UNIVERSITY OF CALIFORNIA, SAN DIEGO

Efficient Use of Execution Resources in Multicore Processor Architectures

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

in

Computer Science (Computer Engineering)

by

Matthew David DeVuyst

Committee in charge:

Professor Dean Tullsen, Chair
Professor Clark Guest
Professor Chandra Krintz
Professor Sorin Lerner
Professor Steven Swanson

2011

The dissertation of Matthew David DeVuyst is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____

_____
Chair

University of California, San Diego

2011

DEDICATION

Soli Deo Gloria

EPIGRAPH

For from Him and through Him and to Him are all things.
To Him be the glory forever. Amen.

—Romans 11:36 (NASB version)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First, and foremost, I would like to thank my Lord and Savior, Jesus Christ. "Whatever you do, do your work heartily, as for the Lord rather than for men" [Colossians 3:23, NASB]. This dissertation, therefore, is dedicated to the Lord. Indeed, it is only possible because of Him; the Apostle Paul writes, "I can do all things through Him who strengthens me" [Philippians 4:13, NASB]. I know I can do nothing apart from God's grace.

To my amazing wife, Lisa. You have been a pillar of support through most of my time in grad school, a constant source of encouragement, comfort, and love. Thank you for putting up with the long days (and nights) of work and for keeping me sane.

To my family, especially my mom and dad. Thank you for all your words of encouragement over the years—whether I succeed or fail, I know you'll love me just the same. Throughout my time in grad school, it was comforting to know that I had a place to go if I needed it. To my in-laws, Gil and Patty, thank you for all of your support and encouragement—you have become like second parents to me.

I would like to thank my advisor, Dean Tullsen, for excellent guidance and great patience. Thank you for all your help in editing papers, helping set a research direction, and offering great ideas when I was stumped. Without your help, this dissertation would not be possible.

Thanks to all of the people at UCSD that I've had the privilege of working with. Rakesh, in my early years of grad school, you set a great example of hard work and dedication; it was something I aspired to throughout my graduate career. Jeff, I cannot thank you enough for all of your help over the years. Not only were you a great guy to go to for technical advice, but our chats were much-appreciated times of refreshment amid the long hours of work. To all my other lab mates, Leo, M.D., Vasileios, Hung-Wei, Rick, Subhra, and Jack, thank you for all of the great feedback that helped make this dissertation possible. I'll always look back with fondness on my time with you at UCSD.

Chapter 2 contains material from "Runtime Parallelization of Legacy Code on a Transactional Memory System", by Matthew DeVuyst, Dean M. Tullsen, and Seon Wook Kim, which appears in *Proceedings of the 6th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*. The dissertation author was the primary investigator and author of this paper. The material in Chapter 2 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 3 of this dissertation contains material from "Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors", by Matthew DeVuyst, Rakesh Kumar, and Dean M. Tullsen, which appears in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium 2006 (IPDPS)*. The dissertation author was the primary investigator and author of this paper. The material in Chapter 3 is copyright ©2006 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.

VITA AND PUBLICATIONS

2000-2002	Computer Technician Point Loma Nazarene University, IT Dept.
2001-2003	Internship Central Intelligence Agency
2003	Internship TrellisWare Technologies
2003	Bachelor of Arts in Computer Science Point Loma Nazarene University
2003-2010	Teaching Assistant University of California, San Diego
2004	Internship Teradata/NCR
2004-2009	Research Assistant University of California, San Diego
2006	Master of Science in Computer Science University of California, San Diego
2011	Doctor of Philosophy in Computer Science (Computer Engineering) University of California, San Diego

PUBLICATIONS

"Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors" Matthew DeVuyst, Rakesh Kumar, Dean M. Tullsen. *Proceedings of the IEEE International Parallel and Distributed Processing Symposium 2006 (IPDPS)*, April, 2006.

"Runtime Parallelization of Legacy Code on a Transactional Memory System" Matthew DeVuyst, Dean M. Tullsen, Seon Wook Kim. *Proceedings of the 6th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*, January 2011.

ABSTRACT OF THE DISSERTATION

Efficient Use of Execution Resources in Multicore Processor Architectures

by

Matthew David DeVuyst

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California, San Diego, 2011

Professor Dean Tullsen, Chair

As the microprocessor industry embraces multicore architectures, inherently parallel applications benefit directly as they easily transform into sets of homogeneous parallel threads. However, many applications do not fit this model. These applications include legacy binaries compiled for a single thread of execution and inherently serial applications. The inability of these two kinds of applications to exploit multicore architectures has created a crisis for the microprocessor industry: customers have come to expect significant performance improvements in all of their application every processor generation, but recent multicore architectures have failed to meet those expectations for many applications. This dissertation explores ways in which these applications can run efficiently on multicore platforms.

The performance of legacy binaries compiled for a single thread of execution can be improved through automatic parallelization. We introduce a new technique to automatically parallelize binaries as they are executing. The parallelization technique leverages the benefits of hardware transactional memory, a synchronization mechanism enabling optimistic concurrency. Our technique exploits this to parallelize code that a traditional parallelizing compiler would be unable to transform due to potential memory aliasing.

Applications with fundamentally serial code can benefit from core customization. The more heterogeneous the cores are, the more likely that a given application will find a core on which it runs efficiently. We investigate two forms of heterogeneity: that created on homogeneous hardware by unbalanced resource assignment, and heterogeneity created by hardware asymmetry. We first consider a homogeneous multicore system composed of multithreading cores. Often the best schedules on such a system are unbalanced. We propose a set of novel scheduling algorithms that consider unbalanced schedules to find good application-to-core assignments. We consider objective functions of both performance and energy. We also explore how applications can benefit from diverse ISAs by considering heterogeneous-ISA multicore systems. We propose a new technique to rapidly migrate a thread among cores of different ISAs, allowing applications to take advantage of hardware heterogeneity for performance gain or energy savings.

Chapter 1

Introduction

The last few years have seen a dramatic shift in the microarchitecture industry: from single-core to multicore architectures. Limitations on the power density of integrated circuits have forced a change in the paradigm of processor design. No longer do we look to instruction-level parallelism (ILP) as the primary source of increased performance potential. Instead, thread-level parallelism (TLP) has become the primary vehicle for better performance. Increasing transistor counts are being put to use to exploit thread-level parallelism, forcing computer engineers to rethink how to make the best use of the additional core resources.

Inherently parallel applications benefit directly from multicore architectures, as they easily transform into sets of homogeneous parallel threads. However, many applications cannot directly benefit from multicore architectures. These applications fall into two categories:

1. Legacy binaries compiled for a single thread of execution.

2. Inherently serial applications.

The former may include some inherent parallelism, but it is hidden from the hardware due to how the application was compiled. The latter simply lacks the thread-level parallelism necessary to directly take advantage of multiple cores.

This has led to a crisis in the microprocessor industry, as customers have come to expect that all of their programs will perform significantly better every

processor generation. Before the multicore era, performance gains for all applications were possible thanks to shrinking feature size and architectures that could increasingly exploit ILP (with superscalar pipelines and aggressive out-of-order engines). Now that architects have reached the limits of ILP exploitation and are using extra transistors to take advantage of TLP, applications not compiled for parallel execution or lacking TLP are falling behind. This dissertation explores ways in which these kinds of applications can be made to run efficiently on multicore platforms.

To improve the performance of legacy binaries that possess some inherent TLP but have been compiled for a single thread of execution, we automatically parallelize them at runtime. This is made difficult by the fact that the inherent parallelism is obfuscated during original compilation. Proving that memory accesses are thread-safe (i.e., are not aliased by potential parallel threads) is difficult (or impossible in some cases) for a traditional parallelizing compiler with source code access, but it is even more difficult when only the machine code is visible. The key to enabling parallelization in spite of this uncertainty is the use of hardware transactional memory (TM). Transactional memory is a promising synchronization mechanism enabling optimistic concurrency. It allows our dynamic recompiler to parallelize code that a traditional parallelizing compiler would be unable to parallelize due to potential memory aliasing.

Many applications (legacy or not) lack inherent TLP, making parallelization impossible. For these applications, we can improve their efficiency on multicore systems by taking advantage of heterogeneity. Traditional parallel applications are broken into sets of homogeneous parallel threads that easily map to homogeneous multicore systems. But the differing (heterogeneous) characteristics of single-threaded applications call for *heterogeneous* multicore systems to execute efficiently. The more heterogeneity, the more likely each application will find the right core to execute on—whether it be, for example, a powerful but under-utilized core for a CPU-intensive application, a small low-power core for a lightweight application, or a core with support for special instructions for a domain-specific application.

In this dissertation we explore how to leverage two different kinds of heterogeneity:

1. heterogeneity created by unbalanced load on homogeneous cores,

2. heterogeneity created by hardware asymmetry—a diverse set of different core types with multiple ISAs.

We first consider a homogeneous multicore system composed of multithreading (SMT) cores. This increasingly common architecture presents a difficult challenge because resource sharing, and therefore thread interaction, occurs in two dimensions. Applications running on different cores share relatively few resources; applications running simultaneously in different hardware contexts on the same core share many resources. Some combinations of applications share resources well, while others create heavy resource contention. Finding good schedules on these architectures proves to be a difficult task because the number of possible schedules to consider is large and positive thread interaction is difficult to predict. We observe that often the best schedules, especially when both performance and energy are primary concerns, are unbalanced ones—it is most efficient when some cores are more utilized than others.

Next, we consider a heterogeneous-ISA chip multiprocessors (CMP). The diversity of not only core types, but also ISAs creates a wide range of custom cores—each designed to efficiently execute a particular kind of application. But for programs to be able to take advantage of this heterogeneity, they must be free to move among cores. Migration can either be used to free up needed resources when new programs enter the system or to exploit different phase behavior within an application. We propose a new technique to quickly migrate programs among heterogeneous cores with a minimal sacrifice of runtime performance.

In the next section, we introduce our runtime parallelization technique for legacy binaries. In the following section we introduce our studies of two heterogeneous systems that use multiple cores to efficiently execute inherently serial applications.

1.1 Parallelizing Single-Threaded Code

Before the multicore era of general-purpose processor design, single-threaded legacy binaries saw significant performance improvements with every processor generation. Because multicore microarchitectures exploit TLP, not ILP, legacy binaries no longer experience significant performance gains.

Parallelization of single-threaded binaries, particularly at runtime, when more information is known about the behavior of the program, has been shown to be effective, to various degrees [CO03, OYOB02, THA+99, YF08, VRR+07, ZMLM08, vPCC07]. Many of these proposals show only modest performance gains, or show high performance gains but make assumptions about extensive new hardware support. From the prior research it is clear that some hardware support is necessary to achieve good speedup from parallelized legacy programs; but architects are reluctant to add expensive special-purpose hardware. We explore the re-use of hardware intended for a different purpose—hardware that is more likely to be implemented by processor makers: hardware transactional memory [HM93].

Transactional memory (TM) is a promising lock-free synchronization mechanism that has been the center of active research in the last few years with the explosion of multicore architectures. Transactional memory gives programmers and compilers powerful shared-memory synchronization primitives. Instructions are grouped into transactions that are guaranteed to commit atomically. TM provides optimistic concurrency: transactions are allowed to execute concurrently and the hardware monitors violations among memory operations contained in the transactions. If a violation is detected, the affected transaction is rolled back and restarted. Thus serialization is only imposed on parallel code if true memory aliasing is present; when it is not present (whether or not that can be guaranteed a priori), serialization is not enforced.

In this dissertation, we introduce a new runtime parallelization technique leveraging the optimistic concurrency of TM to automatically parallelize single-threaded legacy programs. This work addresses a number of challenges posed by this type of parallelization and quantifies the trade-offs of some of the solutions, such as how to select good loops for parallelization, how to partition the iteration

space among parallel threads, how to handle loop-carried dependencies, and how to transition from serial to parallel execution and back. The simulated implementation of runtime parallelization shows a speedup of 1.36 for the NAS benchmarks and a 1.34 speedup for the SPEC 2000 CPU floating-point benchmarks when using two cores for parallel execution.

1.2 Managing Heterogeneity

1.2.1 Scheduling in a CMP of SMT Cores

Applications that cannot be parallelized because they lack inherent TLP may still be able to benefit from multicore architectures, even homogeneous architectures, if they can take advantage of heterogeneity arising from unbalanced schedules to find the best core to execute on.

Before multicore architectures were common, scheduling on a processor that exposes only one hardware context was more about fairness (or responsiveness) than about efficiency, because there is almost no interaction among threads. Scheduling on a processor that exposes two or more hardware contexts of the same type (either multiple cores or multiple SMT contexts) creates a new challenge—scheduling for efficiency—because the interactions of threads executing concurrently determines performance and power usage. Previous work has shown that the coscheduling of threads on an SMT processor greatly impacts performance [ST00]. This is due to the high degree of resource sharing among SMT contexts. The importance of good coschedules is somewhat diminished in a multicore processor because resource sharing is usually limited to the last-level cache (and perhaps one other level of shared cache). In architectures that combine SMT and multicore design, finding good schedules becomes more complicated, as there are two degrees of resource sharing among contexts. Not only is it necessary to find good schedules to maximize performance, but recently, energy has also become a first-class metric. Often this results in the best schedules being unbalanced ones, creating a type of heterogeneity. Deciding which core to run an application on becomes critical.

In this dissertation we propose a new thread scheduling technique for this

increasingly popular architecture—chip multiprocessors with simultaneous multithreading cores. We describe optimizations for both performance and energy efficiency. The key contribution of our scheduler is its consideration of unbalanced schedules. Conventional multiprocessor scheduling, applied to this architecture, will attempt to balance the thread load across cores. We demonstrate that this approach eliminates one of the biggest advantages of this architecture—the ability to use unbalanced schedules to allocate the right amount of execution resources to each thread. However, to accommodate unbalanced schedules, the search space of all schedules (both balanced and unbalanced) is much greater than that of the balanced schedules alone. This dissertation proposes and evaluates scheduling mechanisms that allow the system to identify and migrate toward good thread schedules, whether the best schedules are balanced or unbalanced.

1.2.2 Migrating Execution on Heterogeneous CMPs

In Chapter 4 we investigate how a different kind of heterogeneity—hardware heterogeneity—can benefit serial applications on multicore architectures. Research has shown that architectures with multiple cores of different types can achieve greater performance and power efficiency [KTJR05, HM08]. In the past, when processor design centered around a single core, architects sought to make cores as general as possible, capable of performing well on different types of code. The abundance of cores in new architectures affords architects the ability to specialize some of those cores to run certain classes of code more efficiently. Heterogeneity is a powerful tool to balance generality and specialization.

Another factor that drives heterogeneous design is energy efficiency. Architects designing general-purpose processors must strike a balance between performance and energy efficiency. Often the two goals are in conflict. However, in a heterogeneous chip multiprocessor, different cores can be selected all along the power/performance curve—some cores designed for high performance, others suited to low-power operation.

Designing software to run on heterogeneous CMPs with cores sharing a common ISA is simpler than for heterogeneous-ISA CMPs. Heterogeneous-ISA

CMPs enable a new dimension of heterogeneity. Just as architectures are designed with different goals, so are ISAs. Some ISAs as designed to support a wide variety of instruction types, including sets of special-purpose instructions. Some ISAs are engineered to facilitate small, simple architectures. Also, assumptions about cache and memory latencies play a big role in ISA design. Multicore architectures that are able to exploit these differences will have a power/performance advantage over homogeneous-ISA architectures.

Efficient thread scheduling on heterogeneous-ISA CMPs is difficult because good thread scheduling needs to be dynamic. Not being able to reschedule running threads on different core types limits how effective the system is in exploiting hardware diversity. For example, when the energy level that is available to the system changes, the system should be able to adapt, migrating execution to low-power cores. Or when program behavior changes, execution should migrate to a core that will be able to handle the workload most efficiently. Or when a new high-priority process enters the run queue, the most powerful core(s) should be made available to it. But for a scheduler on a heterogeneous CMP to adapt to such changes, it needs to be able to migrate execution across core types.

This dissertation proposes a novel execution migration technique for CMPs with heterogeneous ISAs. The critical feature of these architectures, that we take advantage of, is their shared memory. We leverage this advantage by keeping the memory image of a migratable process in a state that is almost completely ISA-neutral. This allows migration to be fast, as very little state has to be transformed. We rely heavily on the compiler to both enforce memory image consistency and to extract the information necessary to transform the remaining architecture-specific state. We achieve fast migration (308 microseconds on average) with a minimal sacrifice of runtime performance (performance without migrations)—2.3% on average.

The rest of this dissertation is organized as follows. Chapter 2 presents our solution to getting improved performance from legacy single-threaded code on multicore architectures. Chapters 3 and 4 describe how programs that cannot be parallelized can still thrive on multicore architectures by finding and exploiting

heterogeneity. Chapter 3 focuses how to make use of heterogeneity stemming from unbalanced schedules, and Chapter 4 focuses on a critical first step toward exploiting hardware heterogeneity: migration on heterogeneous-ISA systems. Chapter 5 concludes.

Chapter 2

Runtime Parallelization

2.1 Introduction

General-purpose microprocessor design has undergone a major shift in the last few years as the performance and thermal characteristics of large out-of-order microprocessors failed to scale. The multicore era is upon us, and as feature sizes continue to shrink, dies are divided up into ever more processing cores. The increase in hardware parallelism has outpaced the software industry; there is a large body of applications written for or compiled for a single thread of execution, and these applications do not take advantage of the extra parallelism being offered by modern microprocessors.

Some single-threaded legacy applications are rewritten to take advantage of the additional hardware parallelism, but this is often a difficult and expensive undertaking. Some legacy applications are recompiled with a parallelizing compiler that attempts to extract as much parallelism as possible. But this is not always possible for a number of reasons. It is not possible if the software vendor only distributed the application in binary form and they have gone out of business, no longer support the application, are unwilling to recompile the application because they are worried that it may introduce or uncover bugs, or rely extensively on a compiler that does not support parallelization. Sometimes the user is unwilling to pay more for an updated version.

The software industry recognizes the growing need to produce parallel mul-

tithreaded code to take advantage of the offered hardware parallelism to make the performance of their software products competitive. They are becoming increasingly interested in ways of making the writing of correct parallel code less difficult and error-prone.

What makes parallel programming so difficult to implement correctly is the need to control access to variables in shared memory. Most of the traditional paradigms for controlling access to shared memory involve using locks to control access to regions of code that may operate on shared memory. Often times it is not known by either the programmer or compiler if, at any given moment during the running of a program, multiple threads will attempt to operate on the same memory at the same time. To protect against the possibility of such a conflict, locks are used to conservatively guard access to shared memory.

Parallelization at runtime is even harder because high-level program information is lost. Because a static compiler has access to a higher-level representation of the program, it is in a better position to reason about potential memory aliasing. Runtime parallelization operates on machine code, where it is much more difficult to determine where potential aliasing may occur. Consequently, nearly all memory references must be treated as potentially aliased and guarded with synchronization primitives, like locks, which force serialization.

Transactional memory (TM) is a promising improvement over lock-based synchronization. It optimistically grants access to shared memory, forcing serialization only when a real conflict is detected [HM93, AAK+05, CNV+06, RHL05, YBM+07, MBM+06a, MBM+06b, HWC+04, ST95, HF03, HLMS03]. TM is an area of active research and has been gaining increasing acceptance in industry [DLMN09]. For example, the Rock processor from Sun Microsystems supports hardware transactional memory. We believe that in the near future we will see more microprocessor vendors including hardware support for transactional memory.

Leveraging hardware support for TM, we propose a new technique to automatically (without any user intervention) extract thread-level parallelism from legacy single-threaded binaries with minimal architectural change. We find that

transactional memory enables the parallel execution of many loops that are serialized by traditional synchronization. Our technique uses a dynamic optimization framework: frequently executed loops are identified by hardware at runtime, a dynamic recompiler is spawned in a free hardware context to transform the key loops into parallel code, and the recompiled parallel loops are patched in to the running program. The dynamic recompiler analyzes the loop (at the machine code level, not at the source code level) and, if possible, transforms it into a loop that can be executed in parallel. When the parallelized loop executes, parallel threads are forked onto free hardware contexts.

The primary contribution of this chapter is to show that the reduced overhead and optimistic concurrency of hardware transactional memory enables the effective parallelism of a number of legacy codes, despite the existence of unknown and possibly unknowable memory aliasing.

This chapter describes our parallelization technique and quantifies its effectiveness across a number of benchmarks. In Section 2.2 we give an overview of the related work. Section 2.3 describes our baseline processor architecture, including transactional memory and dynamic optimization implementations. An explanation of our parallelization technique is found in Section 2.4, and a description of the parallel code generation is found in Section 2.5. In Section 2.6 we describe our experimental methodology, and in Section 2.7 we present our results. Section 2.8 concludes.

