs	4
FP ALUs	2
Load/store units	2
Branch predictor	4 Kib gshare
BTB	256-entry, 4-way
Cache block size	32 B
Page size	8 KiB
L1 I-cache size/assoc.	32 KiB/4-way
L1 I-MSHR	16 entries, 32 waiters each
L1 D-cache size/assoc.	32 KiB/4-way
L1 D-cache ports	2 read/write
L1 D-MSHR	16 entries, 32 waiters each
L2 cache size/assoc.	512 KiB/8-way, per-core
L2 cache ports	8 banks, 1 port each
ITLB entries	48
DTLB entries	128
Load-use latency, L1 hit	2 cycles
Load-use latency, L2 hit	14 cycles
Load-use latency, memory	176 cycles
Load-use latency, cross-core	34 cycles
TLB miss penalty	160 cycles
L1 I-cache ideal b/w	60 GiB/s
L1 D-cache ideal b/w	60 GiB/s
L2 cache ideal b/w	60 GiB/s
Bus ideal b/w	30 GiB/s
Mem ideal b/w	30 GiB/s R, 20 GiB/s W
Thread activate latency	15 cycles, to first fetch
Thread deactivate latency	44 cycles, to fetch available
L1 D-stream buffer	8 streams, 4 blocks/stream
L1 D-stream stride table	256-entry, 4-way

commits during their main phase of execution. To evaluate performance subsequent to migration, we force threads to migrate round-robin around cores, triggering a migration every 1 million commits. We consider – as shown previously in Figure 6.2 – the time to commit 10^n instructions after each migration, $n \in \{0 \ldots 6\}$. Examining performance across this wide range of intervals offers insight into how long it takes to amortize the cost of a migration, and the expected throughput for short, medium, and long-lived threads.

In these experiments, we are artificially degrading performance by forcing threads to move. We do this in order to capture performance degradation characteristics after migrations at arbitrary points in execution, independent of the specific reasons for migration. While there are many potential reasons for a particular migration (as discussed in Section 6.1), and a given migration policy may or may not prove beneficial in the long run, *any* migration in a real system will be subject to the overheads we characterize in this work.

We run experiments with a single thread moving among cores, assuming that caches are empty when the thread returns to a given core. This is a simplified model; a more realistic environment would have unrelated background threads occupying other cores, evicting the blocks of the thread under study during its absence. This simplification allows for evaluation free of noise from other threads.

We also run experiments where we actually simulate background threads; we evaluate these more realistic background-thread scenarios in Sections 6.7.6 and 6.7.7. For those cases, we use random sets of the other benchmarks running on the idle cores, with threads on the other cores migrating (between the cores on our simulated system) about as often as the thread under measurement.

6.6.3 Metrics

As discussed in Chapter 2, evaluating the performance of migration and short-thread acceleration support presents its own challenges: typical long-term performance metrics understate the degree to which performance suffers immediately after migrations, and also provide no insight as to how long performance takes to recover after migration operations. We introduce a new metric, *interval*

IPC, which we report in terms of *post-migrate speedup* or *post-migrate slowdown*, as appropriate. See Section 2.2.4 for a discussion of interval IPC.

6.7 Analysis and Results

This section examines a number of mechanisms for predicting the future cache working set of a thread migrating between cores. These mechanisms vary from simple to complex, ideal to realistic.

6.7.1 Bulk cache transfer

The most straightforward predictor of the future working set is the existing contents of the caches. We can copy those contents in bulk immediately after moving register state. We already saw in Figure 6.2 that this was not very effective, at least over the short intervals, due to the quantity of data transferred and the amount of data that does not turn out to be useful. (Prior work [LFF01] has shown, for first-level data caches in particular, that a substantial portion of resident blocks at any moment will not be referenced before being replaced; replacement policy research [QJP+07] has shown, for second-level data caches, that many requested blocks are evicted without being re-used.)

To perform bulk cache transfers at migration time – while the source core's pipeline is retrieving register values for transfer – we read out the set of cache tags belonging to the subject thread, and pack them into a message for transfer to the target core. We assume simple delta-encoding of cache block addresses, consuming 16 bits per tag on average. Prefetching begins as soon as the message arrives, using the existing level-1 memory request ports.

Figure 6.5 shows the impact of adding bulk cache transfers to our single-threaded workloads, with results broken down by individual cache. Over the shorter intervals, performance is significantly worse, as the dependence-free prefetch traffic consumes most available request bandwidth. Without any cache transfers (the baseline), the thread is already hamstrung by the low rate at which it can generate new memory references; adding bulk cache requests increases the

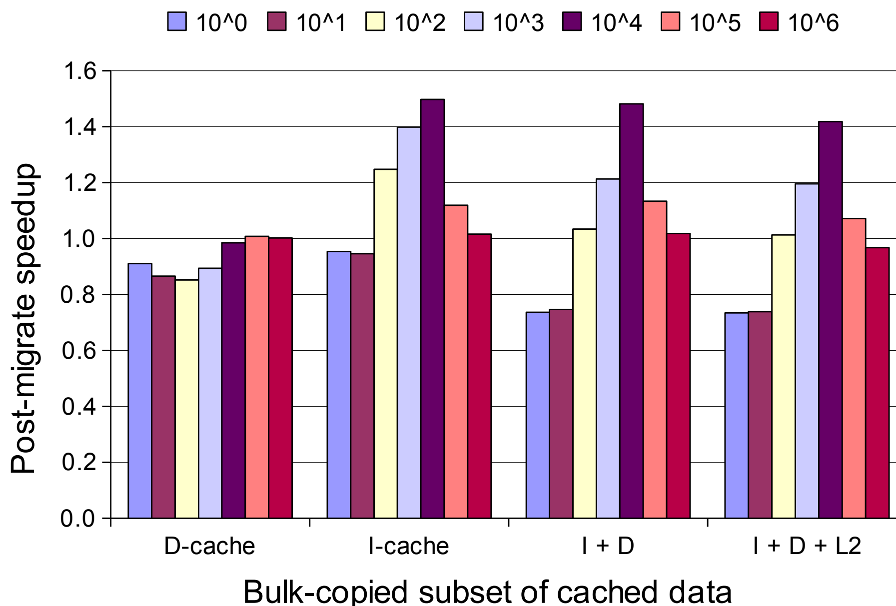


Figure 6.5: Impact of adding bulk cache transfers to single-threaded workloads. Speedups are relative to no migration support.

contention for MSHRs, request ports, and cache replacement priority, worsening the short-term situation. Over longer intervals, we do see benefit from bulk-transferring the L1 instruction cache, and from transferring both L1 caches together.

Moving the data cache by itself has almost no positive effect. This is a recurring theme in our results: the instruction cache is far more critical to post-migration performance than the data cache. Instruction cache accesses take place completely serially in cold cache mode. Data demand misses exhibit some parallelism, and are easily serviced in parallel with the instruction stream misses. This makes the instruction stream the clear bottleneck in this case. In fact, what we often find is that if we are getting many I cache misses, any attempt to prefetch the data stream just gets in the way. The case where all three caches are bulk copied again sees performance gains at large intervals primarily because the I cache is included, and in fact is less effective than the I cache alone. Due to the size of the L2 cache, the transfer cost is not amortized, even over 1 million commits: transferring I + D + L2 is never better than transferring just I + D.

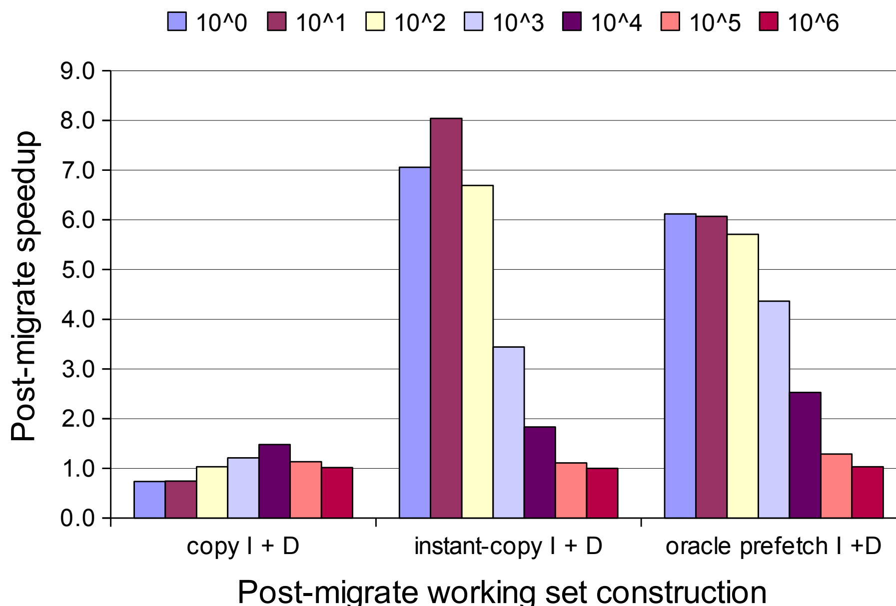


Figure 6.6: Impact of a future-oracle prefetcher, compared to instant (free) transfers and bulk cache copying.

6.7.2 Limits of prefetching

Our next experiment attempts to explore the potential of post-migration fetching support, over blind transfer of cache data. Here, we construct an oracle prefetcher that knows what L1 cache blocks will be touched in the future. It prefetches those in order, looking far enough ahead to fill each of the caches halfway. This result is shown in Figure 6.6. We see that the potential gains are high. Despite incurring the full cost of sending the summaries and transferring the data, the oracle's perfect accuracy allows it to approach the performance of free transfers – actually doing better at 10^3 commits – while far outpacing cache copying. Cache copying suffers from transferring too much data, and doing so with no particular order, which is unlikely to correspond to the access patterns immediately after migration.

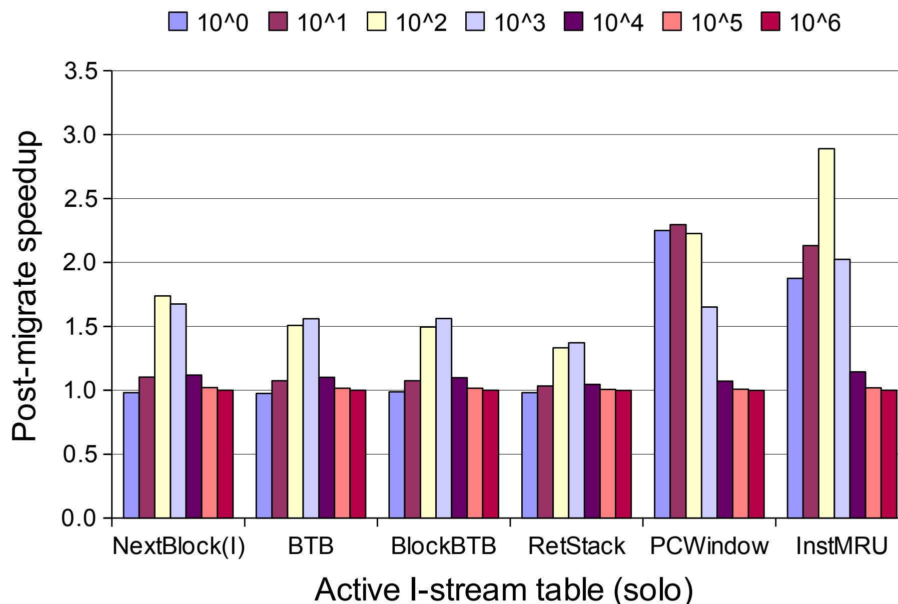


Figure 6.7: Impact of various instruction stream prefetchers, combined with an oracle data stream prefetcher.

6.7.3 I-stream prefetching

I-stream prefetching and D-stream prefetching are of course highly synergistic; we saw this earlier in Figure 6.5, where bulk-copying the D-cache was useless if the I-stream was left unassisted. Therefore, to evaluate individual tables of our memory logger that address the instruction cache, we need to assume a good solution for the data stream. In this section, we evaluate the different instruction stream loggers in the presence of an oracle data stream prefetcher. The oracle still incurs overhead, but has perfect knowledge of the 1000 commits following each migration.

We see these results in Figure 6.7. We cannot conclude too much yet from the absolute speedup, but what we do see is that two very simple approaches are quite effective over both the short and medium term. *PCWindow* simply uses the PC and fetches a window of instructions around it. *InstMRU* simply records the most recent instruction accesses. The latter has a significant advantage over moving the entire cache contents because it can be more timely: being smaller than the cache, it moves those instructions with the highest locality, more quickly.