2.2 Related Work

2.2.1 Transactional Memory

Our parallelization technique relies on data speculation to avoid frequent synchronization. This requires a mechanism to prevent erroneous execution that arises from unanticipated memory aliasing between loop iterations. We achieve this through the use of transactional memory.

Transactional memory was first proposed by Herlihy and Moss as a hardware implementation of lock-free concurrent synchronization [HM93]. Their model

provides the programmer with guarantees about the memory accesses of instructions contained in transactions. One guarantee is that writes to memory within a transaction are not visible to other transactions until transaction commit (at which time all the writes in the transaction are atomically released). Another guarantee is that memory aliasing among transactions (e.g., a write in one transaction to the same address as a read in a concurrently-executing transaction) is detected and appropriate recovery actions taken (e.g., a transaction's state will be discarded and it will be restarted).

From this simple model, a number of significant improvements have been made, including more efficient ways of handling increased buffered state [AAK+05, CNV+06], virtualization [RHL05, YBM+07], improved conflict detection [MBM+06a], sub-cache line granularity [YBM+07], nested transactions [MBM+06b, YBM+07], faster transaction commits [MBM+06a], ordered transactions [HWC+04], and software-based transactional memory [ST95, HF03, HLMS03].

Handling Increased Buffered State

Ananian et al. [AAK+05] assert that a TM implementation should not place restrictions on transaction size. They propose two related hardware TM implementations: *Unbounded Transactional Memory* (UTM) and *Large Transactional Memory* (LTM). LTM is a simplification of UTM that does not require modifications to the memory interface. They use an in-memory data structure, instead of the cache architecture, to track transactional state, allowing transactions to grow as large as system memory.

Chuang et al. [CNV+06] also propose a hardware TM implementation that does not place restrictions on transaction size. Their proposal, *Page-based Transactional Memory*, integrates transaction bookkeeping with the virtual memory system.

Virtualization

Some researchers have proposed ways to virtualize TM so that the programmer is shielded from platform-specific resource limitations. Rajwar et al. [RHL05]

introduce *Virtual Transactional Memory*, a hardware-software hybrid TM implementation that uses virtualization techniques to deal with cache overflows and clock interrupts.

Fast Transaction Commits

In many TM implementations old memory state is kept in place until transaction commit, while transaction state is buffered. In order to make transactions commit faster, Moore et al. [MBM+06a] propose updating memory immediately with transaction state and writing the old versions of data to a log in case of transaction abortion. Their implementation, *LogTM*, also extends a directory-based cache coherence protocol to achieve fast conflict detection.

Sub-cache Line Granularity

LogTM Signature Edition [YBM+07] extends *LogTM* by adding signatures (hashes of read and write sets) to facilitate conflict detection. This decouples conflict detection from L1 cache tags and arrays. Because neither version management nor conflict detection are tied to the cache architecture, their hardware TM implementation can operate at granularities finer than cache lines. Our parallelization technique benefits significantly from this fine granularity (as we show in Section 2.7.2).

Nested Transactions

Many TM models support closed nested transactions, where all nested transactions are flattened into the outermost transaction. This preserves correctness when software composition is used (e.g., when module A opens a transaction and calls module B, which may open addition transactions), but may degrade performance. Moravan et al. [MBM+06b] propose support for open nested transactions, where an abort in an inner transaction does not cause the outer transaction to abort. They also allow escapes in the middle of a transaction in order to suspend transactional execution while non-transactional code (e.g., OS code) is invoked.

Ordered Transactions

In many TM implementations, transactions commit in an arbitrary order—whichever transaction completes first commits first. Recognizing that most parallel applications have places where one transaction must commit before another, Hammond et al. [HWC+04] introduce a hardware TM implementation, *TCC*, that supports ordered transactions. If the programmer wishes, he or she may specify the order of transactions in addition to the boundaries of each transaction. Hardware-managed phase numbers are associated with transactions; a transaction in a newer phase will stall if there are uncommitted transactions in older phases. Ordered transactions have also been proposed by Ceze et al. [CTTC06] and Porter et al. [PCT09].

Software Transactional Memory

First proposed by Shavit and Touitou [ST95], Software Transactional Memory (STM) is an alternative TM implementation strategy based completely in software. Performance is not as good without hardware support, but the solution is more portable and can be used on any machine that supports load-linked/store-conditional operations. Harris and Fraser [HF03] and Herlihy et al. [HLMS03] extend STM to object-oriented languages and propose implementations of dynamic STM—where transactions and transactional objects can be created dynamically.

Our TM Model

We assume hardware transactional memory, but otherwise, we are relatively insensitive to which implementation. We only require that the TM implementation support ordered transactions. Ordered transactions are only allowed to commit in a predefined order.

Advances in transactional memory should improve the performance of our parallelization technique. For example, faster transaction commits will lower the transactional overhead of our parallelization and improve performance; and support for the efficient buffering of more transactional state will allow for the parallelization of larger loops and more iterations per transaction.

2.2.2 Dynamic Optimization

A dynamic optimization framework supports our parallelization technique by efficiently discovering loops that are good candidates for parallelization, calling our dynamic recompiler to parallelize the candidate loops, and patching the parallel versions of loops into running code. Research on dynamic optimization is advancing rapidly. What follows is a summary of important advances.

Bala et al. [BDB00] In 2000 the first software-based dynamic optimization system was introduced. This framework, called *Dynamo*, is a software-only optimizer that is similar to, but also fundamentally different from Just-in-time (JIT) compilation. Like JIT compilers, Dynamo optimizes code as it is running; but unlike JIT, it optimizes native binaries, not bytecode. Dynamo operates by alternating between interpretation and direct execution—frequently executed portions of code (so-called *hot* traces) are optimized and placed in a code cache for direct execution (everything else is interpreted).

Chen et al. [CLG00] The dynamic optimization system *Mojo* is very similar to Dynamo, but adds support for exception handling and multithreaded code. It also distinguishes itself by supporting optimization for a CISC ISA—x86. As in Dynamo, in Mojo the optimization is typically performed in the same thread (and on the same hardware context) as the main execution.

Patel and Lumetta [PL01] They propose a hardware-based dynamic optimization framework, called *rePLay*, that enables aggressive optimizations by stringing together basic blocks in a single control flow called a *frame*. Frames are built speculatively and a hardware-based recovery mechanism reverts program state if an early exit is taken.

Bruening et al. [BGA03] They provide an interface to the software-based dynamic optimizer *DynamoRIO* (an IA-32 version of Dynamo) that makes it easy to create new optimizations. Their framework makes writing optimization modules for a dynamic compiler as easy as it is to write optimization passes for a static

compiler. The abstraction provided by their framework overcomes the difficulties (e.g., memory management) of interleaving the execution of the optimizer with the execution of the program being optimized.

Lu et al. [LCYcH04] In the *ADORE* dynamic optimization system, the execution of the optimizer is segregated to an OS-level thread. The framework uses the hardware performance counters in the Itanium 2 processor to detect hot regions of code; analysis and optimization is performed in software in the optimization thread. This hybrid hardware-software approach combines the best of both worlds. However, context switches to the optimization thread for profiling hamper performance.

Our Dynamic Optimization Framework

The dynamic optimization framework that we employ is a simplified version of the *Trident* framework by Zhang et al. [ZCT05, ZCT06, ZTC07], which is similar to ADORE. In Trident all of the profiling is handled in hardware and the optimization thread is executed on a separate hardware context, completely eliminating interference from the optimization thread on the main execution. Trident supports a broad selection of optimizations, many of which are orthogonal and complementary to our parallelization optimization.

Hot traces (frequently executed regions of code) are detected by simple hardware performance monitors; finding hot traces in hardware incurs much less of a performance penalty on the monitored process. Since the loop analysis and parallelizing recompilation is much more complex, it is best done in software. To accomplish this, when a hot trace is detected, the processor will spawn a new thread in a free hardware context to run a software dynamic recompiler to analyze and optimize the code. In our case, the hot code regions are loops and the optimization performed is parallelization. As Zhang et al. point out, doing the profiling in hardware and the optimization in a separate thread in a separate hardware context results in very minimal performance impact on the target process.

2.2.3 Dynamic Loop Parallelization

Other researchers have proposed the use of dynamic optimization to aid in runtime parallelization of loops.

Jrpm [CO03] parallelizes Java bytecode at runtime in the Java Virtual Machine. They also exploit optimistic concurrency in their parallelization; but unlike our work, which uses transactional memory to achieve this, they use thread-level speculation (TLS). Also, in their work, programmer transformations are necessary to expose loop-level parallelism; our technique is fully automatic and does not require any programmer or user intervention. The goal of our work is to improve the performance of legacy binaries, and we target compiled machine code (not Java bytecode); we address code that does not have the advantages of running in a controlled virtual machine environment, like the Java Virtual Machine.

Ootsu et al. [OYOB02] also propose a runtime parallelization technique for binaries using a dynamic optimization framework. Their work builds on the parallelization technique of Tsai et al. [THA+99] by adding binary translation at runtime, instead of transformation at compile-time, to perform the parallelization. Unlike our work or Jrpm, they do not explore the use of any optimistic concurrency mechanisms to ensure correctness with regard to concurrent memory access. Instead they rely on an explicit serialization of all the instructions that compute all the store addresses in the loop body. To ensure correctness, this code is collected in a serially-executing phase they call TSAG (Target Store Address Generation), that is executed at every loop iteration in every parallel thread (in original program order). Then, as the main body of the loop is executed (in what they call the Computation phase), a special hardware structure called the Memory Buffer coordinates memory dependencies among parallel threads.

Yardimci and Franz [YF08] put forth a dynamic parallelization and vectorization technique for single-threaded binaries. Their technique involves only control speculation, not data speculation. Therefore, they introduce a complex control-flow analysis (especially to handle indirect branches); but they do not parallelize loops whose induction registers do not have deterministic fixed strides, loops that have stores in conditionally-executed code regions (i.e., in the *then* clause of an

if-then statement), or loops with potential cross-iteration data dependencies. Because our approach leverages transactional memory for data speculation, we find ways around all these limitations and are thus able to parallelize more loops.

Vachharajani et al. [VRR+07] add speculation to Decoupled Software Pipelining (DSWP). DSWP parallelizes a loop by partitioning the loop body into stages that are scheduled on threads and executed in a pipelined manner, communicating results via a message-passing or buffering mechanism. Speculative DSWP breaks some recurrence dependencies so that the loop body can be broken into smaller pieces to increase scalability and load balancing. Our work differs from theirs in three ways: our technique works at the machine code level, where analysis and transformation is more complicated due to compiler optimization and loss of information; we use a simpler, more common form of speculation, transactional memory—they use a complicated versioned memory system; their technique utilizes heavy inter-thread (cross-core) communication, potentially requiring hardware support for best performance.

Works by Zhong et al. [ZMLM08] and Von Praun et al. [vPCC07] leverage TM for parallelization, but rely on programmer and/or heavy compiler support; our technique requires neither.

2.2.4 Speculative Multithreading

Speculative multithreading [SBV95, MGT98, KT99] is an alternate approach that attempts to get parallel speedup from serial code. They do so by executing serial code in parallel and recover from misspeculation. They typically rely heavily on data value speculation and prediction, as well as some type of memory versioning. Conversely, we create more conventional parallel code (dynamically), without data prediction, and only exploit speculative execution to the extent that transactional memory already supports it.

2.3 Architecture

Hardware transactional memory provides several key advantages over traditional synchronization, particularly with regard to the problem of runtime parallelization. The key problem in runtime parallelization, that has made it an essentially unsolvable problem except in the most simple cases, is that in the absence of any high-level program information, nearly all loads and stores must be treated as potentially aliased—the necessarily conservative handling of these potential dependencies serializes the code. Transactional memory, by supporting optimistic concurrency, solves a whole set of problems. First, because code is only serialized when there is true aliasing, conservative placement of synchronization has no cost. Second, we can include many writes in a single transaction, minimizing synchronization overhead with no significant loss in concurrency. Because transactional semantics require no correlation between the synchronization mechanism and the data that is protected, parallelization is simply enabled, yet still catches even unanticipated dependencies.

To illustrate the power of this technique, consider the pseudo-assembly code of the loop in Figure 2.1. This code loops over an array of structures: for each structure element, a pointer field is extracted and followed, and the data at that pointer location is modified. Assume this loop is executed many times, is part of a single-threaded application (for which we do not have the source code), and is running on a modern multicore processor on which there are one or more unutilized cores.

Using hardware performance counters, a dynamic optimization framework can detect that this loop executes frequently and can transform it automatically into parallel code—with different iterations of the loop running in different threads. Without the high-level code, we cannot guarantee that parallel iterations of this loop will not attempt to modify the same data in memory. With traditional lock-based synchronization primitives, our parallelizer would transform the loop such that a lock would have to be acquired and released on every iteration of the loop. Since only one thread may hold the lock at any time, even though the loop would be parallelized across multiple threads, the frequent synchronization would force a

```
loop: sub r1, r1, 256
      add r5, r1, r6
      load r2, 64(r5)
      load r3, 0(r2)
      add r3, r4, r3
      store r3 0(r2)
      bne r1, 0, loop
```

Figure 2.1: Potential memory aliasing makes this loop hard to parallelize.

serialization of loop iterations. However, if the loads and stores are not frequently aliased in neighboring iterations, the serialization is unnecessary.

If the optimizer instead uses transactional memory to make the code thread-safe, then when aliasing is infrequent, unnecessary serialization does not hinder performance. Each iteration is wrapped in a transaction and the transactions execute concurrently. Thus, with minimal analysis of the code, we still get guaranteed serialization of iterations when there are dependencies, and parallel execution in the absence of dependencies.

Transactional memory constructs are frequently utilized by programmers or compilers with a high-level (source code) view of the application. At this level alias analysis is much more feasible than at the machine code level. Synchronization code need not be applied in cases where it can be proven that aliasing is not possible. But a dynamic optimizer analyzes and transforms machine code. At this level it often impossible to prove that aliasing will not occur; so very conservative synchronization code is necessary. This makes TM a more attractive synchronization primitive.

2.3.1 Code Transformation Overview

We now present a high-level overview of the code transformation. Section 2.4 provides more details on this process.

Figure 2.2 shows what the transaction-wrapped transformed code from Fig-

```
        fork fix
        branch btx
loop:   EndTransaction
fix:    sub r1, r1, 256
btx:    BeginTransaction
        sub r1, r1, 256
        add r5, r1, r6
        load r2, 64(r5)
        load r3, 0(r2)
        add r3, r4, r3
        store r3 0(r2)
        bne r1, 0, loop
        EndAllTransactions
join:   ...
```

Figure 2.2: Wrapping the loop body in a transaction allows for safe and optimistic parallelization.

ure 2.1 would look like when targeting two parallel threads. Some details (mostly for bookkeeping) have been omitted to simplify the example. The code first forks a new thread to start executing at the label *fix*. This is a lightweight fork—no new stack is created, only registers are copied and the program counter is set. The newly-created thread will execute the induction code to bring the loop-carried registers up-to-date in preparation to execute the second iteration of the loop: in this example 256 is subtracted from register *r1*. Then, a new transaction begins and the loop body is executed. Meanwhile, the original thread will branch to the label *btx*, open a new transaction, and execute the first iteration of the loop. When a thread completes an iteration, i, of the loop, it will close the transaction and open a new transaction to execute iteration $i + 2$. The ordering of the transactions will ensure that state from each iteration is committed (i.e., data is stored to memory) in original program order. When the loop is done executing, all the transactions

will be closed, the spawned threads will be terminated, and execution will resume in the original thread at the *join* label.

Having ordered transaction commits ensures that we do not commit values out of order, and also ensures that the value of the last write to each memory location in the loop is the value that is visible at loop exit.

2.4 Parallelization

There are several things we need in order to enable dynamic parallelization of legacy serial code. First, we need some kind of dynamic optimization framework that can (1) identify candidate loops to parallelize, (2) spawn a thread to analyze each loop and generate parallel code, and (3) patch in the new version. We also need hardware support to catch dependencies between iterations assumed to be parallel (transactional memory, in our case). We need our parallelizer to solve the problem of selecting the right granularity of parallelism. We also need to ensure that we maintain the semantics of sequential execution, particularly as viewed by the code following the parallel region—transactions accomplish this for memory, but registers must be handled in software. Our solutions for each of these issues will be presented in this section and the following one. This section focuses on the overall design of the parallelization process, including the dynamic optimization framework that orchestrates the parallelization process and the scheduling of parallel code on threads. The next section will focus on aspects of the parallel code generation itself.

2.4.1 Dynamic Optimization Framework

The basic unit of optimization targeted by our technique is the loop. To identify loops most effectively, we combine two techniques—a whole-program control-flow analysis that identifies loops, combined with a hardware monitor that identifies frequent branches. This provides a more accurate view of important loops than trying to identify loops based on hot-branch addresses.

When a new process is started, a loop analyzer begins in a spare hardware

context to perform a quick loop analysis of the binary code. Since this is a static analysis, it can be performed once and the results even saved to a file for all future executions. The static analysis is not required, but improves the quality of discovered loops. Because it runs in a separate context and typically requires only a few milliseconds, it neither slows the main thread nor impedes our dynamic parallelizer in any but the shortest of applications.

The loop analyzer operates as follows. First, instructions are grouped into basic blocks; the control flow among basic blocks is modeled with a directed graph. A dominator graph is built; and any edge $i \rightarrow j$ in the control flow graph from a node i to a node j that dominates node i is the back edge of a natural loop (composed of all the basic blocks that form that cycle in the graph) [Muc97]. Loops that contain system calls or computed branches are filtered out because parallelizing such loops would be problematic. Information about each natural loop, like the address of the backward branch instruction, the branch target, the size of each loop, and the list of basic blocks that make up the loop is stored in a software buffer called the Loop Information Store (LIS) that is mapped into the address space of the target process.

As the program executes, a hardware-based profiler, like the Hot Path Profiler in [ZCT05], finds frequently executed loops, called *hot* loops. When a hot loop is identified, hardware monitors measure the average number of cycles required to execute one iteration of the loop. Once the baseline performance of the hot loop has been measured, a dynamic recompiler is spawned in a spare hardware context to attempt to parallelize the hot loop. The recompiler receives from the hardware profiler the identification of the hot loop and it finds a more detailed analysis of that loop in the LIS.

If the dynamic recompiler is able to parallelize the loop, it produces the machine code of a parallel version and inserts it into a region of memory, called the code cache, in the target process' address space. It modifies the first instruction of the serial (original) version of the loop to be a branch instruction to the parallel version of the loop. On the next instance of the loop, the parallel version will be executed.

One of the first instructions in the new version of the loop is a fast fork instruction. When this instruction is executed, a new thread is created on a spare hardware context. This is not a traditional heavy-weight fork. No new stack is created for the new thread and no registers are set, except for the program counter, which is set by adding the current PC to the offset encoded in the fork instruction. Because only the PC has to be set in the newly-created thread, the fork can be very fast.

As the parallel version of the loop executes, its performance is monitored in hardware. The average number of cycles per iteration of the parallel execution is compared to the average number of cycles per iteration of the serial execution. If the parallel version of the loop is not performing any better than the serial version, the parallel version is eventually removed, as in [ZCT05]. After all the iterations of a parallelized loop complete, parallel threads are terminated and serial execution continues on the original thread.

2.4.2 Partitioning the Iteration Space

Loop iterations are distributed among threads in a round-robin fashion. In our baseline implementation, this distribution is done at the granularity of individual iterations. We also experiment with distributing groups of iterations among threads, where each group is wrapped in a single transaction. The idea of partitioning the iteration space among threads as groups of iterations is commonly referred to as *tiling* and has been studied in part by [YF08]. The *tile size* is the number of consecutive loop iterations grouped together and treated as a unit of parallel work.

Large tile sizes are advantageous for three reasons. First, because multiple iterations are wrapped in a single transaction, the overhead of transaction start and commit is amortized. Second, loop induction code (code run before every tile to bring the loop-carried register state up-to-date) can be compacted. The simplest example of this is a register that is incremented by a fixed amount every iteration. Suppose we have two parallel threads and a loop that iterates 100 times, and the value in register *r1* is incremented by one each iteration. With a tile size

of one, before every iteration of the loop, *r1* will be incremented by one to account for the previous iteration that was executed on the other parallel thread. This will result in 99 dynamic instructions of overhead. With a tile size of 10, the value in *r1* can be incremented by 10 to account for the previous 10 iterations. This will only result in nine dynamic instructions of overhead. Finally, the last advantage of large tile size is applicable in cache configurations where false sharing is possible (e.g., for line-granularity transactional memory). If a loop's memory access pattern is such that a single cache line is written across multiple iterations, then false sharing can arise, resulting in frequent transaction restarts that hurt performance. This is an issue we directly address and will discuss in more detail later in this section.

Despite the many advantages of large tile sizes, there are a couple of disadvantages to consider as well. First, there is potentially more wasted computation when transaction restarts or aborts are encountered. Speculative execution is allowed to continue further and the consequence of a transaction that does not commit is more computation that has to be rolled back. Second, with larger tile sizes comes larger transactions. And with this, the possibility of transactional state overflow increases.

Given all these considerations, each loop has an ideal tile size based on factors such as iteration count, loop body size, induction code size, transactional overhead cost, probability of early exit, and probability of transaction restart. Finding a good tile size for each loop is important in getting the best performance from loop parallelization. On the other hand, we find that, for most loops, tile size does not significantly affect performance—other factors, like loop-carried dependencies and transaction restarts, play a much greater role in determining the performance of a parallelized loop (or whether it is even possible to parallelize a loop). For most loops, the tile size is more useful as a parameter to fine-tune a loop's parallel performance.

Given the marginal performance improvements of finding optimal tile sizes and the significant increase in the complexity of analysis required (which is contrary to the design philosophy of our *fast* dynamic recompiler), we do tile size selection in two ways. We can statically select a tile size that has been found to result in

good performance, in general. For the results presented in this chapter, the tile size we select is 16. Alternatively, we can use the dynamic optimization framework to iteratively re-parallelize loops and sample performance to discover better tile sizes.

Nevertheless, there is one simple compiler-based analysis that we find worthwhile to implement in our dynamic recompiler to aid in picking reasonable tile sizes. We find that when transactional memory is implemented at cache line granularity, loop parallelization often suffers from false sharing, which results in transaction restarts that hinder performance. We now discuss how a simple compiler-based analysis can be used to reduce this. To our knowledge, we are the first to propose such a stride-based tiling strategy to reduce false sharing in parallelized loops.

Stride Pattern Detection To Reduce False Sharing

When transactional memory is implemented at cache line granularity, false sharing becomes a concern. Consider the case of a loop that writes every other element in an array of 32-bit integers (one integer per iteration) and a cache with a 32-byte line size (each line holds eight integers). If each iteration is executed in a separate transaction on a separate thread, in almost every iteration there will be write-sharing of the same cache line. Since multiple writes to the same cache line in different transactions will cause a transaction restart because the memory system cannot effectively merge partial writes to cache lines, performance will suffer greatly. In the above example, if we tile four loop iterations per transaction, then the set of written cache lines can be completely disjoint among transactions, resulting in no transaction restarts. This is illustrated in Figure 2.3.

To determine the smallest tile size that will result in the minimum transaction restarts due to false sharing, our dynamic recompiler analyzes the stride patterns of all memory writes in each loop to be parallelized. The algorithm to compute the smallest necessary tile size is simple. For every memory write i, the least common multiple (lcm) of its stride, s_i, and the cache line size, L, is divided by the stride, s_i. This is the smallest necessary tile size for that write. To find the smallest tile size for the set of all n writes in the loop body, the least common

```
for (i = 0; i < 1024; i = i+2) {
  A[i] = i;
}
```

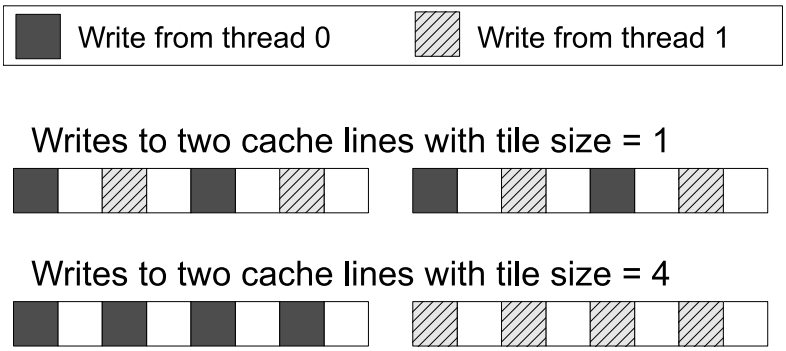

Figure 2.3: The above loop writes every other element in an array. When parallelized with a tile size of one, cache lines containing the array are written by both parallel threads. When parallelized with a tile size of four, the set of cache lines written by each parallel thread is disjoint.

multiple of the minimum tile sizes of each write is computed. Thus, the formula is

$$\text{Tile Size} = LCM(\frac{LCM(s_1, L)}{s_1}, \ldots, \frac{LCM(s_n, L)}{s_n}) \tag{2.1}$$

This serves only as a minimum tile size. If there are other reasons why an even larger tile size would result in even greater performance, a larger tile size can be selected, as long as it is divisible by this minimum tile size.

Following this procedure to pick a tile size that eliminates or reduces false sharing is no guarantee that it will do so. Some stores do not follow stride patterns. Also, more complex stride patterns are not detected by this technique. For some sets of strided writes, it is impossible to find a minimum tile size that separates cache line writes into disjoint sets. For example, if there are two writes in a loop body, both with a stride pattern of one (i.e., the writes are consecutive), and when the first write modifies word N in a cache line, the second write modifies word $N+1$ in a separate cache line, there is no tile size that will eliminate false sharing (assuming the cache line size is greater than one word). Despite the limitations of

this analysis, we find that the write behavior of most loops is regular, simple, and amenable to our simple analysis.

This stride-based tile size picking algorithm does not work well when the arrays are not aligned to cache line boundaries or when the loops that write arrays do not begin at a cache line boundary. Some compilers may attempt to align large arrays on cache line boundaries, but many do not. To address this issue, we propose running the first few iterations of a loop in serial mode until a cache line boundary is reached. To avoid over-complicating our dynamic recompiler, we elect not to implement this feature. Because the version of GCC that we use to compile our benchmark binaries does not align large arrays to cache line boundaries the performance gains in our benchmarks when using our stride-based tile size picking algorithm are minimal (2% for NAS and 4% for SPEC floating point). Since most of the stride patterns in our benchmarks are very simple, and because tiling has other benefits besides reducing false sharing, we find that it is sufficient to use a tile size that is a multiple of the number of words in a cache line, regardless of whether a strided memory access pattern is found.

2.5 Code Generation

This section describes the parallel code generation. There are a number of necessary features to parallel code generation. Parallel code must be able to take advantage of cross-iteration dependency checking (enabled by transactional memory). It must maintain the semantics of sequential execution, handling explicitly what TM does not—data flow through registers. And it must have some way of dealing with function calls in parallelized code.

2.5.1 Transaction Wrapping

The use of transactional memory in automatic parallelization is an important focus of our study. The key benefit provided by TM is optimistic concurrency among parallel threads in the face of statically-unknown memory sharing. While loop-carried dependencies though registers can be determined statically, depen-

dencies through memory cannot always be determined a priori. Some memory sharing can be determined statically, but this can require a complicated and costly analysis. The analysis is further complicated by the applications we are targeting—single-threaded legacy binaries with no source code available. For all these reasons, our dynamic recompiler analyzes loop-carried dependencies through registers but does not explicitly analyze memory sharing among parallel threads, relying on the TM system at runtime to detect and recover from memory violations caused by inter-iteration memory sharing.

Every loop iteration (or consecutive group of loop iterations, if tiling is used), is wrapped in an ordered transaction. The ordering of the transactions match the original sequential ordering of the loop iterations. Because loop iterations are distributed to parallel threads running on different cores in a round-robin fashion, the distribution of ordered transactions among parallel threads is round-robin. For example, if we have a tile size of two and we have two cores running parallel threads labeled *thread0* and *thread1* and a loop that iterates six times, then iterations zero and one will be wrapped in a transaction, *tx0*, and executed on *thread0*. Iterations two and three will be wrapped in another transaction, *tx1* and executed in parallel on *thread1*. Finally, iterations four and five will be wrapped in a third transaction, *tx2*, and execute after iterations zero and one on *thread0*. *tx0* on *thread0* will be required to commit first, *tx1* on *thread1* next, and *tx2* on *thread0* last. If a transaction ever reaches its commit point and there is an older (earlier) transaction that has not yet committed, execution on the thread processing the newer (later) transaction will stall and wait for all older transactions to commit.

Since the iterations (or groups of iterations) in every loop are ordered, it makes sense to think about the ordered transactions in each loop instance as a sequence of ordered transactions that start with transaction 0. Because the notion of sequences of ordered transactions, as opposed to a global ordering of all transactions, more intuitively maps to how we are using ordered transactional memory, we have implemented our ordered transactional memory model to support sequences of ordered transactions. At every loop instance, a new sequence of ordered transactions is started, and every transaction wrapping an iteration (or

group of iterations) is a member of that sequence.

We use a special instruction that marks the beginning of a sequence of transactions, BTS (Begin Transaction Sequence). This instruction is inserted in the prologue of the parallel loop and serves to inform the hardware-based transactional memory manager (TMM) that a new ordered sequence of transactions is about to start. To signal the TMM that a sequence of transactions should come to an end, the ELTX (End Last Transaction) instruction is used. This instruction not only marks the end of a sequence of transactions, but also serves to mark the end of a particular transaction in the sequence of transaction (the last transaction). An ELTX instruction is placed in the loop epilogue—a block of code executed by the primary thread after all the iterations of the loop have completed. In the following sections we discuss what other instructions compose a loop epilogue.

Opening Transactions

Individual transactions are started with the BTX (Begin Transaction) instruction; this signals the TMM that a new transaction is starting. One of these instructions is executed at the beginning of every iteration. In our TM model, as in most TM proposals, register state is saved when a transaction is begun in case the transaction fails to commit and has to be rolled back. We assume a shadow register file for this purpose. Memory instructions executed after the beginning of a transaction are said to execute in the transaction and follow the transactionally-extended cache coherence protocol (transactional bits are marked as appropriate and dirty memory is prevented from leaving the local cache).

Closing Transactions

Transactions can end in three ways. The most common way a transaction ends is by committing. When a transaction commits, the memory state written inside the transaction is made available to other threads. In the case of our cache-based TM implementation this means that cache words marked transactionally-written can be copied to private caches of other cores when requested, or written back to the L2 if evicted. Most transactions commit when they reach an ETX

(End Transaction) instruction and all older transactions have committed. If an ETX instruction is reached and there are uncommitted older transactions, the thread executing the transaction is stalled until all older transactions commit. An ETX instruction is executed at the end of every loop iteration (after the branch to a new iteration has been taken). The other way that a transaction may successfully commit is by the ELTX instruction that we already discussed. This instruction does the same thing as the ETX instruction, but also signals the end of a sequence of transactions. Whereas an ETX instruction is inserted at the end of the loop after the branch to a new iteration, an ELTX instruction is inserted in the loop continuation (in the loop epilogue).

The second way that a transaction may end is by an explicit abort. We implement this functionally in the ATX (Abort Transaction) instruction. This instructs the TMM to invalidate all the local cache words marked as transactionally written, effectively rolling back the memory state of the transaction (and the register state via the shadow register file). Explicit aborts also obey the transaction ordering semantics—no transaction may abort unless all older transactions have committed. Otherwise, speculative transactions with invalid data could cause aborts. ATX instructions are inserted down loop early exit paths. An early exit path is the path taken when a loop exits at a point different from the loop continuation. A C/C++ *break* instruction in the middle of a *for* loop is one example of an early exit. The ATX instruction is one of a few instructions inserted on all early exit paths that serve to prepare the process for the transition back to single-threaded execution.

The third way that a transaction may end is by a transaction restart. Restarts are triggered when a transactional memory violation is discovered (for example, if it is discovered that a store instruction in an older transaction wrote to an address that a load in a younger transaction read from). Violation detection is built into the cache coherence protocol and happens eagerly (during execution of the violation-causing memory instructions) instead of lazily (at the end of the transaction). This means that a transaction may restart at any time during execution. The semantics of a restart are virtually identical to those of an explicit

abort: cache words marked transactionally-written are invalidated and register state is restored. Execution then starts again from the BTX instruction that began the transaction.

2.5.2 Forking and Halting Parallel Threads

We now describe when and how parallel threads are created and destroyed. First, we define the *original thread* as the persistent, primary thread that will execute the target process in both single-threaded mode and parallel mode. We define *parallel threads* as those threads which are created to execute parallel portions of the code. During single-threaded execution, only the original thread executes; during the phases of parallel loop execution, the original thread and the parallel threads each execute some portion of the iteration space of the parallelized loop. When transitioning from single-threaded execution to parallel execution, parallel threads are forked. When transitioning from parallel execution back to single-threaded execution, parallel threads are destroyed. We discuss the forking process in Section 2.4.1; now we describe thread destruction.

Parallel threads are halted when a sequence of transactions comes to an end, which happens when either an ELTX or ATX instruction commits. When this happens, the PC of the original thread is set to the instruction immediately following the instruction that ended the transaction sequence. Note that an ELTX or ATX instruction may be executed in a parallel thread. But single-threaded execution will always resume after this instruction on the original thread; and all parallel threads, including (potentially) the thread that executed the ELTX or ATX instruction, will be halted.

After an ELTX instruction, the original thread executes the rest of the loop epilogue, and execution branches back into the single-threaded code immediately following the single-threaded version of the loop. After an explicit abort instruction (ATX) down a loop early exit path, the original thread re-executes that last partial loop iteration or group of iterations (if loop tiling is used) before branching back to serial code. This is necessary because the transaction abort will throw away the memory state of the partial last iteration or group of iterations. The reason

```
for (i = 0; i < 4; ++i) {
  B[i] = A[i];
  if (A[i] == 0) {
    break;
  }
  x = A[i];
}
```

Figure 2.4: C code to demonstrate loop early-exit procedure.

why we cannot simply execute an ELTX instruction and commit transactional state down an early exit path like we do for the loop continuation has to do with preserving correct register state. As we discuss in Section 2.5.3, register state is saved to memory (spilled) at the end of every iteration. Register state can then be read from memory (filled) by the original thread in the epilogue in order to capture the register state of the last iteration of the loop, regardless of which parallel thread executed the last iteration. The problem is that some register values may be incorrect on an early exit. Loop-carried register-bound dependencies will be correct because they are explicitly computed (described in Section 2.5.3), but some register values that are written in the loop body but not loop-carried may be incorrect.

To demonstrate this, consider the following example. The C source code for this example is in Figure 2.4. Suppose this loop is executed with one iteration per transaction and there are two parallel threads. Also, suppose that the compiler binds both i and x to registers. Iterations zero and two will be executed on the original thread and iterations one and three will be executed on the parallel thread. Note that the last iteration of the the loop will execute on the parallel thread. The register holding the value of i is loop-carried, so the dynamic recompiler will generate induction code for it (more on this in Section 2.5.3). This will mean that before all iterations, except the first, $++i$ will be executed to bring this loop-carried register value up-to-date. When iteration three begins, the register value of

the register bound to i will be two, by virtue of the induction code, and the value of the register bound to x will be *A[1]* since the last iteration to execute on this thread would have been iteration one. When iteration three ends, both registers will have the correct values. But suppose the early exit is taken from this loop in iteration three. i will be correct, but x will not be correct (it will be *A[1]* instead of *A[2]*).

This is why, on an early exit, the original thread must re-execute the last partial iteration—to capture the correct register state. It will also serve to create the correct memory state (since the memory state of the last partially-executed iteration would have been erased by the transaction abort). The original thread will fill the register state from the last successfully committed transaction and re-execute the last partial iteration in single-threaded mode. In our example, this would mean that the original thread would load the register state as it was at the end of the third transaction (after iteration two). The value of i would be two and the value of x would be *A[2]*. Then iteration three would be re-executed, setting *B[3]* to *A[3]* and incrementing i to three.

2.5.3 Data Flow Through Registers

Dependencies and data sharing among iterations through memory are handled by the transactional memory system at runtime, but dependencies and data sharing through registers must be analyzed by the dynamic recompiler and handled explicitly in the parallelized code. Dependency analysis involving only registers is significantly less complex than memory analysis, but there are several challenges to generating parallel code out of sequential. Loop-carried dependencies through registers are problematic because cores do not share registers. Also, on loop exit, the continuing thread must see a single register file with the latest update to each register, even though the last write could have occurred in any core. The latter issue requires register value updates when (1) register values are conditionally written, or (2) when the last iteration of the loop does not execute on the original thread.

```
1  store   r1, 0(r2)     // Mem[0+r2] <= r1
2  add   r2, 4, r2       // r2 <= r2 + 4
3  mult   r4, 5, r2      // r4 <= r2 * 5
4  load   r1, 512(r2)    // r1 <= Mem[512+r2]
5  add r3, r1, r2        // r3 <= r1 + r2
6  add r5, r2, r3        // r5 <= r2 + r3
7  add r6, r3, r6        // r6 <= r3 + r6
```

Figure 2.5: Pseudo-assembly code to demonstrate induction code generation.

Loop-carried Register Dependencies

We can manage most loop-carried dependencies by treating the dependent computation as induction code. When the dynamic recompiler analyzes a loop before parallelization, in addition to building a control-flow graph for the loop, it builds a data-flow graph to model the flow of data though registers. A list of registers that are loop-carried is generated by identifying all registers that are read before being written in the loop body. Induction code is extracted from the loop by following the data-flow graph backwards, starting at the last writes to each loop-carried register, including every instruction that is necessary to compute the new values of the loop-carried registers. This induction code is added to the beginning of the loop body to generate the correct live-ins.

Consider the pseudo-assembly code in Figure 2.5, representing a portion of a loop body. Registers *r1*, *r2*, and *r6* are loop-carried. The last write to register *r1* is on line four (the load instruction); this instruction is data-dependent on instruction two. Register *r2* is last written in instruction two. And register *r6* is last written in instruction seven; this instruction is data-dependent on instruction five, which is data-dependent on instructions two and four (which is dependent on instruction two). Therefore, the induction code would include instructions two, four, five, and seven. Executing these instructions in this order brings all the loop-carried registers up-to-date (i.e., accounting for the previous iteration executed in another parallel thread).

Conditionally-written Registers

Registers written in conditionally-executed basic blocks are problematic for two reasons. When they are loop-carried registers, the conditional statements can cause significant induction code expansion. For non-loop-carried registers, correctly identifying the last write to the register is difficult. In both cases, we exploit the transactional memory system's facility of guaranteeing correct ordering.

To keep the induction code small, we do not include control flow in induction code. Instead, both loop-carried registers and non-loop-carried conditionally-written registers are passed through memory and no induction code needs to be generated for them—transactional memory will ensure correct ordering. This is enabled though the use of spill (store) and fill (load) instructions. For each conditionally-written register, the dynamic recompiler inserts a spill instruction at the end of the basic block(s) where the register is conditionally written. If the register is loop-carried as well, a fill instruction for that register is inserted at the beginning of the loop. For all conditionally-written registers, a spill instruction is inserted in the loop prologue (to be executed before any iterations) and a fill instruction is inserted in the loop epilogue and down early exit paths. For loop-carried conditional register dependencies that are infrequent, this allows the code to typically execute in parallel. When the dependencies are frequent, there will be frequent restarts, and hardware monitors will recognize our failure to achieve speedup on the loop; this will cause the loop to revert to the original code.

For a non-loop-carried conditionally-written register, this mechanism serves to provide the original thread with the correct register value after loop termination (and parallel thread termination), regardless of which thread may have last executed the conditional block that wrote the register. For a loop-carried conditionally-written register, this mechanism serves to safely forward the loop-carried register value when necessary. Parallel execution is optimistic in that subsequent iterations running on parallel threads never wait for the forwarded value. Only when the conditional code is actually executed and the value becomes loop-carried is the later iteration held up. There is no need for any additional mechanism to detect these cases and recover when they arise because the TM sys-

```
1 for (i = 0; i < 1000; ++i) {
2   old_count = count;
3   if (A[i] == '!') {
4     count++;
5     last = i;
6   }
7 }
```

Figure 2.6: C code to illustrate two kinds of register writes. Assuming that the compiler binds *count* and *last* to registers, the registers representing these variables are conditionally-written. *count* is loop-carried and *last* is not.

tem will detect the sharing (at the spill address) and restart the newer transaction that had run ahead optimistically without the forwarded value.

The C code in Figure 2.6 illustrates both kinds of conditionally-written registers. This loop iterates over a character array and counts the number of exclamation marks found, recording the location of the last mark for later reference. Assuming the variables *count* and *last* are bound to registers, *last* represents a non-loop-carried conditionally-written register and *count* represents a loop-carried conditionally-written register. The dynamic recompiler will insert two register spill instructions at the end of the basic block where *count* and *last* are conditionally written (after line five) to store the value of *count* and *last* to memory. If the last iteration where *last* is updated is executed on a parallel thread, that thread can be safely terminated and the correct value of *last* filled from memory by the original thread after loop termination. The value of *count* is filled from memory at the beginning of every iteration. In the common case, where the last character was not an exclamation point and the conditional code was not executed, the private caches of all parallel threads will read-share this value and the same value will be read every time. When the conditional code is executed and the new value of *count* is spilled to memory, parallel iterations that follow that iteration will be restarted and will then read the correct value for *count*.