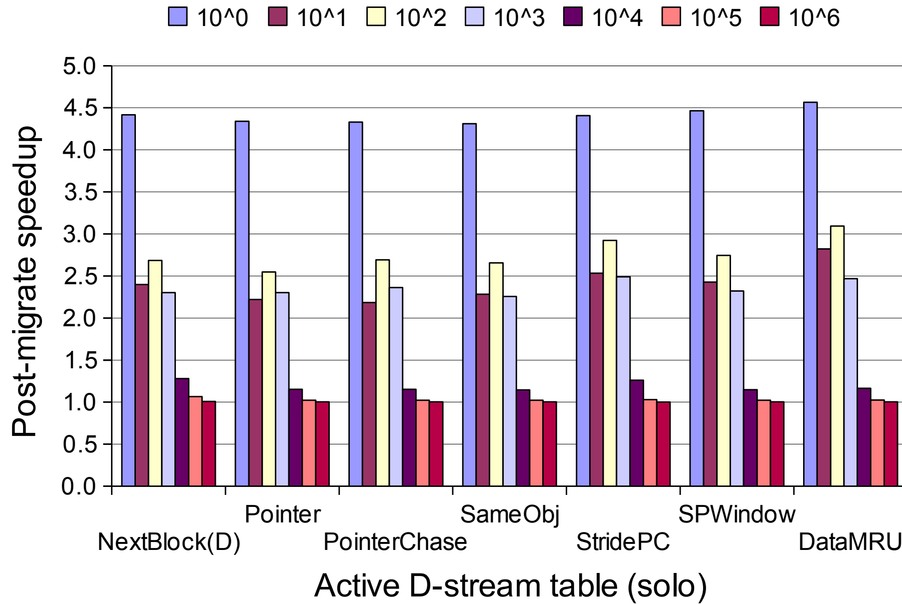


Figure 6.8: Impact of various data stream prefetchers, combined with an oracle instruction stream prefetcher.

6.7.4 D-stream prefetching

As we successfully prefetch the I-stream into the new caches, the D-stream then becomes the bottleneck. So again, to evaluate individual tables of our memory logger that address the data cache, we need to assume a good solution for the instruction stream. This section, then, evaluates the different data stream loggers in the presence of an oracle instruction stream prefetcher, where the oracle has perfect knowledge of I-cache behavior over 1000 commits following each migration.

These results comprise Figure 6.8. Here, the variations are lower than in the instruction stream case, partially because of the diversity of access patterns, but also because of overlap. Several of these tables track the same accesses in different ways. Again, though, we see that we can get away with very simple tables. *DataMRU* and *StridePC* are both good over various intervals.

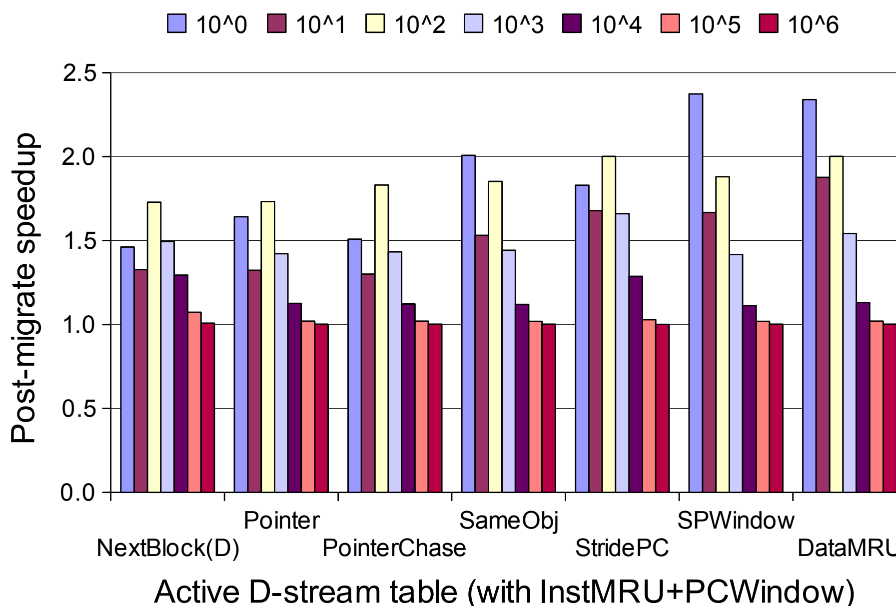


Figure 6.9: Impact of various combinations of realistic instruction and data stream prefetchers.

6.7.5 Combined prefetchers

Next, we examine several combinations of instruction and data stream prefetchers. These represent completely realistic prefetch scenarios with no oracle knowledge. For the instruction stream, we use a combination of PCWindow and InstMRU. Results with just InstMRU were quite similar, because InstMRU somewhat subsumes PCWindow (the current PC is always in the MRU table). However, by combining them (at no cost: the PC is available), we fetch a larger window of instructions around the current PC than we do for the other addresses.

We combine these with several of our data stream predictors, and present the results in Figure 6.9. We see that, taken individually, the “StridePC” and “DataMRU” data-stream predictors give excellent performance, at both 100 and 1000 instructions. For threads as short as 100 instructions, then, we can achieve speedups as high as 2X using this working set prediction framework.

Table 6.3 shows the transfer intensity and the accuracy of several prefetching schemes. Those include the best two prefetch combinations from the previous figure, as well as bulk prefetch of the L1 caches. We see from these results that

Table 6.3: Prefetcher activity and accuracy, mean over 200 migrations.

Prefetcher	*Blocks/migrate*	*Accuracy*
InstMRU+PCWindow+StridePC	212	64.00%
InstMRU+PCWindow+DataMRU	110	59.44%
Bulk transfer I+D	1195	48.63%

access stream monitoring-based tables provide both more accurate and much more highly directed prefetching than moving the cache state itself.

6.7.6 Allowing previous-instance cache re-use

Thus far, we have been migrating individual threads in an otherwise idle system, in order to evaluate prefetching absent interference from other threads. However, that is not the primary scenario we are targeting; with no other threads executing, a frequently-migrating thread would quickly build up copies of its working set on each core, and gain much less from prefetching. Therefore, to simulate cache interference from other threads, but still maintain a relatively noise-free simulation scenario, we have assumed thus far that when a thread is migrated to a core, it may not re-use data already present on that core from a previous instance — assuming its entire working set has been replaced by other execution.