Another type of control-flow (in addition to conditional control-flow) that

complicates register dependence handling occurs in nested loops. If a loop-carried register dependence is in a nested loop or depends on code in a nested loop, then induction code would have to include the nested loop; however, this is undesirable because it significantly increases the overhead. Using the spill/fill technique that we use for conditional code would hurt performance as well. The spill/fill technique performs well for most conditional code because, for infrequently executed conditional code, transaction restarts caused by the loop-carried dependence are rare. Forwarding loop-carried register values through spills and fills in code that is guaranteed to execute every iteration results in frequent transaction restarts, which drives performance down. As a result, we do not parallelize loops with induction code (for an outer loop) inside nested loops.

Register State of the Last Iteration

We must also identify the last writer of registers that are not conditionally written. This is an easier problem, but may still involve data transfer. Unlike conditionally-written registers, at loop termination we know exactly which thread had the correct, most up-to-date value of the non-conditionally-written registers: the thread that executed the last iteration of the loop. This allows us to track register values without any restart-inducing data sharing. At the end of every loop iteration, register state is spilled to a thread-specific memory location; no cache lines have to be write-shared by different threads for this. Upon loop termination, after the parallel threads have been halted, the original thread fills the register state from the memory location of the thread that executed the last loop iteration. The dynamic recompiler inserts the fill code in both the loop epilogue and the early exit code. When executed on an early exit, the register state of the last full iteration (before the iteration that took the early exit) is loaded because the transaction abort that is executed on an early exit causes transactional state to be rolled back. This sets the state of the original thread to a place just before the iteration that will take the early exit. Recall from Section 2.5.2 that after an early exit is taken, the original thread re-executes the last iteration of the loop; so loading the register state as it was just before the last loop iteration is necessary

to prepare for this re-execution.

We experiment with two kinds of spills and fills to preserve correct register state. The first kind of spill/fill is a single register spill/fill, similar to the kind we use for conditionally-written loop-carried registers. Using this kind of spill/fill requires finding all the written (live-out) registers and inserting individual spill/fill instructions for each. Finding written registers is straightforward, but the runtime overhead of executing many spill/fill instructions is costly. The number of registers that need to be spilled and filled could be reduced if we knew which registers were actually live-in to the code following the loop. However, this kind of live-in analysis at the machine code level turns out to be difficult (especially in the presence of computed branches) and costly (because many potential execution paths have to be explored). Instead of trying to discover all the registers that are live-in to the code following the loop or finding all the registers that are written in the loop and inserting spill/fill code for each one of them, we use register file spill/fill code that stores and loads all the registers to/from memory. These instructions consume more memory bandwidth but cut down on the dynamic instruction count and allow the dynamic recompiler to execute faster, as it does not need to perform a more complicated register analysis. More complex spill/fill instructions could be introduced to improve performance, like spill/fill instructions that take a bitmap specifying which registers to store/load. Such instructions exist in the ARM ISA.

Loads in the Induction Code

Load instructions in the induction code pose a unique problem. Since the induction code represents a portion of the instructions that execute in the prior iteration, including a load instruction essentially reorders memory operations (if the loop body contains a store instruction)—because the load may execute after other memory operations in the previous iteration that it should execute before.

For example, consider the C code in Figure 2.7. First, *A[i+1]−>weight* is loaded. Then the value of *last*, which is *A[i]−>weight*, is stored to *B[i]−>weight*. *last* is a loop-carried dependence and, assuming it is bound to a register, it is a loop-carried register dependence that can be handled with induction code.

```
last = A[0]->weight;
for (i = 0; i < 1000; i++) {
  tmp = A[i+1]->weight;
  B[i]->weight = last;
  last = tmp;
}
```

Figure 2.7: C code that iterates over two arrays of pointers to structs (that have a *weight* field) and copies the weights in one set of structs to the other set.

The induction code would include the load instruction and would essentially load *A[i+1]->weight* into *last*. But suppose that *A[i]* and *B[i−1]* point to the same struct. If the previous iteration has finished execution, the induction code may load the new weight of the struct instead of the old weight, which is incorrect because, in serial execution, the old weight would have been saved in *last* (since the load should occur before the store).

Because memory instructions in induction code can cause memory ordering errors, all loop-carried registers that depend on a load should be checked to ensure that the correct value was loaded. To avoid any serialization of parallel code, we place the check at the end of every transaction, before the transaction commit. For every loop-carried register that depends on a load, the live-in state of that register at the beginning of the transaction is checked against the live-out state of that register at the end of the previous transaction. If the register values do not match, the correct register value is loaded and the transaction restarts. Since the check happens just before transaction commit and part of the live-in register state is being compared to the live-out register state of the previous transaction, there needs to be a mechanism to store register state at the beginning of the transaction. But this is already a requirement of the transactional memory system—register state has to be saved in case the transaction is aborted/restarted and transactional state rolled back.

Fortunately, cases where a loop-carried register depends on a load (in loops

that contain store instructions) are infrequent. But for those registers that are affected, the dynamic recompiler inserts code before transaction commit to compare the register values.

2.5.4 Function Calls

Function calls in loops create a number of challenges for parallelization. Recall that parallel threads are created with a lightweight fork instruction that does not create a new stack. So parallel threads share a common stack space in memory. This works well for easily sharing and using stack data, like local variables in the scope of the loop, but difficulties arise when multiple parallel threads try to grow the stack and create new local variables, as is necessary when a function is called. If unchecked, threads would overwrite the data of peer threads.

Function calls are also problematic because control flow leaves the carefully-analyzed loop code and enters serial code. Not only would we have to ensure that this code is parallelization-safe, but we may also need to modify this code to support parallelization. If a called function calls other functions, even if this rarely happens (e.g., to handle corner cases or error conditions), then those functions would need to be analyzed and potentially modified as well. The amount of code that must be analyzed and transformed to parallelize a simple loop can grow very quickly when all called functions must be explored.

We also have to ensure that every function returns to the call site. While high-level programming languages can provide this guarantee and a compiler may be easily able to assert this, machine code does not provide these guarantees and analysis at this level is not straightforward. For example, at the machine code level a function call usually takes the form of a branch instruction that has the side-effect of saving the next PC value in a given register; and a function return is implemented as a branch instruction whose address is taken from a register. By convention, most assembly programmers and compilers use a standard register to save the return value, though this is not required; and there are some cases where this convention is not followed. Also, there is no requirement at the machine code level that the register containing the return address is not modified, or even that

the same register be used for the function return. This means that there is no guarantee that a function ever returns or that it returns to the call site. Our dynamic recompiler must check for these possibilities. But this analysis becomes much more difficult if the return address is saved to memory—which is typically done when another function is called. Ensuring that the return address is not modified requires a difficult and expensive alias analysis to check that no other writes modify the address containing the return register value.

These problems can be solved by inlining functions. An inlined function is not allocated its own stack frame—so we do not have to deal with parallel threads allocating new stack frames. The second problem is solved because as the dynamic recompiler analyzes and prepares a function for inlining, it ensures that it returns to the call site.

Our recompiler inlines one level of function calling. Any function calls within called functions are replaced with early exit points that facilitate the transition from parallel execution back to serial execution at the point of the function call site. This does not completely prevent the problems springing from having a shared stack, because parallel threads will still be using the same stack memory for the inlined functions. However, the transactional memory system will prevent violations on the ordering of memory operations to the shared stack—sequential semantics will still be preserved, though parallel performance may suffer a little. Function inlining allows us to easily modify a copy of the function intended for parallel execution without making changes to the original copy of the function intended for sequential execution.

This also allows us to guarantee proper function return and the correct handling of the return address. We must ensure that after parallel execution completes, a return address pointing to a call site in parallel code is not followed. Therefore, the address of the call site in the serial code must be saved to the return address register. To accomplish this, the dynamic recompiler transforms the function call into a sequence of instructions that (1) explicitly saves the address of the original call site to the return register specified in the original branch instruction and (2) branches to the inlined copy of the function in the parallelized loop code. Likewise,

Table 2.1: Architecture Detail

Cores	2, in-order	Shared L3 cache	4M, 2 way
Total Fetch Width	4	L1-L1 transfer	14 cyc
Int/FP regs/core	100/100	Load-use, L1 hit	2 cyc
I cache/core	64k, 2 way	Load-use, L2 hit	16 cyc
D cache/core	64k, 2 way	Load-use, L3 hit	58 cyc
Shared L2 cache	512k, 2 way	Load-use, L3 miss	158 cyc

the return instruction is transformed into a fixed branch back to the call site in the parallel loop code.

2.6 Experimental Methodology

This initial study of the use of hardware transactional memory to facilitate automatic runtime parallelization of legacy code is in some respects a feasibility study to determine to what extent we can expose the available parallelism. For that reason, we keep our execution model relatively simple—we assume two cores. One core runs all the serial portions of code as well as one of the parallel threads in parallelized code regions. The other core executes parallel threads when available and the dynamic compiler thread (which only runs about 3% of the time). The maximum expected parallel speedup is 2.0.

We have chosen to model low latency cache-to-cache transfers: 14 cycles. We have done so for two reasons. First, we feel that these latencies will drop significantly as chip designers get better at designing CMPs with more hardware parallelism. The old shared-bus model is showing its age, not originally being designed with CMPs in mind. As more sophisticated core interconnects are developed, cache-to-cache transfer latency will drop. Second, we find that for any speculative parallelization technique, including our own, if the memory sharing among parallel threads is high, as it usually is in most integer applications, high cache-to-cache transfer latency can degrade performance.

See Table 2.1 for more details of our processor architecture. In the following section, we describe the architecture of our transactional memory model.

2.6.1 Transactional Memory

We model a generic transactional memory system. The only relatively uncommon characteristic that we require is the support for ordered transactions. Many proposed transactional memory systems view all concurrent transactions as equal; however, to preserve program order we require preferential treatment for transactions executing earlier (less-speculative) loop iterations. The ordering of transactions is considered in decisions regarding which transaction(s) to restart when a memory violation occurs, and in restricting when a transaction can commit. The modifications necessary to add ordering among transactions are straightforward and a number of transactional memory proposals support ordered transactions [HWC+04, CTTC06, PCT09].

We model our transactional memory system at word granularity because we find that memory violation detection at cache line granularity results in poor performance for many loops due to frequent transaction restarts caused by false sharing. Consider the case of a very simple loop that writes successive elements in an array of integers. If each iteration is wrapped in a transaction and executed concurrently, conflict detection at cache line granularity would result in false sharing as each parallel thread tries to write to different parts of the same cache line. Because of the significant performance advantages offered by violation detection at word granularity, we assume this granularity. Note that our parallelization technique is compatible with cache line granularity violation detection and will still yield a speedup, just not as great. A further discussion of some of the trade-offs of granularity and a performance comparison can be found in Section 2.7.2.

In trying to model as generic a transactional memory system as possible, we assume a TM system that is similar to the simple original model proposed by Herlihy and Moss [HM93]. Like many newer TM proposals, we model the buffering of transactional state in the local caches instead of a special transactional buffer. The traditional MESI cache coherence protocol [PP84] is extended to support memory violation detection (as is done by Hammond et al. [HWC+04] and Porter et al. [PCT09], for example).

A few bits are added to cache lines to support violation detection via the

cache coherence protocol: a sub-word-write bit, a stale bit, and three bits per word: a transactionally-written bit (TXW), a transactionally-read bit (TXR), and an unsafe bit. If line granularity is used, only a stale bit, a single TXW bit, and a single TXR bit are necessary. The sub-word-write bit is used to prevent false sharing. It is set if less than a word is written. If this bit is set on a cache line and if two parallel threads write to the same line or an older transaction writes to a line that a newer transaction reads from, a violation is asserted and the newer transaction is restarted. If the sub-word-write bit is not set, then false sharing WAW and RAW dependencies do not pose a problem and do not trigger a transaction restart. The stale bit is set on a line that an older transaction reads from and a newer transaction writes. The read value is only good for the duration of the transaction and the line is invalidated on transaction commit. The TXW and TXR bits are set on words (or lines if line-granularity is used) that are written or read (before ever being written), respectively, in a transaction. If a word is written in an older transaction and read (before being written) in a newer transaction, a violation is asserted and the newer transaction (and all transactions newer than it) are restarted. The unsafe bit is set when another thread writes to a word in a shared line. A later access to a word marked with an unsafe bit results in a violation and transaction restart. It is out of the scope of this chapter to explain all the details of this cache coherence protocol. For a more detailed explanation see Porter et al. [PCT09].

2.6.2 Simulation and Benchmarks

We now describe our simulation methodology and the benchmarks we use to evaluate the performance of runtime parallelization.

To avoid program start-up behavior, we simulate steady-state execution well into the program. Therefore, we assume that prior to the measurement interval, the one-time static program analysis has completed, and that loops that our system would try to parallelize but fail to achieve speedup have already been identified and rejected. The mechanics and efficiency of the code cache uninstallation process has been shown in prior work (e.g., Zhang et al. [ZCT05]). Thus, our results are

somewhat optimistic; but, in that prior work, it was found that after an initial warm-up, the code changed very infrequently.

To measure the effectiveness of our parallelization technique, we implement a dynamic recompiler to parallelize loops in binaries compiled for the Alpha architecture. At its core is a disassembler and transactional memory-aware parallelizing compiler. For each loop it is given to parallelize, it generates a parallel version of that loop in Alpha machine code. Built around its core are two interfaces: a stand-alone interface so that it can be run as a stand-alone executable in order to aid in debugging, and an interface to the dynamic optimization framework, allowing for efficient communication of parallelization tasks. The dynamic optimization framework passes it hot loops as it detects them, and the dynamic recompiler passes back parallel versions of those loops. The dynamic recompiler was not designed to be a robust full-featured compiler, but to be very small and lightweight, allowing it to quickly recompile loops at runtime.

We extended SMTSIM [Tul96a], an event-driven Chip Multiprocessor (CMP) and Simultaneous Multithreading (SMT) processor simulator, to support transactional memory and a dynamic optimization framework. The simulator is configured as a CMP and executes Alpha binaries, including our dynamic recompiler compiled for the Alpha ISA.

We focus on legacy code, and in particular we want to address two types of applications—those for which thread-level parallelism is clearly available but the code was compiled single-threaded to run on legacy processors, and those clearly written for single-threaded execution, but some thread-level parallelism may still be available. We use NAS 3.3 benchmarks to represent the former (except *dc* which does not compile properly for Alpha OSF/4 using GCC) and all the SPEC2000 CPU floating-point benchmarks to represent the latter. All the benchmarks were compiled with GCC 4.3 at optimization level -O2. Even when thread-level parallelism is clearly evident in the source code, we find that compiling for a single thread at this high optimization level seriously obfuscates that parallelism in many cases.

The NAS benchmarks are each fast-forwarded one billion dynamic instruc-

tions before detailed simulation in order to skip over program initialization. A SimPoint [SPHC02b] analysis is performed for each SPEC benchmark to find representative points of execution for detailed simulation. The A inputs are used for the NAS benchmarks and the reference inputs are used for the SPEC benchmarks. For the SPEC benchmarks that have multiple reference inputs, the first (alphabetically ordered) inputs are used.

For the results we present in this chapter, loops are parallelized into two threads and the default tile size is 16.

2.7 Results

The performance gains of our parallelization with constant tile size are presented in Figure 2.8 (the dark bars). The average speedup among the NAS benchmarks is 1.36 and the average speedup among the SPEC FP benchmarks is 1.34. The performance of some benchmarks, like *mg*, *mgrid*, and *swim*, came close to the theoretical limit of 2X speedup. Others, like *facerec* and *fma3d* (to name a couple), see no performance gain. In general, despite the challenges of identifying, transforming, and exploiting parallelism in serial code at runtime, we are successful, to some degree, in a significant percentage of the applications.

There are three primary reasons why some benchmarks cannot benefit from parallelization. First, in some cases, there is an inherent lack of thread-level parallelism. In other cases, thread-level parallelism is present, but it is not expressed in a way that is amenable to our parallelization technique. For example, a reduction [PE95], like a loop that sums up values in an integer array, has inherent parallelism; but if it is not coded carefully to express that parallelism, a critical loop-carried dependency will be created between every consecutive iteration of the summation. Since our dynamic recompiler operates at a very low level, it is not always able to recognize and transform every expression of parallelism. Third, and most importantly, some optimizations (like software pipelining) and machine code generations during original compilation obscure or impede subsequent parallelization. Thread-level parallelism that is evident in the high-level language of the

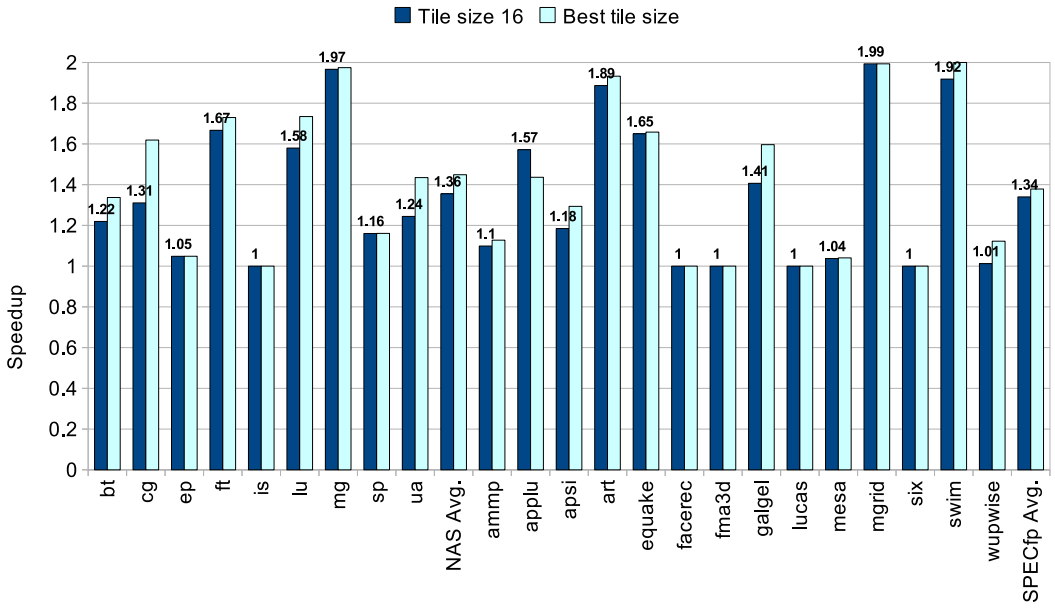

Figure 2.8: Speedups of NAS and SPEC FP benchmarks. The dark bars and data labels show performance when the tile size is fixed at 16. The light bars show performance when an optimal tile size is selected for each loop.

source code may not be obvious or may not exist in the optimized single-threaded binary. The poor performance on *ep* is clearly an example of this case.

The number of parallelized loops in each benchmark that contribute to overall speedup vary quite a bit. Some benchmarks, like *cg*, *mgrid*, and *swim*, contain only one or two important parallelized loops that account for the performance gain. Other benchmarks, like *lu*, *ua*, *applu*, and *art*, have eight or more important loops that contribute to their performance gain. The benchmark *ua* has 13 important loops that each contribute a performance gain of a few percent.

2.7.1 Tile Size

Figure 2.9 demonstrates how variance in tile size affects performance. For benchmarks in both suites, some degree of tiling is necessary in order to obtain the highest performance gains; however, we find that tiling is not absolutely necessary to see reasonable performance gain. The best tile size for NAS benchmarks is 16,

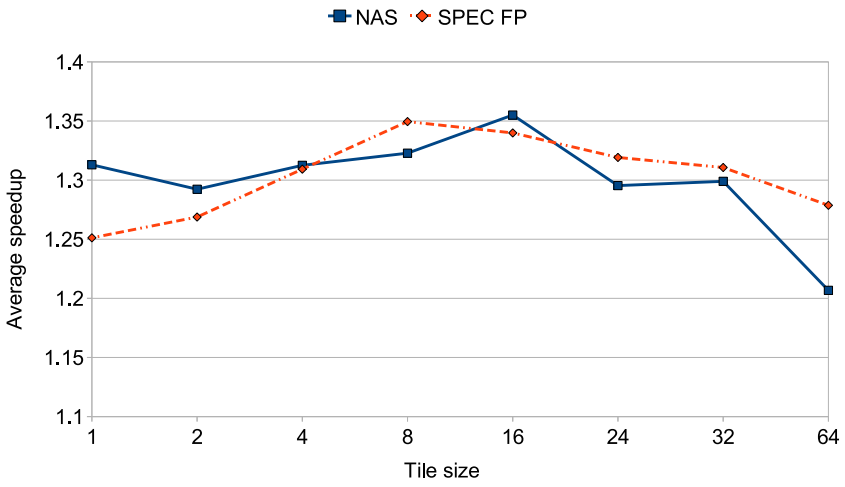

Figure 2.9: Speedup across various tile sizes.

while the best tile size for SPEC FP benchmarks is eight. As tile size increases much beyond 16, performance decreases in spite of the decreasing parallelization overhead. Because transactions are larger, transaction restarts are more costly because more worthless instructions are executed—instructions whose effects will be undone. Also, large tile size creates a load imbalance among parallel threads during the last iterations of the loop.