For a less contrived scenario, in this section we remove the restriction on cached data re-use. In order to provide cache replacement pressure on the cores which are not running the thread under evaluation, we schedule an independent workload (chosen randomly from SPEC2000, as described in the next section) on each, and move these background threads among cores periodically, outside of our prefetch-evaluation time periods. Figure 6.10 shows the resulting performance. Compared to Figure 6.9, our gains are reduced, but overall performance trends are similar and remain valid. For example, mean suite-wide speedup for the "DataMRU" combination over 100 post-migrate commits has dropped from about 2.00 to 1.61. This drop is a combination of the additional cache re-use, along with contention for interconnect and memory resources. However, the available performance gains are still quite high.

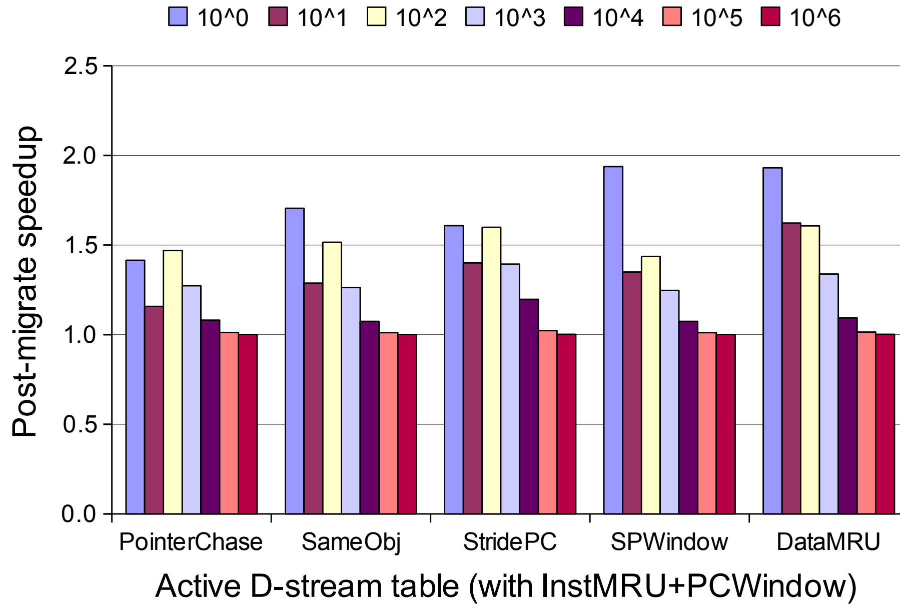


Figure 6.10: Realistic prefetchers, with previous-instance cache reuse and background thread movement.

6.7.7 Impact on other threads

While we have focused on the benefits of migrate-time prefetching on migrating threads in isolation, it is likely that a four-core system such as ours would have threads scheduled on some other cores, competing for interconnect and memory bandwidth. To evaluate the cost of our prefetching scheme on these bystander threads, we again added four additional non-migrating threads to the system: one waiting to execute on whichever core our "nomad" thread is using at any given time, and three others executing on the three remaining cores. (The background threads are chosen at random from the SPEC2000 suite such that, across all experiments, each individual benchmark is equally represented.)

We find that these background threads suffer negligible overall performance degradation from the addition of prefetching. This system does not prefetch continuously, but rather issues bursts of prefetches targeted specifically in response to thread migrations. Focusing on the short post-migrate time windows when the prefetcher is actually active, background threads will indeed slow down due to interference, but the impact is small and short-lived. For a specific example, starting

with the “StridePC” experiment described in Section 6.7.5, and adding stationary background threads to the system, we measure a mean background thread speedup of 0.961 over the time it takes the nomad thread to make 100 post-migrate commits; as the initial burst of prefetch activity tapers off, bystander performance recovers rapidly, to a mean of 0.999 speedup by 10000 post-migrate nomad commits.

6.7.8 Adding a shared last-level cache

To demonstrate that our gains are largely insensitive to underlying main-memory latency, we model the addition of a shared L3 cache. We add an 8MiB 16-way associative shared L3 cache, shared among the cores, with an overall load-use latency of 40 cycles (L2 latency remains 14 cycles). With this L3 added to both the baseline and experimental cases, we find that performance is nearly identical to the results already shown. Returning again to the example of the “StridePC” experiment from Section 6.7.5, we find that adding the L3 decreases nomad speedup by a mean of 0.1% for 100 post-migrate commits, and by a mean of 0.10% across all time scales.

When a thread is migrated, our prefetcher serves to mitigate the impact of cache misses at its new core. Intuitively, many of these misses and prefetches will be serviced with core-to-core transfers from the prior core, – without requiring access to memory – with latency that is competitive with shared-L3 access. While adding a shared L3 decreases the cost of retrieving those blocks unavailable from the prior core, it does not address the problem of poor memory parallelism due to serial demand misses.

6.7.9 Simple hardware prefetchers

While we present a framework which allows for evaluation of several schemes in concert – we describe a number of capture tables which vary in complexity – we achieved our best results with a small combination of simple tables, e.g. InstMRU + PCWindow + DataMRU.

To contrast with a conventional hardware prefetcher, for this section we add

next-block prefetchers to both L1 caches: these prefetch the successor to each block, at the first touch after that block is filled. Using these prefetchers instead of our proposed migration-targeting prefetchers, we observed only 1.010 mean speedup over 100 post-migrate commits. Over longer time scales, the benefits ramp up, to e.g. 1.104 mean speedup over 10000 post-migrate commits.

We find that over the short time scales we are most interested in for migration support, these traditional prefetchers are unable to contribute: they fall prey to the nascent thread's slow progress, which prevents them being trained quickly enough to help. However, since our proposed system issues targeted prefetches only in response to specific migrations, and these conventional prefetchers operate continuously, the two approaches complement each other.

6.8 Summary

As we proceed further into the multi-core era, migrations – scenarios where the state of a thread on one core needs to migrate to another core – will occur more often. Beyond the straightforward benefit of accelerating support for an operation we expect to become more frequent, speeding up migration will make several new execution models more applicable and effective: since this support decreases the cost of creating threads, it leaves each thread with less overhead to amortize away, thereby allowing profitable operation at finer granularity. This expands the horizon for finding and exploiting pockets of parallelism.

In this chapter we have described a working set predictor and prefetcher which greatly speeds up post-migration execution, as much as doubling the performance for short-lived threads. These solutions require small, simple tables to monitor the access stream of running threads, and a minimal address-generation engine to issue prefetches for migrating (or newly-forked) threads.

We have shown that I-stream delivery is most critical to post-migrate performance; without assistance, the fetch unit is left to fill the I-cache with serial demand-misses. However, we have improved the delivery of both the instruction *and* data streams, to boost the performance of short threads. We have also shown

that simply copying cache contents is extremely ineffective over the short term: it moves too much data, at too much expense, and much of that data is not useful over the short term. We have demonstrated techniques that as much as double the performance for short threads. These solutions required only small, simple tables to monitor the access streams of a running thread on each core.

Acknowledgements

This chapter contains material from "Fast Thread Migration via Cache Working Set Prediction", by Jeffery A. Brown and Dean M. Tullsen, which has been submitted for possible publication by the Association for Computing Machinery in *Proceedings of the Nineteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Conclusion

Exploitation of parallelism has for decades been central to the pursuit of computing performance. This is evident in processor design at many levels: in the pipelining of execution stages, in superscalar dispatch among replicated functional units, in pipelining and banking of memory subsystems, and more recently, in the proliferation of self-contained processing cores within multi-core processors. The shift to multi-core designs is a profound one, since – due to the loose coupling among cores – available hardware parallelism promises to scale much farther than at prior levels, limited by interconnect degree and thermal constraints. This explosion in parallelism necessitates changes in how our hardware and software interact.

In this dissertation we have focused on hardware aspects of this interaction, in order to provide support for efficient on-chip parallel execution in the face of increasing core counts. We have introduced a mechanism for coping with increasing memory latencies in multithreaded processors, improved the coherence subsystem for a chip-multiprocessor landscape, introduced efficient hardware primitives for thread migration and scheduling, and demonstrated an effective system for predicting near-term working sets, thereby enabling efficient thread creation and migration. Each of these facets represents an important concern for parallel execution in future designs; shortcomings in any of these complementary areas will be a roadblock to continued performance scaling.

7.1 Memory Latency in Multithreaded Processors

In Chapter 3, we demonstrated how long-latency memory operations can overwhelm a simultaneous-multithreaded processor core's capacity to schedule around instructions that are dependent on memory operations. Despite SMT's ability to share resources among independent threads on a cycle-by-cycle basis, we saw that a miss-intensive thread can substantially degrade the throughput of a co-scheduled thread, by up to a factor of ten.

The underlying reason for the observed degradation is that aggressive resource management policies, which serve SMT execution well under ordinary execution conditions, fall victim to resource hoarding: instructions dependent on long-latency misses occupy precious execution resources, but are unable to make forward progress and free those resources in a timely manner. As further dependent instructions are fetched into the pipeline and added to the instruction queues, fewer and fewer scheduling resources remain available for co-scheduled threads. At the heart of the problem is the inability of the baseline processor to either revoke resources allocated to in-flight correct-path instructions, or to predict these pathological cases in advance and withhold resources.

We introduced a reactive solution which effectively detects these instances of pathological resource hoarding as they begin to develop, and responds by flushing instructions in order to free hoarded resources for use by other threads. This mechanism avoids affecting the performance of well-behaved computation, does not need training for any particular workload, does not necessitate additional storage, and does not depend on any particular programming idioms for detection. In our evaluation, this technique more than doubled two-thread throughput, and cut average workload response time by about a factor of two.

7.2 Cache Coherence for CMPs

In Chapter 4, we considered the problem of implementing cache coherence across cores. Today's dominant multi-core programming models rely on shared memory: values written to an address by one core will be visible, subject to constraints, to subsequent reads of that address on other cores. The cache coherence system is responsible for coordinating the propagation of memory values between cores. In order to support shared-memory programming in the face of increasing core counts, we evaluated the use of directory-based coherence – a traditional approach to implementing coherence in large-scale multiprocessors – on an aggressive chip-level multiprocessor with a scalable 2-D mesh interconnect, directory caches, private L2 caches, and per-tile memory controllers.

We saw that a straightforward implementation of a traditional directory protocol on a multi-core architecture – while sufficient – failed to take advantage of the latency and bandwidth landscape typical of current and future chip multiprocessors. In particular, chip multiprocessors differ starkly in several ways from their predecessors:

- From the perspective of a node experiencing a cache miss for a given block of memory, the cost of contacting the directory controller at the node responsible for that block is often, in a chip multiprocessor, drastically lower than the subsequent cost of accessing the latter node's attached memory to retrieve a copy of that block. In a traditional processor, both costs are dominated by that of inter-node communication.
- From the perspective of the tile which hosts the memory controller for a given block of main memory, the caches of other nodes in a chip multiprocessor are much "closer" in terms of latency than main memory. In a traditional multiprocessor, this is reversed: the cost of communicating with other nodes tends to dominate that of consulting locally-attached main memory.
- From the perspective of a given node, the latency of communication with different nodes in a chip multiprocessor varies much more than in a traditional

multiprocessor; communicating with a distant core takes several times as long as with an immediate neighbor. In traditional multiprocessors, the latency of using the interconnect at all dominates the differences in per-node latencies.

We proposed a multi-core specific customization of directory-based coherence, that takes advantage of these differences by consulting additional cores in service of some cache misses, obviating some costly off-chip accesses. We further refined this to account for the travel distances of requests and larger, data-carrying replies. We demonstrated suite-wide speedup, reduction in average L2 miss service latency, and a decrease in interconnect utilization.

As we seek to use increasingly parallel chip multiprocessors more effectively with parallel workloads, overall performance will become increasingly sensitive to that of the coherence subsystem. As such, it is important to reconsider the design of coherence mechanisms, ensuring they take into account the unique characteristics of multi-core architectures.

7.3 Multithreading Among Cores

While the exploitation of parallelism at multiple levels remains necessary to achieve good performance, the most recent arena for the expansion of on-chip parallelism – the shift to multi-core processors – is in multiple respects a significant departure from prior techniques. For the purposes of this dissertation, the most important of the differences are:

- Scalability: compared to parallelism enhancement via increasing pipeline depth or issue width, increasing core counts promises to allow performance scaling over a much larger range of values. Rather than requiring drastic design changes to the processor, adding additional cores is primarily a process of replication followed up by interconnect and power scaling.

- Explicitness: unlike lower-level enhancements such as pipelining, the addition of cores is a change which must be visible to workloads in order for them to benefit from the additional hardware parallelism. Prior enhancements such

as instruction pipelining operate beneath the veil of the instruction-set architecture; to utilize additional cores, however, workloads must be presented as distinct threads for execution on distinct cores.