When the tile size of each loop is varied and the best-performing parallel loop version is selected, the distribution of ideal tile sizes is highly distributed: 19% of the time the best tile size is one, 9% of the time it is two, 9% of the time it is four, 15% of the time it is eight, 8% of the time it is 16, 14% of the time it is 24, 11% of the time it is 32, and 15% of the time it is 64 (the complete distribution of best tile sizes is given in Figure 2.10). These results imply that for best performance, the dynamic optimization framework should attempt parallelizations at a variety of tile sizes. Conversely, if loop analysis can find the optimal tile size, it could reduce the number of parallelized loop versions to be tried. Event-driven dynamic compilation has been shown to be quite effective at just this kind of trial-and-error optimization [ZCT06] due to the very low overhead of recompiling threads that run in another core.

The light bars in Figure 2.8 show speedup when the optimal tile size per

loop is selected. The average NAS benchmark speedup is 1.45 (a 9% performance improvement) and the average SPEC FP speedup is 1.38 (a 4% performance improvement). The performances of most benchmarks improve very little when optimal tile sizes are selected on a per-loop basis instead of a single static tile size. However, a few benchmarks, like *cg* and *galgel*, show a more significant performance improvement. The performance of *applu* is slightly lower due to a negative interaction between some of the loops of different tile sizes (minor second-order caching effects) that our loop selection mechanism is not able to detect a priori.

In all of the results presented so far, parallel loops of various degrees of nesting are installed. We experiment with two more constrained selection policies. In the first of these policies the dynamic optimization framework selects only parallelizable innermost loops. Loops that have no nested inner loops are included in this selection. In the second policy, only outermost loops are selected. Loops that are not nested within any other loops are included in this selection. Based on how we have defined these policies, loops that are not nested in any other loops and do not contain any nested loops themselves are selected in both policies. Note that there are some loops that fall in neither category: loops that are contained within another loop and have a nested inner loop. We compare the performance of parallelization under these two policies and our unconstrained policy.

The performance results are given in Figure 2.11. For most benchmarks, parallelization of the innermost loops results in better performance than parallelization of the outermost loops. There are three reasons why innermost loops are generally better candidates for parallelization: the thread-level parallelism our technique exploits most readily is fine-grained; fine-grain thread-level parallelism is more abundant than coarse-grain thread-level parallelism in the code we run; also, because the size of outer loops are greater, the restart cost of a failed transaction is greater. Larger loops have a greater potential for inter-iteration dependencies, resulting in more transaction restarts that hurt parallel performance. This figure, then, shows two key results. If we are trying to really minimize the complexity of the dynamic recompiler threads, restricting it to innermost loops is very effective. However, there is a reasonable gain to considering all loops as candidates.

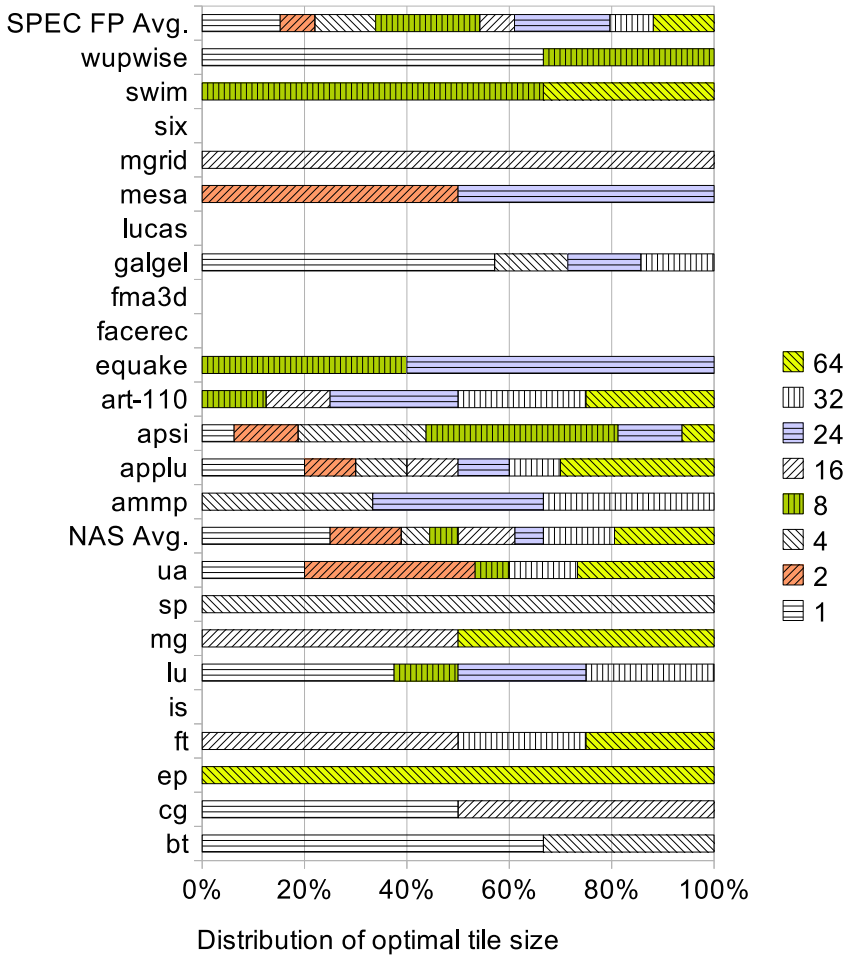

Figure 2.10: This shows, for each benchmark, what fraction of parallel loops in that benchmark have the best performance at various tile sizes. Those benchmarks with no bars are benchmarks where no loops could be effectively parallelized.

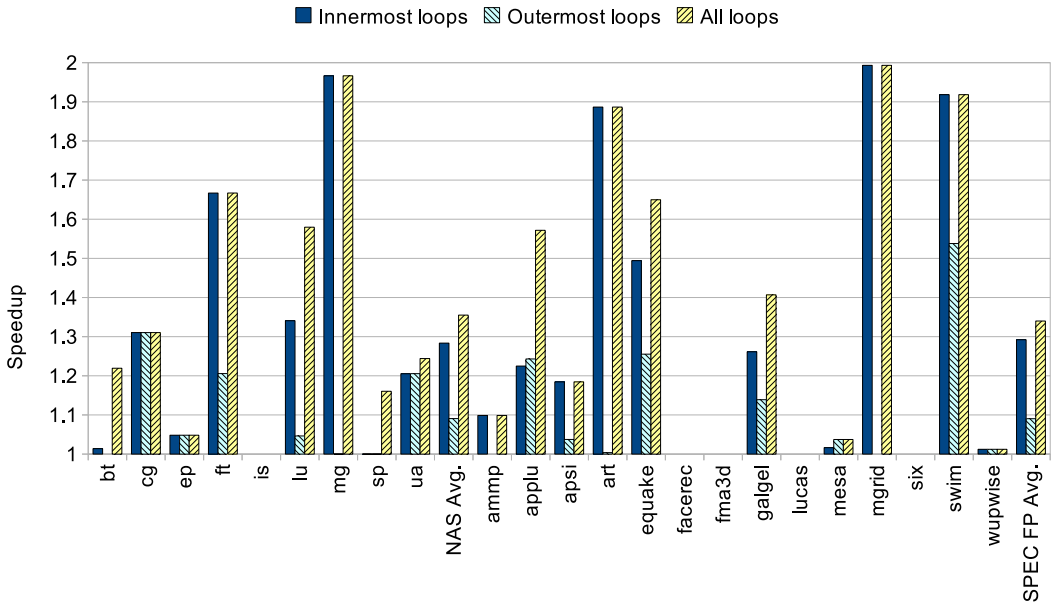

Figure 2.11: Speedups of NAS and SPEC FP benchmarks under different loop selection policies: innermost loops only, outermost loops only, and unconstrained.

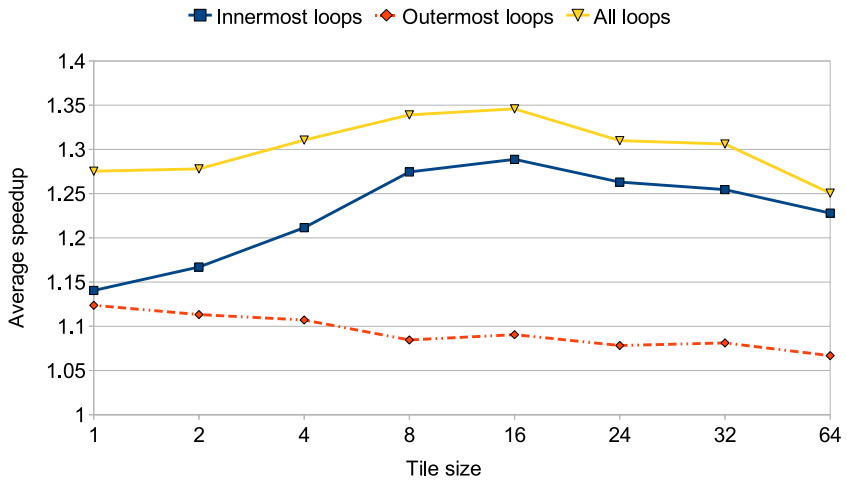

Figure 2.12: Speedup across various tile sizes under different loop selection policies.

The results in Figure 2.11 show performance under different loop selection policies with the tile size fixed at 16. But to see how the loop selection policy affects performance at various tile sizes, we simulate execution under the three policies at various tile sizes. The results are shown in Figure 2.12. Just as parallelizing innermost loops proves more profitable at the tile size of 16, it is more profitable at all other tile sizes as well (at least among the reasonable sizes we test). From this graph we also see that larger tile size more quickly results in worse performance for outermost loops. This is because outermost loops are typically larger: tiling does marginally less to amortize parallelization overhead (relative to smaller loops); and because dependencies are more likely, more execution is wasted in restarted transactions even when tile size is relatively small. Most importantly, from these results we observe that the unconstrained loop selection policy consistently yields better performance across all tile sizes.

2.7.2 TM Cache Granularity

All of the above results have been with the transactional memory system configured at word granularity instead of cache line granularity. Word granularity is more expensive, but yields performance results even on traditional transactional code [CMT00]. We explore how that granularity affects performance.

Figure 2.13 compares performance across all the benchmarks when transactional memory is implemented at word granularity verses cache line granularity. The average speedup across the NAS benchmarks drops from 1.36 with word granularity to 1.16 for line granularity; and the average speedup across the SPEC floating-point benchmarks drops from 1.34 to 1.21. While TM at word granularity increases data cache complexity and size, it dramatically increases parallel performance. TM at cache line granularity causes significant false sharing, both write-after-write (WAW) false sharing and read-after-write (RAW) false sharing. This increases the transaction restart frequency, which reduces parallel performance.

When TM is implemented at cache line granularity, small tile size results in more frequent false sharing. At higher tile sizes, some false sharing can be eliminated as parallel threads are more likely to operate on data in disjoint cache

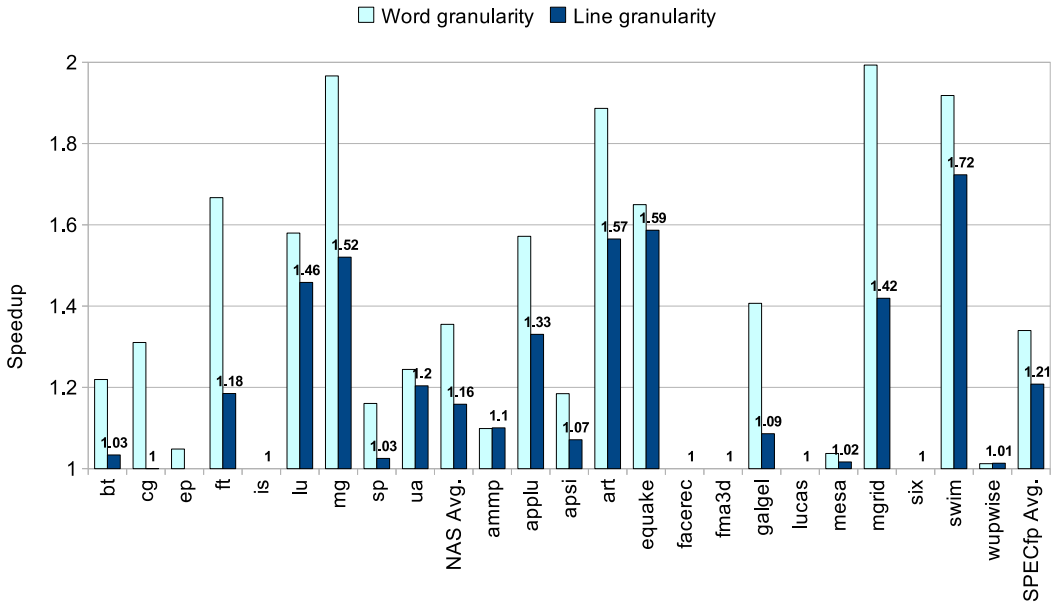

Figure 2.13: Speedups of NAS and SPEC FP benchmarks. The light bars show performance when TM is implemented at word granularity. The dark bars and the data labels show performance when TM is implemented at cache line granularity. The tile size in both cases is 16.

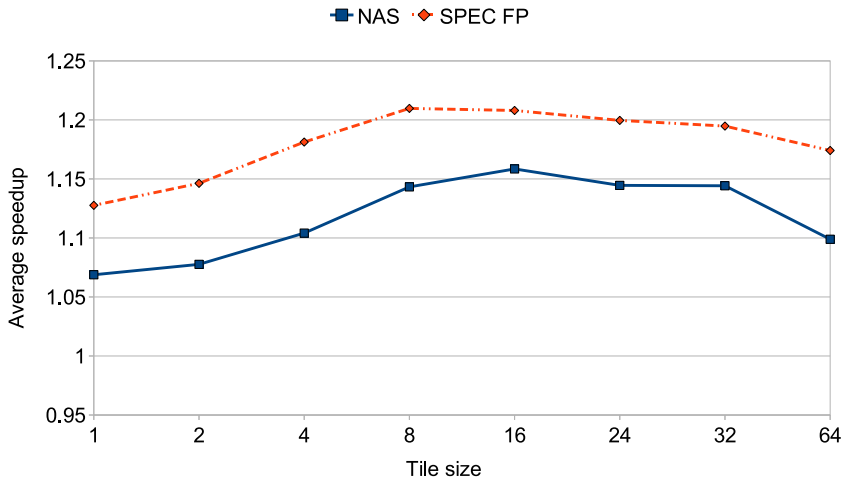

Figure 2.14: Speedup across various tile sizes. TM is implemented at cache line granularity.

lines. This effect is highlighted in Figure 2.14. Here performance is worse at lower tile sizes. Across all tile sizes the average performance of the NAS benchmarks is distinctly less than the average performance of the SPEC FP benchmarks. This is not something we observe at word granularity. This suggest that the NAS benchmarks are more prone to false sharing.

Despite the expected frequency of false sharing between iterations of legacy code compiled for single-threaded execution, we still see significant (albeit reduced) opportunity to find and exploit parallelism. This is because we assume a system that can identify poor-performing loops; thus, loops with frequent false-sharing-induced transaction restarts will be discarded.

2.8 Conclusion

In this chapter we present a runtime parallelization technique that leverages hardware transactional memory and the runtime flexibility and efficiency of dynamic optimization. It allows single-threaded legacy binaries to achieve performance improvements in the increasingly common context of multicore microarchitectures. Our parallelization technique makes use of transactional memory to provide optimistic concurrency and to make strong guarantees about correctness in code that a traditional compiler would have a hard time proving correct. Furthermore, parallelization is accomplished without assistance from the user or programmer and without access to the original source code.

We discuss some of the unique challenges posed by runtime parallelization and show how we address these challenges in our design. Our runtime parallelization yields 36% performance improvement across the NAS benchmarks and 34% performance improvement across the SPEC2000 floating-point benchmarks, utilizing two-core parallelism. We show that a loop selection policy that considers only loops at a particular nesting level (e.g., innermost loops only) fails to achieve the highest performance. We show that most applications are fairly intolerant of tile size (the number of iterations per transaction). They are more sensitive to the granularity of the underlying transactional memory system, achieving significant

gains with word granularity conflict detection.

Acknowledgments

The authors would like to thank the anonymous reviewers for many useful suggestions. They would also like to thank Leo Porter for his help with the transactional memory model. This work was supported in part by NSF grant CCF-0702349 and a gift from Intel.

This chapter contains material from "Runtime Parallelization of Legacy Code on a Transactional Memory System", by Matthew DeVuyst, Dean M. Tullsen, and Seon Wook Kim, which appears in *Proceedings of the 6th International Conference on High-Performance and Embedded Architectures and Compilers (HiPEAC)*. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Unbalanced Scheduling on a Multithreading Multiprocessor

3.1 Introduction

Applications with inherent thread-level parallelism may be parallelized into balanced homogeneous sets of parallel threads to efficiently run on homogeneous multicore platforms. However, many programs do not have natural TLP and cannot be parallelized. Fortunately, these applications can still benefit from multicore architectures if they can exploit heterogeneity to find the best core to run on—the core that will maximize performance or consume less energy. While this is especially true on heterogeneous multicore hardware, heterogeneity can still be found and exploited on homogeneous hardware—by taking advantage of unbalanced schedules. CMPs with SMT cores have the ability to simultaneously execute multiple applications on the same core. This creates the potential for unbalanced schedules, representing a form of heterogeneity, that may improve energy efficiency or performance (both per-thread and overall). These architectures present new opportunities and new challenges to achieving maximum efficiency via effective thread scheduling.

The challenges arise from the fact that given a set of applications and a set of multithreaded cores, the space of possible schedules of threads to cores can

be enormous—making it difficult to either predict or discover the best schedules. However, there are also tremendous opportunities. In this environment we have much more control over which threads, and how many, are coscheduled on cores. Even if we assume relatively balanced schedules (the same number of threads assigned to each core), we can select the groupings of threads assigned to each core so as to minimize negative interference between threads. In this chapter we show that we need not assume balanced schedules; and, in fact, the ability to create unbalanced schedules provides an important degree of freedom. This is an important result because conventional multiprocessor schedulers, applied to this architecture, will always seek to balance the number of threads on each core—we show that this is often the wrong decision.

Previous work [SPHC02a] has shown that resource demands vary significantly between applications, and even between phases of the same application. Recently, heterogeneous (or asymmetric) multiple-core architectures have been shown to be effective at exploiting this phenomenon by mapping each job to the core that most closely matches the resource demands of the application [KFJ+03, KTR+04]. We can exploit the same principles in a CMP of SMT processors without the burden of hardware heterogeneity. In this case, the heterogeneity comes from the fraction of core resources made available to each thread. For example, consider a CMP where one core is already running two threads and another core is idle. In this case, a new thread could either be scheduled on the first core, providing low marginal performance but possibly expending even less marginal power, or it could be scheduled on the idle core, providing high marginal performance but resulting in high marginal power. Scheduling on the already-loaded core may be best if the execution resource demands of the thread are low.

This chapter examines system-level thread scheduling policies for a multithreaded multicore architecture. Particular attention is paid to enabling performance and energy efficiency through unbalanced schedules. These schedules give the system the ability to cluster threads that have low execution demands and amortize the power cost of using a core.

Because the search space of possible schedules for such architectures is large,

we rely on schedulers that learn from experience and migrate to the best schedules. They do so either through directed sampling or by making small adjustments to the current schedule (which is assumed to be good). Our studies cover many different degrees of thread-level parallelism. We consider schedules that leave cores idle, even when there are more threads than cores. This is particularly useful when energy and power are primary concerns. When both performance and power are first-class concerns for the scheduler, the nature of the best schedules become difficult to predict; thus, it becomes critical to have scheduling policies that dynamically adapt to the particular workload's execution behavior, and discover the right strategy.