In order to reap the continued benefits promised by scalability, we have enhanced the underlying hardware to defray the additional costs imposed by the new explicitness requirement.

The problem of parallelizing computation, whether in terms of specific workloads or of abstract algorithms, is a well-studied area of computer science. Many approaches have been devised, ranging from purpose-designed inherently-parallel data flow languages, programming languages amenable to automatic parallelization, parallelizing compilers for serial languages, programmer-generated parallel mark-up directives atop serial languages, down through the software-hardware stack to unassisted hardware-level parallelization in the form of speculative multithreading: the construction of new threads which perform direct computation and prefetching, yet preserving the instruction-level semantics of the original program.

While approaches to workload parallelization are numerous and varied, if execution takes place on a conventional shared-memory chip multi-processor, all approaches are affected by the overheads associated with thread creation and scheduling operations. Since the cost of these thread-management primitives necessarily overshadow any possible gains from workload parallelization – coarsening the minimum useful computation grain size, and decreasing the frequency with which scheduling decisions can be considered – we have introduced and evaluated several techniques to decrease these overheads. We have attacked along two fronts: first, considering the movement of explicit thread state such as register values, and then widening our scope to include memory working sets.

7.3.1 Registers: Thread Migration & Scheduling

In Chapter 5, we extended a conventional chip multi-processor model with hardware support for transferring architected thread state (i.e., register values) in and out of the execution cores. These transfers have very low latency – tens of cycles – and overlap with execution in order to minimize disruption to pipeline

operation. Coupled with shared, off-core storage for this thread state, we call this model the "Shared Thread Multiprocessor".

With this new hardware facility we have enabled cheap, "multithreading-style" thread interaction between the cores of a chip multi-processor. In our experimental evaluation, we have demonstrated two immediate gains from this new ability. First, our new mechanism is fast enough that the latency of a thread migration is often less than that of an access to off-chip memory; we took advantage of this by moving threads to take advantage of compute resources left temporarily idle in the shadow of stalls for main memory access. Second, we took advantage of the lowered cost of thread scheduling primitives to profitably operate a more conventional multi-thread workload scheduler at a higher sample rate than would be possible at time intervals typical of operating system schedulers.

7.3.2 Memory: Working Set Prediction & Migration

In Chapter 6, we continued to pursue high-performance inter-core threading support, expanding our focus to include the larger problem of managing a thread's instruction and data working sets. We started with the observation that, even given efficient hardware support for register-set transfers, newly-migrated threads tend to perform very poorly for quite some time after each migration. Even though register sets and control-flow were being efficiently transferred within tens of cycles, commit throughput still dropped precipitously after control transfer, bogged down by sequences of serial cache misses.

After a migration, the target execution pipeline, as well as the caches and chip-wide interconnect, were being left severely under-utilized due to the lack of cached instruction and data working sets. The fundamental problem was that the only way a thread could accumulate a useful working set is by executing instructions, which generates demand references for the instruction and data blocks that comprise its working set. Without a cached working set present, that rate of execution is very low – due to cache misses – which in turn decreases the rate at which additional blocks of the working set are requested. These vicious cycles of poor performance were observed to persist through tens to hundreds of thousands

of instructions.

To address this, we introduced a table-driven system which passively observes threads as they execute, and prepares working-set summaries used to speed migration. At a target core, a simple prefetcher uses these summaries to rapidly prefetch useful instruction and data working sets subject only to the throughput limitations of the underlying caches and interconnect, thereby breaking the cycle of poor performance.

After evaluating summarizers of varying complexity, we demonstrated that a practical combination of a handful of the simplest tables as much as doubled the performance of newly-migrated threads. We observed that the instruction working set was most critical for performance, due to the inability of the pipeline to progress beyond instruction-miss stalls. We also showed that a straightforward solution – copying cache contents outright – was extremely ineffective.

7.4 Final Remarks

Modern processors are not only replete with opportunities for parallel execution, they depend on it to achieve good performance. As core counts increase, thread-level parallelism is rising in prominence as a means by which system-wide performance can continue to grow; efficient hardware support for multi-threaded execution is critical. In this dissertation, we have identified several impediments to the performance of parallel execution – in inter-thread scheduling, in memory coherence implementation, and in support for forking or migrating threads – and we have presented solutions which improve performance in each of these areas.

Bibliography

[ACC+90] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton J. Smith. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, pages 1–6, June 1990.

[AGGD02] Manuel E. Acacio, José González, José M. García, and José Duato. A novel approach to reduce L2 miss latency in shared-memory multiprocessors. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 25, April 2002.

[AGS05] Murali Annavaram, Edward Grochowski, and John Shen. Mitigating Amdahl's Law through EPI throttling. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 298–309, June 2005.

[AKK+93] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[AN90] Arvind and Rishiyur S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3):300–318, March 1990.

[APD01] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 52–61, July 2001.

[Arv81] Arvind. Data flow languages and architecture. In *Proceedings of the 8th International Symposium on Computer Architecture*, page 1, May 1981.

[BGM+00] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert

Stets, and Ben Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 282–293, June 2000.

[BKT07] Jeffery A. Brown, Rakesh Kumar, and Dean M. Tullsen. Proximity-aware directory-based coherence for multi-core processor architectures. In *Proceedings of the 19th ACM Symposium on Parallel Algorithms and Architectures*, pages 126–134, June 2007.

[BT08] Jeffery A. Brown and Dean M. Tullsen. The shared-thread multiprocessor. In *Proceedings of the 21st International Conference on Supercomputing*, pages 73–82, June 2008.

[BWC+02] Jeffery A. Brown, Hong Wang, George Chrysos, Perry H. Wang, and John P. Shen. Speculative precomputation on chip multiprocessors. In *Proceedings of the 6th Workshop on Multithreaded Execution, Architecture and Compilation*, pages 35–42, November 2002.

[CB95] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high performance processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

[CCM+06] JaeWoong Chung, Hassan Chafi, Chi Cao Minh, Austen McDonald, Brian D. Carlstrom, Christos Kozyrakis, and Kunle Olukotun. The common case transactional behavior of multithreaded programs. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 266–277, February 2006.

[CFS+04] Joachim Clabes, Joshua Friedrich, Mark Sweet, Jack DiLullo, Sam Chu, Donald Plass, James Dawson, Paul Muench, Larry Powell, Michael Floyd, Balaram Sinharoy, Mike Lee, Michael Goulet, James Wagoner, Nicole Schwartz, Steve Runyon, Gary Gorman, Phillip Restle, Ronald Kalla, Joseph McGill, and Steve Dodson. Design and implementation of the POWER5 microprocessor. In *IEEE International Solid-State Circuits Conference*, pages 55–57, February 2004.

[CGG04] Pedro Chaparro, José González, and Antonio González. Thermal-aware clustered microarchitectures. In *Proceedings of the 22nd IEEE International Conference on Computer Design*, pages 48–53, October 2004.

[Com00] Compaq Computer Corporation, Shrewsbury, MA. *Alpha 21264 Microprocessor Hardware Reference Manual*, February 2000.

[CPT08] Bumyong Choi, Leo Porter, and Dean M. Tullsen. Accurate branch prediction for short threads. In *Proceedings of the 13th International*

Conference on Architecture Support for Programming Languages and Operating Systems, pages 125–134, March 2008.

[CRVF04a] Francisco J. Cazorla, Alex Ramírez, Mateo Valero, and Enrique Fernández. Dynamically controlled resource allocation in SMT processors. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 171–182, December 2004.

[CRVF04b] Francisco J. Cazorla, Alex Ramírez, Mateo Valero, and Enrique Fernández. Optimising long-latency-load-aware fetch policies for SMT processors. *International Journal of High Performance Computing and Networking*, 2(1):45–54, 2004.

[CS06] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 264–276, June 2006.

[CSCT02] Jamison D. Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. Pointer cache assisted prefetching. In *Proceedings of the 35th International Symposium on Microarchitecture*, pages 62–73, November 2002.

[CSK+99] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous subordinate microthreading (SSMT). In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 186–195, May 1999.

[CSM+05] Theofanis Constantinou, Yiannakis Sazeides, Pierre Michaud, Damien Fetis, and Andre Seznec. Performance implications of single thread migration on a chip multi-core. *ACM SIGARCH Computer Architecture News*, 33(4):80–91, November 2005.

[CSO+00] Yu Cao, Takashi Sato, Michael Orshansky, Dennis Sylvester, and Chenming Hu. New paradigm of predictive MOSFET and interconnect modeling for early circuit simulation. In *Proceedings of the 2000 Custom Integrated Circuits Conference*, pages 201–204, May 2000.

[CWS06] Koushik Chakraborty, Philip M. Wells, and Gurindar S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the 12th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 283–292, October 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 *Proceedings of the 28th International Symposium on Computer Architecture*, pages 14–25, July 2001.

[DT99] Fredrik Dahlgren and Josep Torrellas. Cache-only memory architectures. *Computer*, 32(6):72–79, June 1999.

[EE07] Stijn Eyerman and Lieven Eeckhout. A memory-level parallelism aware fetch policy for SMT processors. In *Proceedings of the 13th International Symposium on High-Performance Computer Architecture*, pages 240–249, February 2007.

[EE09] Stijn Eyerman and Lieven Eeckhout. Per-thread cycle accounting in SMT processors. In *Proceedings of the 14th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 133–144, March 2009.

[EMA03] Ali El-Moursy and David H. Albonesi. Front-end policies for improved issue efficiency in SMT processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 31–40, February 2003.

[EPS06] Noel Eisley, Li-Shiuan Peh, and Li Shang. In-network cache coherence. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 321–332, December 2006.

[FMJ+07] Joshua Friedrich, Bradley McCredie, Norman James, Bill Huott, Brian Curran, Eric Fluhr, Gaurav Mittal, Eddit Chan, Yuen Chan, Donald Plass, Sam Chu, Hung Le, Leo Clark, John Ripley, Scott Taylor, Jack Dilullo, and Mary Lanzerotti. Design of the POWER6 microprocessor. In *Proceedings of the 2007 IEEE International Solid-State Circuits Conference*, pages 96–97, February 2007.

[GAD+06] Michael Golden, Srikanth Arekapudi, Greg Dabney, Mike Haertel, Stephen Hale, Lowell Herlinger, Yongg Kim, Kevim McGrath, Vasant Palisetti, and Monica Singh. A 2.6GHz dual-core 64bx86 microprocessor with DDR2 memory support. In *Proceedings of the 2006 IEEE International Solid-State Circuits Conference*, pages 325–332, February 2006.

[GWM90] Anoop Gupta, Wolf-Dietrich Weber, and Todd C. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *Proceedings of the 1990 International Conference on Parallel Processing, Volume 1*, pages 312–321, August 1990.

[HDH+10] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabrice Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann,

Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob Van Der Wijngaart, and Timothy Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the 2010 IEEE International Solid-State Circuits Conference*, pages 19–21, February 2010.

[HKN+92] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 136–145, May 1992.

[HKS+05] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *Proceedings of the 19th International Conference on Supercomputing*, pages 31–40, June 2005.

[HMH01] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, April 2001.

[HP04] A. Hartstein and Thomas R. Puzak. The optimum pipeline depth considering both power and performance. *ACM Transactions on Architecture and Code Optimization*, 1(4):369–388, December 2004.

[HS98] Sébastien Hily and André Seznec. Standard memory hierarchy does not fit simultaneous multithreading. In *Proceedings of the 2nd Workshop on Multithreaded Execution, Architecture and Compilation*, January 1998.

[HWO98] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 58–69, October 1998.

[Int08] Intel. First the tick, now the tock: Next generation Intel microarchitecture (Nehalem). *Intel white paper*, 2008.

[JJ92] O'Shea Jackson and Mark Jordan. It was a good day. *Fabrication Review*, 3(7), November 1992.

[JN07] Tim Johnson and Umesh Nawathe. An 8-core, 64-thread, 64-bit power efficient SPARC SoC (Niagara2). In *Proceedings of the 2007 International Symposium on Physical Design*, page 2, March 2007.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In

Proceedings of the 17th International Symposium on Computer Architecture, pages 364–373, June 1990.

[KAO05] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE MICRO*, 25(2):21–29, March 2005.

[KEW+85] Randy H. Katz, Susan J. Eggers, David A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a cache consistency protocol. In *Proceedings of the 12th International Symposium on Computer Architecture*, pages 276–283, June 1985.

[KFJ+03] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 81–92, December 2003.

[KM03] David Koufaty and Deborah T. Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.

[KST04] Ron Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, March 2004.