This chapter makes the following contributions. It studies, for the first time, a spectrum of scheduling policies for a multithreaded multicore architecture where both performance and energy are prime considerations. It shows that unbalanced schedules (uneven distribution of threads among the cores) often outperform balanced schedules—the best scheduling policies are those that consider both balanced and unbalanced schedules. We show that one can often get higher performance by clumping badly behaving threads together on the same core than by spreading them around. This is because such threads can interfere destructively with the otherwise high-performing threads. Running them with other low-performing threads is less likely to significantly impede those other threads. The benefits of unbalanced scheduling increase as the objective function puts more emphasis on power efficiency.

Additionally, we demonstrate that intelligent non-sampling-based scheduling policies can often outperform the policies that require sampling of the search space; this is significant because, given a moderate number of cores and a moderate number of threads, the search space for possible schedules can become large.

This chapter also extends symbiotic scheduling [ST00] to a CMP of SMTs. Symbiosis-based random scheduling heuristics that perform well for a SMT core also perform well for a CMP of SMT cores, but with smaller marginal gains. However, due to the much larger search space, we show that in this case there is even more gain to be had with more intelligent policies. Finally, we show that there are significant benefits to doing energy-aware scheduling. For 12 threads on

a four core four-way SMT processor, it can result in up to 7.4% savings in energy, 10.3% savings in energy-delay product, and 35% savings in power. Savings are even greater with fewer threads.

The rest of the chapter is organized as follows. Section 3.2 describes previous work related to thread scheduling and multithreading chip multiprocessors. Section 3.3 discusses the architecture that we evaluate—a CMP of SMT cores. In Section 3.4 we present our scheduling mechanisms and policies. We discuss our experimental methodology in Section 3.5 and present and analyze the results in Section 3.6. In Section 3.7 we summarize our findings.

3.2 Related Work

Multithreaded CMPs are becoming increasingly common. One of the first SMT CMP architectures, the POWER5 [CFS+04] (produced in 2004 by IBM), includes two SMT cores, supporting four hardware contexts. Its successor, the POWER6 microprocessor, released in 2007, also includes two two-way SMT cores. IBM's latest offering, the POWER7, boasts four, six, or eight cores, each four-way SMT. Sun's first multithreaded CMP, the UltraSPARC T1 (a.k.a. Niagara) [Sun05], released in 2005, contains four, six, or eight cores, each four-way SMT. The T1's successor, the UltraSPARC T2 (released in 2007) supports 64 thread contexts (eight cores, eight-way SMT). Sun's latest offering, the SPARC T3 supports 128 thread contexts (16 cores, eight-way SMT). Intel's Nahalem microarchitecture [Sin08] (introduced in 2008) includes processors with up to eight cores and two-way SMT. With an abundance of multithreaded CMPs on the market today and the trend of increasing hardware parallelism, thread scheduling on such architectures is increasingly important.

Scheduling for SMT Processors

There is a long history of research on scheduling for multithreaded single-core processors.

Parekh et al. [PEL00] propose a number of *thread-sensitive* scheduling al-

gorithms for SMT processors. These algorithms use measurements of how each thread utilizes different parts of the core. They find that the most effective feedback metric is IPC. High IPC floating-point threads are coscheduled with high IPC integer threads.

Snavely and Tullsen [ST00] introduce a sampling-based scheduler that samples job coschedules and then predicts coschedules with high *symbiosis*. Symbiotic jobs are ones that run well together—that use complementary core resources instead of aggressively competing for the same shared resources at the same time. Finding symbiotic coschedules on an SMT is important because the level of resources sharing is much higher than in multiprocessors. Snavely et al. [STV02] extend their scheduler to support priorities and show up to 40% improvement in throughput while still meeting priority goals. While the kind of coscheduling in their work is temporal, the coscheduling in our research is spatial—coscheduling symbiotic jobs on different cores. Also, they do not consider power or energy.

Settle et al. [SKJC04] add architectural support to expose direct measures of shared cache utilization to the OS scheduler. By not relying on sampling, their scheduler can adapt more quickly to changing program behavior. However, by focusing only on cache interference among coscheduled threads, they miss some of the other points of resource contention in SMT systems (e.g., register file, hardware buffers).

Scheduling for CMPs

The degree of resource sharing among threads on different cores (usually just one or two of the furthest levels of cache) is significantly less than on a single SMT core. Hence, the cost of a bad coscheduling decision on a CMP is less than the cost on an SMT core. Nevertheless, finding optimal schedules for CMPs with single-thread cores is very difficult. Jiang et al. [JSCT08] prove that the problem is NP-compete. They present a sequence of scalable approximation algorithms that find near-optimal schedules.

Scheduling becomes even more difficult when the cores are asymmetric (heterogeneous). Kumar et al. [KFJ+03] and Ghiasi and Grunwald [GG03] in-

troduce single-ISA heterogeneous multicore architectures and demonstrate how intelligently scheduling single threads can result in significant power and energy savings or performance gains. Kumar et al. [KTR+04] go beyond finding the best core for a single thread and consider finding good schedules for workloads of size up to the number of available hardware contexts. They employ a sampling strategy to find good schedules as programs enter new phases of execution. In their hardware model, some cores are multithreaded, and they explore simple heuristics to guide the scheduler to good thread assignments.

Li et al. [LBKH07, LBK+10] propose a scheduler, called AMPS, for SMP and NUMA asymmetric architectures. They implement AMPS in the Linux kernel and model asymmetric multicore architectures by modulating clock speed and changing L2 size and execution width. AMPS has three distinguishing features: (1) *asymmetry-aware load balancing* assigns load to each core proportional to its computing power, (2) *faster-core-first* scheduling assigns threads to under-utilized powerful cores first, and (3) *NUMA-aware migration* predicts thread migration overhead and manages migration on NUMA architectures. Like the scheduling heuristics of Kumar et al. [KTR+04], AMPS is focused on performance only.

In this chapter, we consider scheduling for objectives that are a composition of both power and performance. We consider more powerful heuristics that are better suited to SMT CMP architectures.

Scheduling for Multithreaded CMPs

El-Moursy et al. [EMGAD06] model scheduling on a simple processor consisting of two cores, with two SMT contexts each. They observe that the number of potential schedules on such an architecture and the dynamic nature of program phase behavior makes finding good schedules very difficult. They experiment with several hardware metrics (including register file contention, functional unit contention, and L2 cache contention) to predict what threads run well together. They find that metrics of ready and in-flight instructions prove the most effective in aiding the scheduler to make the right decisions. Their only objective function is performance.

Improving Cache Performance

Fedorova et al. [FSSN05] examine scheduling for L2 cache miss rate on an SMT CMP architecture. They introduce an L2-conscious scheduling algorithm based on balance-set scheduling. They assume high thread-level parallelism, and do not consider unbalanced schedules. In this research, we focus on direct measures (i.e., performance, power, energy) rather than indirect—thus, if L2 miss rate is the dominant factor, we will migrate to schedules that minimize it (but perhaps more slowly). If there are other important factors, we will find better schedules.

Avoiding Thermal Emergencies

Powell et al. [GPV04] seek to alleviate the power density problem, a condition found in many modern processors and potentially aggravated by an SMT CMP architecture. Their work leverages the ability to schedule on two levels: inter-core and intra-core. Because their goal is very different, the proposed scheduling solution, Heat-and-Run, is not effective for reaching our goals. Heat-and-Run moves jobs off of a core (to a cooler one) before the core can exceed thermal thresholds. In a sense, they employ unbalanced schedules to favor cool cores, and let hot cores cool down.

3.3 Architecture

We focus on a single hardware architecture but evaluate it under different constraints and different levels of thread parallelism (different loads). This architecture is large enough to make scheduling a complex problem; and we believe the principles exposed by our results will scale to larger configurations.

The architecture is a chip multiprocessor consisting of four homogeneous simultaneous multithreaded [TEL95] cores implemented in the $0.1\mu m$ process with shared L2 and L3 caches. Each core has its own L1 data cache and L1 instruction cache. Because threads can migrate between cores, they will exercise the cache coherence policy to correctly handle dirty data in the L1 data caches. When a thread migrates to another core and requests data that was dirty in the L1 data

cache of the core it was previously running on, the dirty data will be written to both the L1 cache of the core on which the thread is now running and the shared L2 cache. The implementation is assumed to be at 2.1 GHz and latencies are determined accordingly.

There are four contexts per SMT core, for a total of 16 contexts in the system. Each SMT core is out-of-order. There exist two dimensions of thread-level parallelism (TLP) in our architecture as there are multiple cores, each with multiple contexts. In the first dimension of TLP, threads coscheduled in different contexts on the same core share many core resources including a register file, hardware queues, and a set of functional units. In the second degree of TLP, threads scheduled on different cores share fewer resource (such as the more distant levels of the memory hierarchy).

In this work, we assume that unused cores are completely powered down, rather than left idle. Thus, unused cores suffer no static leakage or dynamic switching power. This does, however, introduce a latency for powering a core on or off. In [KFJ+03], it is estimated that a given processor core can be powered on in approximately one thousand cycles of the 2.1 GHz clock. They assume that when a processor core is powered down, the phase-lock loop that generates the clock for the core is not powered down. Rather, the same phase-lock loop generates the clock for all cores. Consequently, the power-up time of a core is determined by the time required for the power buses to charge and stabilize. However, in this work we did not want to assume constraints on power supply or PLL design, so we perform all experiments assuming that it takes 30 microseconds to turn on or off a core. We estimate this time based on reported data on PLL stabilization times and system overheads. Note that this is over 60 times more conservative than was assumed in some prior work [KFJ+03], and it allows plenty of time for dirty data to be flushed from the caches, as well. We also assume, conservatively, that a core keeps dissipating idle power at a steady rate until it is completely powered off and also during the entire power-on procedure. We do not consider dynamic voltage scaling (DVS) in this work.

Table 3.1 contains more details of the architecture.

Table 3.1: Architecture Detail

Cores	4	TLB miss penalty	790 cycles
Contexts per core	4	I cache size	32k
Reorder Buffer entries	256	I cache miss penalty	8 cycles
Active List entries	512	I cache associativity	4
Total fetch width	4	D cache size	32k
Integer registers per core	100	D cache miss penalty	8 cycles
FP registers per core	100	D cache associativity	4
shared L2 cache size	2 MB	shared L3 cache size	4 MB
L2 miss penalty	40 cycles	L3 miss penalty	315 cycles
L2 access time	12 cycles	L3 cache access time	35 cycles

3.4 Scheduling Policies

We assume an operating system level thread scheduler that makes global (CMP-wide) scheduling decisions. Thus, scheduling decisions must be made at a very coarse granularity, and the cost of moving a thread is very small relative to the typical interval between moves. The processor typically makes scheduling decisions based on sampled data of processor power and performance. Processor power and performance can be estimated by reading counter registers that are already found in most modern processors. New samples are collected and new scheduling decisions can be made as often as every operating system timer interrupt (although, in general, we strive to change schedules much less often than the timer interrupt) or when the mix of jobs changes.

We assume all jobs have equal priority. Differing priorities could significantly increase the utility of unbalanced schedules, thus increasing the importance of scheduling algorithms that consider such schedules. However, the correct metrics to evaluate the goodness of our schedules are less clear in the presence of priorities, so we do not demonstrate that advantage in this work.

The scheduling policies we evaluate are described below. They come in two broad categories: *sampling-based policies* and *electron policies*. The electron polices are so called because the state of a core may cause it to either try to push a job away or to attract new jobs.

3.4.1 Sampling-based Policies

The sampling-based policies work in a series of alternating temporal phases: a sampling phase and a steady phase. During the sampling phase, a number of different schedules are tried; at the end of the sampling phase the sample schedule with the best *metric value* is chosen as the schedule to be in effect throughout the steady phase. The *metric value* depends on what we wish to optimize for, whether it be power, energy, energy-delay product, performance, or something else. In addition to the sample schedules, the schedule in place during the last steady phase is also a candidate for selection—this ensures that in the ideal (but somewhat rare) case with no noise or phase behavior, we will always move toward better schedules. In all cases, we create 10 sample schedules, which we find creates a good balance between the desire to maximize our chances of finding a good schedule and to minimize the ratio of the duration of the sample phase to the duration of the steady phase. Each sample runs for two time slices, and the measurement takes place on the second, to eliminate cold-start effects. The steady phase then runs 10 times as long as the sample phase.

While the creation of schedules is geared towards intelligently navigating through the permutation space, selecting the best schedule involves using the right metric to characterize the goodness of a schedule. The sampling policies, then, are generic and can be specialized to different objective functions by changing the evaluation metric. The evaluation metrics used to choose the best sampled schedule are described in Section 3.4.3.

We consider the following sampling-based scheduling policies:

Balanced Symbiosis It has been shown previously that some groups of applications run well together, while others result in destructive interference [TB01], causing individual applications to slow down. This property can be used to enhance system performance by scheduling "friendly" threads together on one core. Threads are randomly assigned to contexts for each sample schedule. The only constraint, in this case, is that the number of threads scheduled on each core is the same, or within one if the threads don't divide evenly onto cores. At the end

of the sampling phase, whatever decision metric has been chosen will be used as the criterion for selecting a schedule for the steady phase. This policy is closest to the symbiotic job schedulers described by Snavely and Tullsen [ST00] for a single SMT core (those schedules were "balanced" in the sense that they always used all thread contexts). Because this scheme only considers balanced schedules, most schedules considered are reasonable ones—but potentially better schedules that are not balanced will never be considered.

Symbiosis This policy is similar to the prior one, except that it does not constrain the threads to be assigned evenly among the cores. As a result, it has the potential to find better schedules than the first policy can find; but it also has the potential to sample a larger number of clearly bad schedules.

Balanced Random This policy chooses a random, but balanced, schedule with no sampling. This is the closest approximation to what a conventional load-balancing multiprocessor scheduler would do, and is the baseline in many of our graphed results.

Prefer Last This is a class of policies that assume that the configuration we are running now has merit, and the next configuration will be similar. In this case, sampling is biased towards a similar configuration with only a fraction (30%) of the schedules deviating from that. The different forms of *prefer last* are described below—they differ in how we define "similar" schedules. With these policies, we can apply an intelligent bias to the sampling (e.g., use all cores, use few cores)—but the bias is not hard-coded in the scheduler; the bias is derived from recent history. *Prefer last* policies are introduced in [KTR+04]; however, we look at many more flavors and apply them in a different context.

Prefer Last – Numbers A new schedule is similar to the original schedule if it runs the same number of threads on each core as the original did. However, the particular assignment of threads is done randomly. This policy seeks to retain the same level of schedule imbalance, but does not strive to favor or coschedule the

same threads.

Prefer Last – Swap A similar schedule is created by choosing two threads (on different cores) from the original schedule to swap places. This policy evolves slowly both in the number of threads assigned to cores and the composition of threads coscheduled.

Prefer Last – Move In this variant, a similar schedule is created by randomly choosing one thread and moving it to a randomly-selected empty context on another core and leaving all other threads in place. This policy tends to preserve the sets of threads coscheduled, but allows the distribution of the load (in number of threads) to evolve more quickly. Note that this policy does not work in this exact form for very high system loads (i.e., when all the contexts are saturated).

All these policies are evaluated assuming that any unused cores are power-gated. When a sampling phase begins, all the samples to be tried out in that sampling phase are created and then ordered by the number of cores they utilize. The schedules that use the same number of cores as were used in the last steady phase are tried first. All the schedules that use a different number of cores are grouped together by the number of cores they use and are ordered such that all the sample schedules using the same number of cores are tried consecutively. This minimizes the number of power-gating changes that are made.

3.4.2 Electron Policies

The policies in the previous section rely on sampling for intelligent scheduling. However, such policies become less effective as the search space expands.

Rather than sampling, the electron policies rely on more explicit evidence that a particular core is over-scheduled, under-scheduled, or just poorly scheduled. Cores will attract threads that fill a void and repel threads when contention is high. Threads will move around each interval to create a better fit. The schedule naturally adapts as threads enter new phases of execution.

Following is the description of electron policies customized for each metric that we study.

Electron – Performance This policy assumes that utilization of a core's resources has a correlation with performance. The IPC of each core in the previous period is calculated. The core with the highest aggregate IPC repels one thread, and the core with the lowest IPC attracts a thread. If the latter has a free context, the thread is transferred. If the condition is not met, we do not change the schedule. Ninety percent of the time the selection of the thread to repel away from the high IPC core is the highest IPC thread on that core, and ten percent of the time a random thread from that core is selected. The thread selection policy is based on the assumption that the highest IPC thread has the most aggressive resource requirements and hence needs to find a core with the least current utilization. Occasional random selection guards against the same thread infinitely hopping from core to core.

Electron – Energy The objective here is to minimize the overall energy consumption of the processor. The energy of each core in the previous period is calculated. The non-idle core with the lowest energy repels one randomly-chosen thread, and the core with the highest energy attracts a thread. If the former was not idle and if the latter has a free context, then the thread is transferred; otherwise the schedule does not change. Over time, this policy tends to cluster threads so that more cores are left idle, then distributes the jobs efficiently among those cores.

Electron – EDP This policy tries to minimize overall energy-delay product of execution by identifying cores under-performing on this metric. The energy-delay product of each core in the previous period is calculated. The core with the highest EDP attracts a thread, which is supplied from the core with the lowest EDP—the thread to move is chosen randomly. If the latter was not idle and if the former has a free context, the thread is transferred; otherwise the schedule does not change. If a core was idle (meaning it consumed no power but completed no instructions) its

EDP is considered to be infinite. This assumption discourages idling of cores—we find that leaving a core idle (when EDP is the targeted metric) is usually inefficient.

Note that making locally good decisions for a metric like EDP does not guarantee making globally good decisions [SKTC05].

For the electron policies, the duration of a period is moderately long (i.e., a schedule change can occur at most once every four operating system time-slices in our experiments). If a new schedule results in a core being left idle, that core is powered down immediately. For all the policies, at every scheduling change, if the new schedule calls for cores to be powered on, the required cores are powered on before the new schedule takes effect. Also, if the new schedule calls for cores to be either powered on or powered off, the scheduler does not sample any performance or power counters during the transition, so as not to skew the statistics associated with the new schedule.

Note that electron schedulers run the risk of continuing to alternate among two schedules. To help avoid such situations, history information has been incorporated into the scheduling policy decisions. A history is maintained of the last 10 schedules run (duplicate schedules included) and the associated performance and power. If the schedule that the electron policy proposes is found in the history, then the scheduler knows about how well it will perform. In such cases, it will select the "best" schedule among all the schedules in the history, where the "best" schedule is the one found to be most efficient with regard to the desired decision metric. If the proposed schedule is not found in the history, its performance is unknown, so it will be run. Because the history length is finite and contains previously used schedules regardless of their uniqueness among the other history entries, the electron scheduling policies will be able to adapt to workload phase changes. Stale performance data is evicted from the history and previously applied schedules are eventually applied again.

3.4.3 Decision Metrics

This section describes the metrics used to select from among the sampled schedules discussed in Section 3.4.1. The objective function during scheduling may

be static for a given environment and a given market segment. In other cases, it may change dynamically as the processor changes power conditions (e.g., plugged vs. unplugged, full battery vs. low battery, thermal emergencies), as applications switch (e.g., low priority vs. high priority jobs), or even within an application (e.g., a real-time application is behind or ahead of schedule).

We consider four system-level objective functions in this chapter: performance, power, energy, and energy-delay product; and we tune the evaluation metric for those objective functions in the following ways.

Performance The metric for performance is *calculated weighted speedup* (CWS). This is derived from the weighted speedup metric proposed by Snavely and Tullsen [ST00], and used in this chapter to evaluate simulated performance (described in Section 3.5). However, it cannot be applied in the same way at runtime without oracle information—this is because it depends on knowing how the program would run on a baseline configuration (e.g., single-threaded). Thus, we define each application's "average performance", for the purpose of calculating CWS, as the average of its performance over a number of sampled configurations. Thus, for the runtime scheduler, one thread's contribution to the total calculated weighted speedup is its IPC for that sample, divided by its average IPC for all samples. We find this to provide better performance than using IPC as the choice metric. With IPC, causing one thread's throughput to drop from four to two is twice as bad as causing another thread to drop from two to one. With CWS, they are considered equally bad—this represents a more system-level view of performance which says that a 2X slowdown in any application is considered bad, regardless of the raw IPC.

Energy-Delay Product The energy-delay product (EDP) of the processor, which is $PowerPerCycle/IPC^2$, is computed for each of the sampled schedules. The schedule with the lowest EDP is considered the best.

Energy The energy of the processor (measured in watts per instruction), is $PowerPerCycle/IPC$ $((watts/cycle)/(instructions/cycle) = watts/instruction)$.

It is computed for each of the sampled schedules—the sample with the lowest energy is considered the best. Performance (IPC) is still a factor in the energy equation, because faster execution consumes power over a shorter time-span.

Power The total power is computed at each sampling schedule. The total power is the sum of the power of all the cores plus the power of the shared structures (L2 and L3 caches). The schedule with the lowest power is considered the best.

3.5 Experimental Methodology

In this section, we discuss the various methodological details of the evaluation framework and scheduling mechanisms for this chapter.

3.5.1 Scheduling Parameters

Our thread scheduler is assumed to be a part of the operating system; it makes scheduling decisions based on sampled data of processor power and performance. New samples are collected and new scheduling decisions can be made at every operating system time-slice interval. We assume an operating system time-slice of a quarter million cycles. This time-slice is artificially short to keep our simulations from taking too long. It allows us to model a larger number of sample/steady intervals per simulation, and is still long enough to capture interesting application phases for most of our benchmarks. This enables our scheduling policies to be evaluated and compared based on how quickly and accurately they can adjust to the changing workload behavior. The time-slice interval durations we use are significantly longer than any lingering cold-start artifacts of the simulation methodology.

3.5.2 Workload Construction

We select twelve benchmarks from the SPEC 2000 benchmark suite to construct workloads for the evaluation. Sub-setting is done such that the fraction of compute and memory bound benchmarks is the same as that in the *entire* SPEC

suite—so the subset is representative of the entire suite. Each benchmark is fast-forwarded for two billion instructions before detailed simulation. Table 3.2 contains a list of the twelve benchmarks. All simulations use the reference data sets.

We perform all our evaluations for various values of available thread-level parallelism for multiprogramming workloads. For each level of thread-level parallelism, we construct and use eight workloads by randomly selecting eight different subsets of the 12 benchmarks. These groups are formed such that the contribution of a benchmark to the result remains the same across different TLPs, similar to the sliding window methodology typically used in SMT research [TEL95]. For the 12-thread experiments and higher, at most only one group could be constructed if we don't allow duplicate threads. Thus, multiple instances of randomly-selected application(s) are run in a single group. Table 3.3 lists the workloads used for our study. In all the results reported in this chapter, the results are obtained by averaging the statistics across the eight groups.

3.5.3 Simulation Methodology

All our simulations are done using a chip-multithreaded multiprocessor derivative of SMTSIM [Tul96b]. The simulator supports the MESI coherence protocol and executes statically linked Alpha binaries. Appropriate modifications are made to simulate the effect of OS-level scheduling as well as the availability of hardware counters. To gather power statistics we integrate a modified version of Wattch [BTM00] into our simulator. We modify Wattch to collect, calculate, and report power statistics on a multicore architecture with a shared L2 cache. We make use of Wattch's conditional clocking power model (labeled `cc3` in Wattch source code). We also introduce core-level power-gating into the model. The modeling details for power-gating are discussed in Section 3.3.

3.6 Analysis and Results

This section presents the effectiveness of the scheduling policies that adapt to varied program behavior and consider both balanced and unbalanced schedules.

Table 3.2: *Benchmarks*: Each benchmark is labeled with a code (used in Table 3.3) and a brief description

Benchmark	Code	Description
gap	0	Interpreter
fma3d	1	Crash simulator
mesa	2	3D graphics
equake	3	Wave simulator
crafty	4	Chess game
wupwise	5	Quantum chromodynamics
mgrid	6	Multi-grid solver
gzip	7	Compression
gcc	8	Compiler
apsi	9	Meteorology
swim	A	Shallow water modeling
ammp	B	Chemistry

Table 3.3: *Workloads*: The first part of each pair is the workload label. The second part of each pair encodes the benchmarks that form the workload—each digit in this number is the code of a particular benchmark. There are workloads consisting of 4, 6, 8, 12, and 16 threads.

4a	8165	6a	8165B0	8a	8165B072
4b	6359	6b	635924	8b	6359240B
4c	A960	6c	A96048	8c	A9604851
4d	5879	6d	5879B1	8d	5879B143
4e	A23B	6e	A23B79	8e	A23B7968
4f	4230	6f	423076	8f	423076A9
4g	47A8	6g	47A8A1	8g	47A8A10B
4h	1B20	6h	1B2035	8h	1B20354A

12a	8165B07284A3	16a	8165B07284A36359
12b	6359240B38A1	16b	6359240B38A1A960
12c	A96048516723	16c	A960485167235879
12d	5879B1435062	16d	5879B1435062A23B
12e	A23B79689514	16e	A23B796895144230
12f	423076A97B51	16f	423076A97B5147A8
12g	47A8A10B0962	16g	47A8A10B09621B20
12h	1B20354AB879	16h	1B20354AB8798165

We pay particular attention to the case where both performance and energy are important and provide more detailed results for that case. We also examine the various policies for objective functions specific to energy, power, and performance.

3.6.1 Scheduling for Both Energy and Performance

Scheduling for both energy and performance at the same time presents an interesting challenge for a CMP of SMT cores. The marginal performance achieved by using an additional core in a CMP of SMTs is typically higher than the marginal performance improvement from using an additional SMT context on the same core. Also, the marginal performance improvement from an SMT context continues to decrease as the number of threads increases [TEE+96]. So, it is often better to schedule as few applications on each SMT core as possible if scheduling only for performance—threads spread out across cores. On the other hand, the converse is true for energy. That is, energy efficiency increases with the number of contexts in operation [STC00], so we tend to aggregate threads when scheduling for energy. Figure 3.1 shows how energy and performance vary with the number of contexts for our processor model. We see that, on average, marginal performance drops off as we add threads, and is typically much less than the performance of using a second core. Conversely, energy efficiency is maximized as we add threads to a single core. Scheduling for a CMP of SMTs such that both energy and performance are optimized, then, requires a careful balance between these two competing objectives.

We perform all our evaluations in this section using the energy-delay product (EDP) metric. EDP recognizes the importance of both energy and performance and is used widely as an important objective function for desktop as well as server processors.

Unbalanced Scheduling Schedulers for traditional multiprocessors seek to evenly distribute the system load over the available processing contexts. Such schedulers constrain the schedules to be balanced, limiting their flexibility to optimize for multiple competing objectives at the same time.

If our scheduler allows unbalanced scheduling, we have the opportunity to

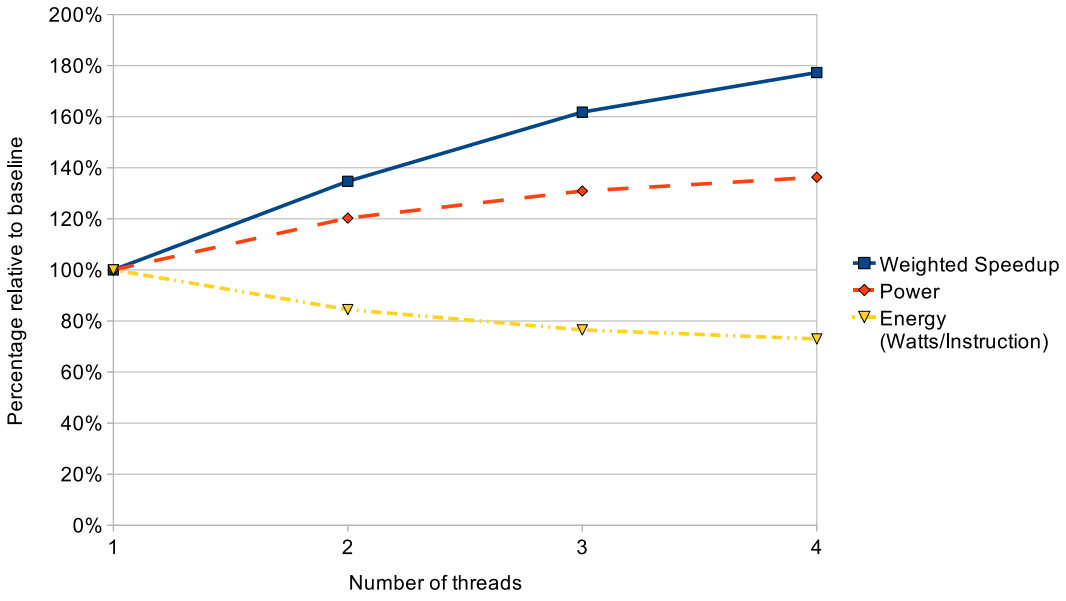

Figure 3.1: Average marginal utility and cost of using each SMT context on a single core.

meet both of these objectives—we can aggregate jobs that have low execution resource demands for energy efficiency, while still giving more resources (e.g., on other cores) to those jobs that demand them for performance.

Figure 3.2 compares an optimal static scheduling policy (*Static Ideal*) that allows unbalanced scheduling against the best static scheduling policies that are constrained to be balanced. *Static Balanced* ensures that each core runs the same number of threads (or within one if the number of threads is not evenly divisible by the number of cores), and hence, is similar to the traditional load-balancing schedulers. *Static Cluster Balanced* ensures that only as many cores as necessary to run a given number of threads are kept on and the rest are power-gated; among the active cores, each core runs the same number of threads. This policy minimizes the system energy consumption at the expense of performance. The results are shown for different levels of thread-level parallelism. Thus, if we have six threads, *Static Balanced* will only consider schedules of threads to the four cores like 2,2,1,1; and *Static Cluster Balanced* will only consider schedules like 3,3,0,0.

Figure 3.2 shows that there is a significant advantage to doing unbalanced

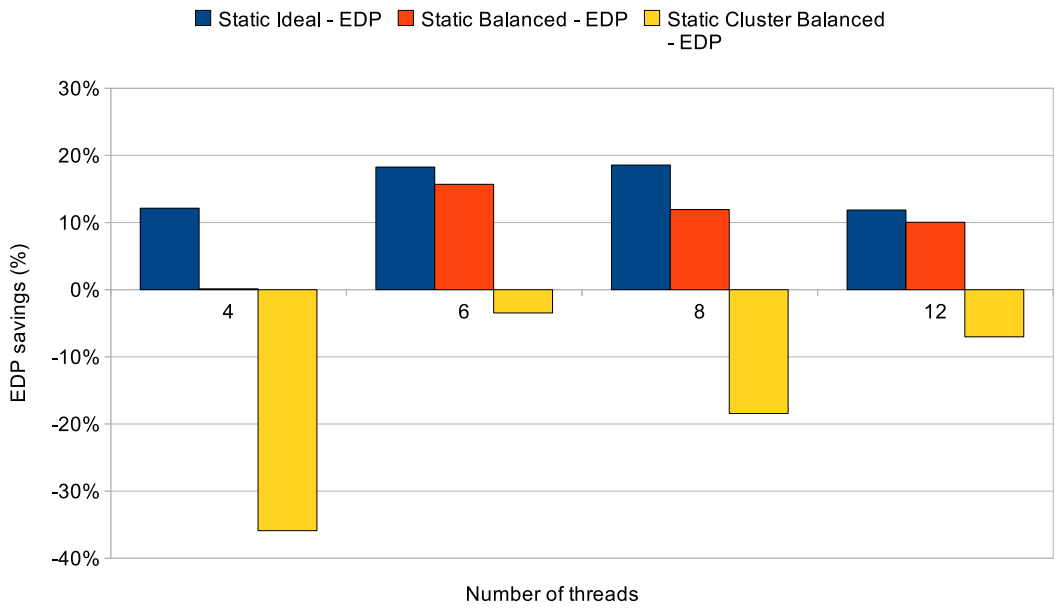

Figure 3.2: The effectiveness of unbalanced and balanced static scheduling policies in reducing energy-delay product. Results are presented as fraction of EDP savings relative to *Balanced Random*.

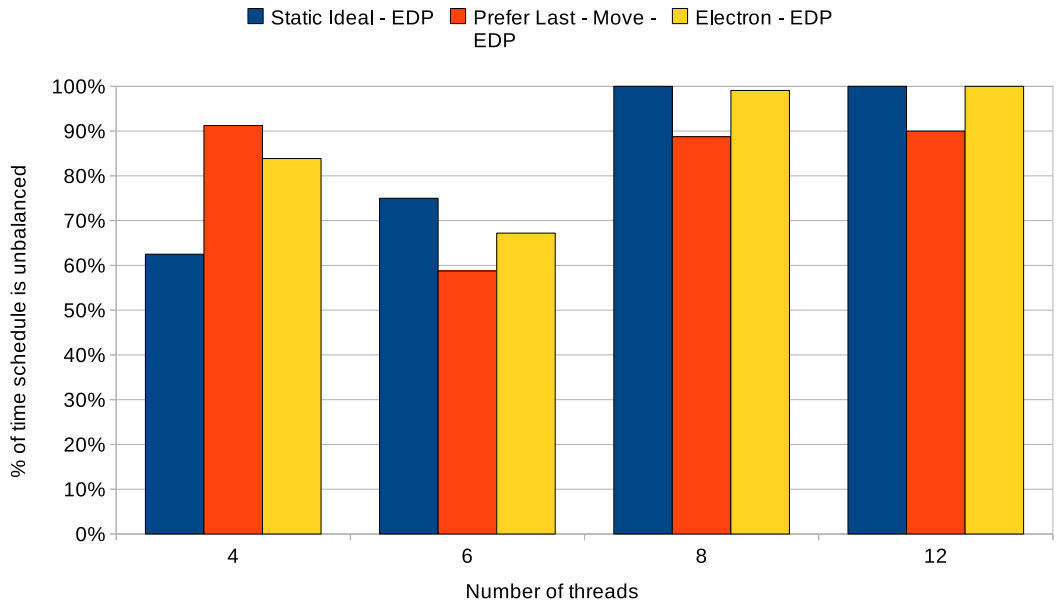

Figure 3.3: Extent of imbalance for the best schedules found by static policies targeting EDP.

scheduling—*Static Ideal* results in consistently higher EDP savings than the best balanced scheduling policy (*Static Balanced*). Savings are 12% higher for four threads and 6.6% higher for eight. As cores become more heavily saturated with threads, the flexibility for doing intelligent unbalanced scheduling decreases and the relative benefits decrease.

While these results are averaged over eight workloads, we observe that *Static Ideal* often results in unbalanced schedules (see Figure 3.3, black bar—the other bars are discussed later). For example, for four threads, five out of eight workloads are scheduled in an unbalanced manner in the ideal case. For six threads, this number is six out of eight. All the best static schedules for eight and 12-threaded workloads are unbalanced. We also observe that most of the schedules are unbalanced even for the best dynamic scheduling policies discussed in the following sections.

The advantages due to unbalanced scheduling depend on the characteristics of the workload. Benchmarks *gcc* and *gzip* are averse to running with other applications due to high instruction cache working set sizes and high core utilization, respectively. Balanced scheduling policies force these applications to be coscheduled with some other thread on the same core, resulting in a significant performance hit. However, *Static Ideal* allows these applications to be on a core by themselves, resulting in high overall efficiency. On the other hand, we find that *ammp*, *swim*, and *apsi* are often coscheduled, as they have lower inherent ILP and see minimal destructive cache interference. Workload 8e (see Table 3.3) contains both *gcc* and *gzip*, and the best unbalanced schedule has 14% lower EDP than the best balanced schedule.

The graph also makes a case for energy-aware scheduling. *Static Ideal* results in more than 12.1% EDP savings for four threads and 18.5% savings for eight threads. In fact, we observe that the policy can result in savings of 8.7% even for 16 threads (not shown in the graph) over a naive schedule that only seeks to balance the load.

The static results are ideal, identified by exhaustive search. The next section presents realistic dynamic scheduling policies.

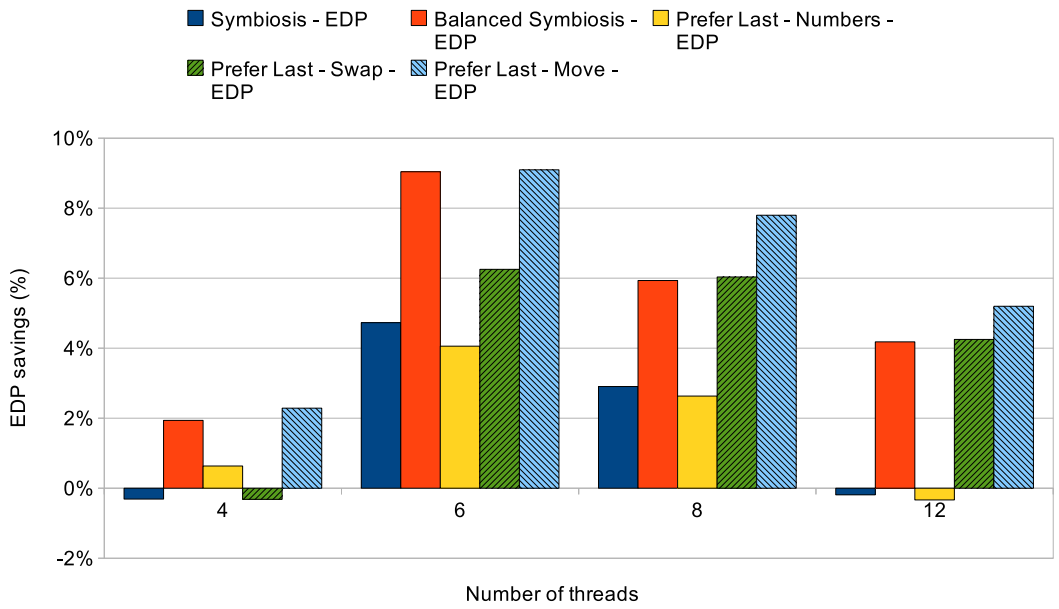

Figure 3.4: Energy-delay product for sampling-based scheduling policies.

Exploring the Search Space Through Directed Sampling Sampling-based scheduling policies try to adapt to the changing workload behavior by sub-setting the search space and then making the best choice of schedule from among the reduced space. The effectiveness of a scheduling policy is hence primarily determined by how effectively it does the sub-setting. Figure 3.4 compares the various sampling-based scheduling policies. Results are shown for both symbiosis-based policies as well as *Prefer Last* policies. The results are shown for various levels of thread-level parallelism and with *Balanced Random* policy as the baseline.

The graph leads to several interesting observations. First, there are again significant benefits to doing energy-aware dynamic scheduling. The best sampling-based policy (*Prefer Last – Move*) results in 2.3% EDP savings for four threads, 7.8% savings for eight threads, and 8.3% savings for 16 threads (16 thread case not shown in graph). These benefits are achieved through unbalanced scheduling of threads to cores. For example, 91% of the schedules chosen by the *Prefer Last – Move* are unbalanced for four threads. The percentage of unbalanced schedules is 59%, 89%, and 90% respectively for six, eight, and 12 threads. Figure 3.3 shows

the fraction of unbalanced schedules for other policies as well.

The results show that symbiosis-based scheduling policies that perform well for an SMT core [ST00] also perform well for a CMP of SMT cores. The best symbiosis-based policy (*Balanced Symbiosis*) results in 1.9% EDP savings for four threads and 5.9% savings for eight threads. The continued benefits due to symbiosis-based policies can be explained by the need to coschedule "friendly" threads together even on a chip multithreaded processor. A more surprising result is *Balanced Symbiosis* outperforming *Symbiosis*. Although *Symbiosis* has the freedom to try out both balanced as well as unbalanced schedules, it also has a greater likelihood of sampling bad schedules for a given number of samples. *Balanced Symbiosis*, on the other hand, is conservative and only samples reasonably good (balanced) schedules, converging on better schedules more quickly. This is particularly true with six threads, where even the balanced scheduler is allowed (forced) to sample moderately unbalanced schedules.

Another significant observation is that the policies that learn from the makeup of the current configuration (*Prefer Last*) can result in high overall system efficiency. In fact, the best *Prefer Last* scheduling policy (*Prefer Last – Move*) outperforms the best symbiosis-based policy (31% higher savings for workloads of eight threads). This is due to a more targeted sub-setting of the search space by the *Prefer Last* policies. *Prefer Last – Numbers* emerges as the weakest sampling-based policy because it does not preserve coschedule relationships and changes the number of threads per core slowly. *Prefer Last – Swap* preserves coschedule relationships, but also changes the distribution (in number of threads) slowly. *Prefer Last – Move* outperforms both the above policies as it not only preserves the sets of threads coscheduled, but also allows the distribution of the load to evolve more quickly.