[KST10] Md Kamruzzaman, Steven Swanson, and Dean M. Tullsen. Software data spreading: Leveraging distributed caches to improve single thread performance. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, June 2010.

[KT98] Venkata Krishnan and Josep Torrellas. An direct-execution framework for fast and accurate simulation of superscalar processors. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, October 1998.

[KTR+04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 64–75, June 2004.

[KZT05] Rakesh Kumar, Victor Zyuban, and Dean M. Tullsen. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 408–419, June 2005.

[LFF01] An-Chow Lai, Cem Fide, and Babak Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 144–154, July 2001.

[LGH94] James Laudon, Anoop Gupta, and Mark Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *Proceedings of the 6th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 308–318, October 1994.

[LL97] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 241–251, June 1997.

[LLG+92] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford DASH multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[Luk01] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 40–51, July 2001.

[MG02] Pedro Marcuello and Antonio González. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 55–64, February 2002.

[MHW03] Milo M. K. Martin, Mark D. Hill, and David A. Wood. Token coherence: decoupling performance and correctness. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 182–193, June 2003.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[MMB+08] Jeffrey C. Mogul, Jayaram Mudigonda, Nate Binkert, Partha Ranganathan, and Vanish Talwar. Using asymmetric single-ISA CMPs to save energy on operating systems. *IEEE Micro*, 28(3):26–41, May 2008.

[MMG+06] Avi Mendelson, Julius Mandelblat, Simcha Gochman, Anat Shemer, Rajshree Chabukswar, Erik Niemeyer, and Arun Kumar. CMP implementation in systems based on the Intel Core Duo processor. *Intel Technology Journal*, 10(2):99–108, May 2006.

[MN99] Maged M. Michael and Ashwini K. Nanda. Design and performance of directory caches for scalable shared memory multiprocessors. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 142–151, January 1999.

[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 *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 129–140, February 2003.

[NY98] Mario Nemirovsky and Wayne Yamamoto. Quantitative study of data caches on a multistreamed architecture. In *Proceedings of the 2nd Workshop on Multithreaded Execution, Architecture and Compilation*, January 1998.

[ON90] Brian W. O'Krafka and A. Richard Newton. An empirical evaluation of two memory-efficient directory methods. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 138–147, June 1990.

[PELL00] Sujay Parekh, Susan Eggers, Henry Levy, and Jack Lo. Thread-sensitive scheduling for SMT processors. Technical Report 2000-04-02, University of Washington, 2000.

[PK94] Subbarao Palacharla and Richard E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 24–33, April 1994.

[PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th International Symposium on Computer Architecture*, pages 348–354, June 1984.

[PRA97a] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. The impact of instruction-level parallelism on multiprocessor performance and simulation methodology. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 72–83, February 1997.

[PRA97b] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997.

[QJP+07] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely Jr., and Joel S. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 381–391, June 2007.

[QMS+05] Carlos García Quiñones, Carlos Madriles, F. Jesús Sánchez, Pedro Marcuello, Antonio Gonzáles, and Dean. M. Tullsen. Mitosis compiler: An infrastructure for speculative threading sed on pre-computation slices. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 269–279, June 2005.

[RMS98] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.

[SA05] Lawrence Spracklen and Santosh G. Abraham. Chip multithreading: Opportunities and challenges. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 248–252, February 2005.

[SBCvE90] Rafael H. Saavedra-Barrera, David E. Culler, and Thorsten von Eicken. Analysis of multithreaded architectures for parallel computing. In *Proceedings of the 2nd ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, July 1990.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 414–425, June 1995.

[SKET07] Kyriakos Stavrou, Costas Kyriacou, Paraskevas Evripidou, and Pedro Trancoso. Chip multiprocessor based on data-driven multithreading model. *International Journal of High Performance System Architecture*, 1(1):24–43, 2007.

[SM98] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, pages 2–13, January 1998.

[SMM+09] Richard D. Strong, Jayaram Mudigonda, Jeffrey C. Mogul, Nathan L. Binkert, and Dean M. Tullsen. Fast switching of threads between cores. *ACM SIGOPS Operating Systems Review*, 43(2):35–45, April 2009.

[SSC00] Timothy Sherwood, Suleyman Sair, and Brad Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd International Symposium on Microarchitecture*, pages 42–53, December 2000.

[SSC02] Suleyman Sair, Timothy Sherwood, and Brad Calder. Quantifying load stream behavior. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 197–208, February 2002.

[ST00] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proceedings of the 9th International Conference on Architecture Support for Programming Languages and Operating Systems*, pages 234–244, November 2000.

[SV87] Gurindar S. Sohi and Sriram Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 27–34, June 1987.

[TB01] Dean M. Tullsen and Jeffery A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th International Symposium on Microarchitecture*, pages 318–327, December 2001.

[TE93] Dean M. Tullsen and Susan J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 278–288, May 1993.

[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 *Proceedings of the 23rd International Symposium on Computer Architecture*, pages 191–202, May 1996.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 392–403, June 1995.

[TKTC04] Eric Tune, Rakesh Kumar, Dean M. Tullsen, and Brad Calder. Balanced multithreading: Increasing throughput via a low cost multithreading hierarchy. In *Proceedings of the 37th International Symposium on Microarchitecture*, pages 183–194, December 2004.

[TLEL99] Dean M. Tullsen, Jack L. Lo, Susan J. Eggers, and Henry M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 54–58, January 1999.

[TTG95] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24(2):139–151, February 1995.

[Tul96] Dean. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd International Computer Measurement Group Conference*, pages 819–828, December 1996.

[VHR+07] Sriram Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, Shailendra Jain, Sriram Venkataraman, Yatin Hoskote, and Nitin Borkar. An 80-tile 1.28TFLOPS network-on-chip in 65nm CMOS. In *Proceedings of the 2007 IEEE International Solid-State Circuits Conference*, pages 5–7, February 2007.

[YN95] Wayne Yamamoto and Mario Nemirovsky. Increasing superscalar performance through multistreaming. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, pages 49–58, June 1995.

[ZA05] Michael Zhang and Krste Asanović. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 336–345, June 2005.

[ZS00] Craig B. Zilles and Gurindar S. Sohi. Understanding the backward slices of performance degrading instructions. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 172–181, June 2000.

[ZS01] Craig B. Zilles and Gurindar S. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 2–13, June 2001.

[ZT97] Zheng Zhang and Josep Torrellas. Reducing remote conflict misses: NUMA with remote cache versus COMA. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 272–281, February 1997.