Non-sampling Strategies The previous policies rely on sampling for intelligent scheduling. However, such policies get increasingly less effective as the assignment space gets larger; for a given machine, the size of the assignment space increases with the number of threads that need to be scheduled. The sampling strategies also experience an overhead, as the sampling intervals will have lower performance

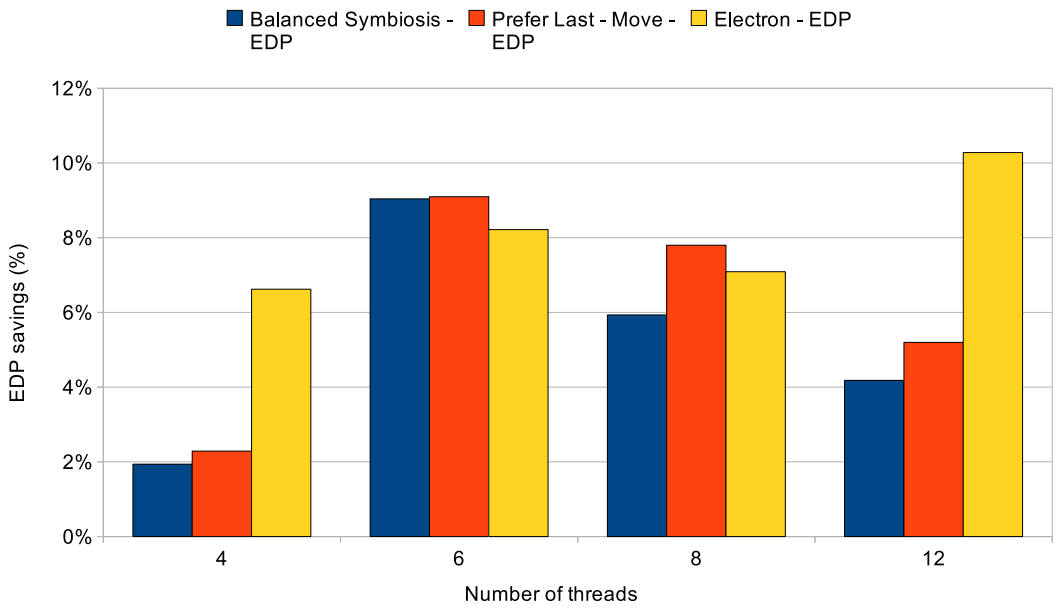

Figure 3.5: Effectiveness of the non-sampling electron policy, compared to two sampling policies.

than the steady intervals.

Figure 3.5 shows the results for the *Electron EDP* scheduling policy. The electron policies rely on more explicit evidence that a particular core is over-scheduled, under-scheduled, or just poorly scheduled. This is especially useful when the assignment space is too large to sample effectively. In fact, the electron policy outperforms all the sampling-based policies when thread-level parallelism is high. EDP savings are twice that of *Prefer Last – Move* and 2.4 times *Balanced Symbiosis* for 12 threads. While the sampling strategies struggle with the size of the search space, the fact that the electron policies incur no sampling overhead allows those policies to adapt much more often and navigate the large schedule space more effectively.

We also observe that the best schedules for *Electron EDP* are unbalanced. For 12 threads, for example, 100% of the schedules are unbalanced. 83% and 99% of the schedules are unbalanced for four and eight threads respectively. This again confirms the usefulness of providing more flexibility to the scheduler.

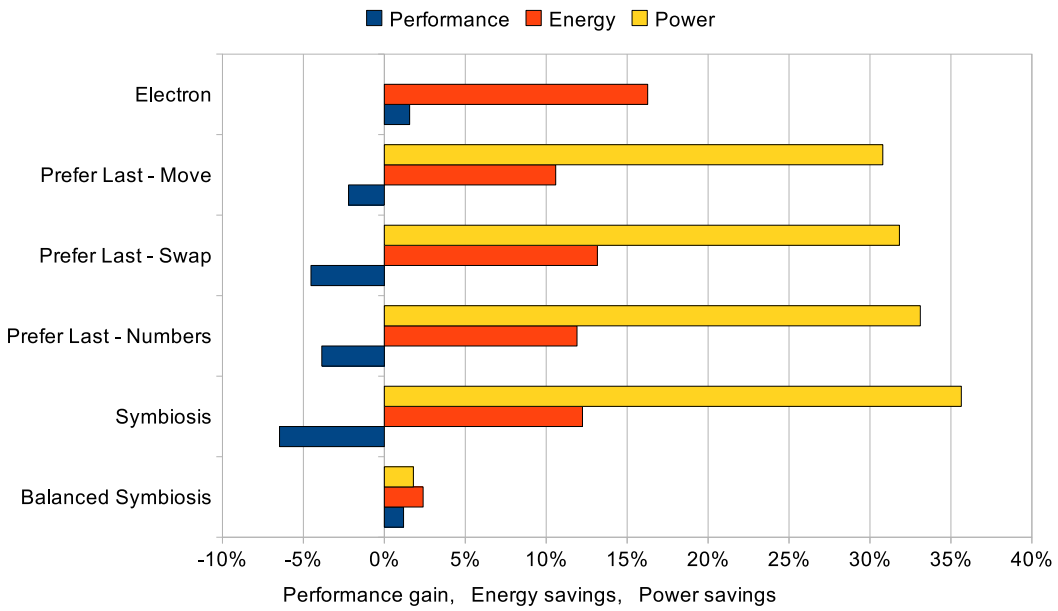

Figure 3.6: The impact on performance, energy, and power of various thread scheduling policies.

3.6.2 Scheduling for Other Metrics

This section discusses scheduling for other scenarios where it is less critical to have both low energy and high performance at the same time. These scenarios differ from the previous section in that at least the shape of good schedules is more predictable. However, we find that even in these cases, directed scheduling policies still enable us to find specific schedules that better use the available resources and group threads in ways that minimize negative interference.

Figure 3.6 compares various scheduling policies directed towards performance (measured as weighted speedup), energy, and power (using 8-thread workloads). With these metrics, it is still beneficial to consider unbalanced schedules (see Figure 3.7). However, in general, the improvement is less significant from these schemes. Simply balancing the system load evenly over compute nodes is often a sufficient mechanism for extracting good performance. What gain there is comes primarily from finding symbiotic schedules that are better than the random groupings.

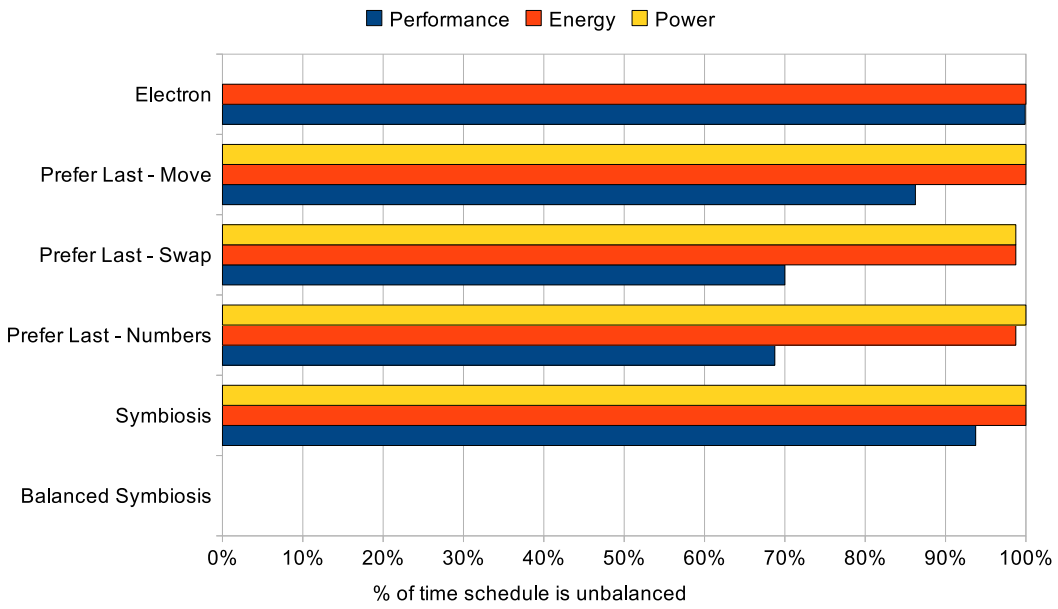

Figure 3.7: Extent of imbalance in the dynamic schedules targeting the various metrics.

For the power-related metrics (energy and power), we see that significant benefits can be had with directed scheduling policies. Note that no power bar is shown for the electron policy because no electron policy is optimized for power (since such a policy would be trivial). *Electron Energy* emerges as the best scheduling policy for energy. It results in over 15% energy savings—high gains can be attributed to the push/pull behavior trying to cluster threads intelligently on as few cores as possible. Other scheduling policies that allow unbalanced scheduling also result in over 10% savings. *Balanced Symbiosis* fails to achieve significant savings because it is constrained to utilize all the cores all the time. Scheduling for power exposes the value of unbalanced scheduling even more. *Balanced Symbiosis* results in less than 2% power savings. Policies that allow unbalanced scheduling can lead to more than 35% power savings.

3.7 Conclusions

A chip multithreading architecture, with multiple SMT cores on chip, has a unique ability to partition distributed execution resources to each application according to its individual needs. But this requires appropriately assigning threads to cores, as the execution resources available to a thread depend on how many threads are assigned to the same core and exactly which threads it shares the core with.

A traditional multiprocessor scheduler, applied to this architecture, will not identify the best schedules. To do this, a good scheduler must be able to explore both balanced and unbalanced schedules. Additionally, it must be able to distinguish between good and bad coschedules and navigate the huge space of potential schedules to continuously evolve toward better ones.

This chapter proposes several operating system level thread scheduling policies for such an architecture. These adaptive policies are particularly critical when both performance and energy are first-class concerns. In that scenario, neither distributing threads across cores nor aggregating threads on few cores is clearly the best policy, but the right schedules are workload-dependent and can only be identified by policies that adapt dynamically to the current program behavior. We show gains, versus a random scheduler that always uses balanced schedules, of 6-11% in energy-delay product. We also observe gains when scheduling for pure energy, performance, or power.

Acknowledgments

The authors would like to thank the reviewers for their feedback and Jeff Brown for his assistance with the simulator. This research was funded by NSF grant CNS-0311683, Semiconductor Research Corporation grant 2005-HJ-1313, grants from Intel, and an IBM Fellowship.

This chapter contains material from "Exploiting Unbalanced Thread Scheduling for Energy and Performance on a CMP of SMT Processors", by Matthew DeVuyst, Rakesh Kumar, and Dean M. Tullsen, which appears in *Pro-*

ceedings of the IEEE International Parallel and Distributed Processing Symposium 2006 (IPDPS). The dissertation author was the primary investigator and author of this paper.

Chapter 4

Execution Migration on a Heterogeneous CMP

4.1 Introduction

The previous chapter demonstrates how heterogeneity, via unbalanced schedules on homogeneous hardware, can be exploited by applications that cannot benefit from parallelization. With custom cores (hardware heterogeneity), there is even more opportunity for single-threaded applications to find a good core on which to run. This chapter addresses the challenge of scheduling on heterogeneous architectures (heterogeneous-ISA CMPs, in particular), focusing on the mechanism by which a thread can move to the "right" core.

Current industry offerings include only homogeneous CMPs—all cores on a die are identical; however, research has shown that heterogeneous CMPs can achieve even greater performance and power efficiency [KTJR05, HM08]. Allowing the set of heterogeneous cores on a chip to conform to different instruction set architectures adds a new degree of freedom to heterogeneous CMP design. ISAs can be designed to meet different goals: some are designed to make hardware implementation simple, to reduce code size, to reduce memory accesses, to enable more energy-efficient hardware design, to support a wider range of computation types (e.g., floating-point). Heterogeneous-ISA CMPs allow architects more flexibility in

creating more efficient multicore microprocessors.

There are many obstacles that must be overcome to make general-purpose heterogeneous-ISA CMPs feasible. This chapter proposes a solution to one of those problems: process migration. The ability to migrate a running program among heterogeneous cores is critical because it allows programs to capitalize on the available heterogeneity by being able to adapt to both phase changes and environmental changes. Some examples include:

- When the power state of the computer changes (e.g., a laptop going from normal to low-battery operation), programs running on a core designed for high performance could be migrated to a core designed for energy efficiency.
- When a new process with high priority and high performance demands enters the run queue, other processes could be migrated away from the most powerful core.
- If a program enters a new phase of execution with different computational demands (e.g., floating-point intensive code or cryptographic code), execution could migrate to a core with strong support for the new computation type (e.g., dedicated floating-point hardware or cryptographic instructions).
- If a part of the chip becomes too hot, programs executing on cores in that region could migrate away, possibly to cores of different ISAs, in a cooler region of the chip, or to cores that are designed to run cooler.

While no general-purpose heterogeneous CMPs are on the market at this time, special-purpose heterogeneous multiprocessors have been produced. Called heterogeneous multiprocessor systems-on-chips (MPSoCs), these architectures are designed for computation in specific domains that require a diverse set of algorithms to solve a problem. Examples of heterogeneous MPSoCs include the C-5 Network processor [C-P01], the Philips Viper Nexperia (a multimedia processor) [DJR01], the Texas Instruments (TI) OMAP architecture (a cell phone processor) [Tex06], and the Cell processor [KDH+05] (designed for use in game systems). These systems do not support migration. This is a significant handicap

because the ability to migrate removes the burden from the programmer to figure out where the code should be running at all times. The programmer does not have to carefully plan how various pieces of a program will execute on specialized cores. Instead, the compiler can produce different versions of a single program for each core type; and as the program moves through various phases of execution (that map to specialized processing cores), execution will automatically migrate to the appropriate core. This chapter focuses on migration in the context of general-purpose heterogeneous CMPs. Migration in this context is more important than on embedded systems where hardware-software co-design is more common and flexibility is less important. Nevertheless, migration can benefit heterogeneous MPSoCs as well by easing programmer burden.

Execution migration on a heterogeneous-ISA CMP is much more challenging than on a homogeneous CMP because program state (in registers and memory) is kept in an architecture-specific form. This requires state transformation during migration, which can be very expensive. Prior work on migration among heterogeneous systems has only considered migration among heterogeneous machines (e.g., among nodes in a grid), not among heterogeneous cores in a CMP. Migrating execution among cores on the same chip (instead of among machines) affords a unique opportunity for low-cost migration because state copy is not necessary since memory is shared. To take advantage of this opportunity, it is essential that as little transformation as possible be done at migration time. We accomplish this by ensuring that the memory state of the program at points throughout execution is nearly identical across compilations for different architectures. During migration most data objects do not need to be copied or repositioned. As a result, all pointers remain correct after migration, avoiding the cost and complexity of finding and fixing pointers during migration.

In this chapter we describe an execution migration technique for heterogeneous-ISA CMPs with the following goals (in order of increasing importance):

1. There should be frequent opportunities for migration—points in execution when migration is possible.

2. The technique should not require type-safe code.

3. Making a program suitable for migration should not require any special effort by the programmer or user. Programs should be migratable simply by compiling them with our migration-aware build toolchain.

4. No special hardware support should be necessary for migration.

5. Migration should have low performance overhead.

6. The normal runtime performance of the application (without migration) should be minimally impacted.

7. Correctness should be preserved—that is, after a migration, a program should behave as if it had always been running on that core; its output should not deviate from a non-migrated instance running on either core.

The contributions of this chapter are as follows. We describe a new execution migration technique suited to a new domain (heterogeneous-ISA CMPs). The key insight of this technique is that because migration overhead is not dominated by state copy (since memory is shared among cores), migration can be fast if a small amount of state has to be transformed. We measure the costs and overheads of our migration technique. Finally, we describe areas where there is opportunity and need for further work to produce faster migration-safe code with support for more features.

In the next section we discuss related work in the area of execution migration as it has been applied to other heterogeneous domains. Then in Section 4.3 we give an overview of our migration strategy. Section 4.4 describes how we ensure a nearly identical memory image across compilation for different architectures to make migration fast. Section 4.5 describes migration itself. Section 4.6 details our experimental methodology, while Section 4.7 shows the performance of migration, and Section 4.8 concludes.

4.2 Related Work

To our knowledge, no previous work has addressed the problem of migrating a process among heterogeneous-ISA cores on a chip multiprocessor. There are, however, two closely related problems. The first is the problem of process migration among computers of different architectures that are members of a cluster, grid, or distributed system. The second is the problem of checkpointing and recovery (CPR) of processes on computers of different architectures. In every proposed solution to both of these problems, large-scale state copy is necessary and dominates latency. The problem that this chapter addresses is different because state copy is not necessary since memory is shared. Consequently, there is an opportunity for low-cost migration if state transformation cost is low. As a result, we spend considerable effort optimizing pieces of our migration overhead that would have been inconsequential in those systems.

Nevertheless, much can be learned from the proposed solutions to the problems of inter-machine migration and CPR. This section focuses on how researchers have addressed these two related problems. First, we discuss the motivations that have been driving past research on these problems. Then, as we summarize important works in the area, we extrapolate key ideas that are applicable to the problem that we address, point out differences in the problems, and contrast the solutions.

Process migration among compute nodes (whether homogeneous or heterogeneous) in a cluster, grid, or distributed system is desirable for a number of reasons. It may be necessary to migrate running processes for load balancing—to dynamically distribute computation needs among the collection of resources as job needs change or compute resources change. It may also be necessary to migrate a process to take advantage of data locality; sometimes it is more expensive to move data to where the data needs to be computed than it is to move the computation to the data. Resource fulfillment is another common reason for migration—some machines may be able to run certain jobs more efficiently than others. Finally, processes may need to be migrated away from a machine that will be taken offline for maintenance.

A similar problem to inter-machine migration is checkpointing and recovery.

In fact, migration may be viewed as a special case of CPR, where migration is accomplished by taking a checkpoint of process state on one machine, copying that checkpoint to another machine, and restoring the checkpoint there. CPR is useful in many scenarios, fault tolerance being the most common. If the failed node is not repaired quickly and it is not practical to recover on a node with an identical architecture, then heterogeneous CPR allows recovery to take place on a node with a different architecture. Other uses of CPR include debugging support, auditing, and service interruption for maintenance. Since the problems of inter-machine process migration and CPR are so closely related, most of the solutions presented in the literature address both problems simultaneously.

4.2.1 Homogeneous Migration and Checkpointing

Approaches to homogeneous inter-machine process migration and CPR are significantly different than approaches for the heterogeneous case. For heterogeneous migration and CPR, application details must be considered because the application is represented differently on different platforms. However, for the homogeneous case, the state of the application can be treated as a black box. Only the system's view of the application's state must be considered—the process memory image and register state (including program counter) can be saved without modification, left in its architecture-specific representation. The application may be migrated/checkpointed by any external agent—either the operating system [HD06, BGW93] or a user-level runtime system [LTBL97, PBKL95]. This is called a system-level approach. It has the advantages of being easy to implement, not interfering with runtime performance, and requiring no changes to application code (by the programmer or a compiler/preprocessor).

While efficient *homogeneous* migration and checkpointing is an interesting problem, the problem we address (of process migration on a heterogeneous-ISA CMP) is more closely related to the problem of inter-machine *heterogeneous* migration/CPR; the rest of the related work discussed in this section focuses on inter-machine heterogeneous migration and checkpointing.

4.2.2 Theoretical Models

Von Bank, et al. [vBSS94] give a theoretical model, including a formal definition of heterogeneous process migration. They develop the model around points of equivalence among multiple representations of a single program with identical stimuli. From this model, they describe, in abstract terms, how computations occur in a stack-based procedural language like C. They identify three properties of an equivalence point (a point of potential migration): an equivalent set of live variables, all the live variables stored in memory instead of registers (not a strict requirement, but an implementation convenience), and the ability to relate the function call graphs of the program compiled for two machines. They recognize that the compiler must avoid optimizations that move code across equivalence points or carry temporary values across equivalence points. They assert that the language system (the compiler, assembler, linker, etc.) can be designed to ensure points of equivalence at a desired granularity; but the finer the granularity, the more performance will suffer because the compiler will be more constrained and less able to make machine-dependent optimizations—a conclusion that we confirm empirically in this chapter. They postulate that the ideal granularity is at a subset of the function call sites. The particular subset depends on the "compatible subgraph", which is the subgraph of the function call graphs for each target that are the same. Though their model is well-defined, they do not provide any detail on how to implement a compiler that is able to create equivalence points and facilitate migration. This is one of the contributions of this chapter.

4.2.3 Early Work

Some of the earliest work on process migration was by Dubach and Shub [DRS89, Shu90]. Their solution is meant to work on the equivalent of a strongly-typed language (to facilitate data translation). They follow a principle they call "greatest common denominator" (GCD), where data is placed in memory with extra padding, if necessary, in order to accommodate the largest data representation among architectures to which execution might be migrated. This is applied to most memory segments except the code section. This allows most pointers to

work without the need for correction. We applied the same GCD principle to our work; only in our case, the migrated process uses the exact same memory image instead of a copy of it. Also our compiler ensures that function entry points appear at the same offsets within the code sections so that, after the OS remaps the pages of the code section during migration, the function entry points will be at identical addresses. As a result, function pointers do not need to be located and fixed.

4.2.4 Instrumenting Well-Typed C Code

Many works propose migration through the instrumentation of well-typed C code. The definition of well-typed code is different among these works, with some work supporting some type-unsafe constructs. In contrast, the migration technique this chapter proposes does not place any such restrictions on the code. Nevertheless, our technique has a great deal in common with inter-machine migration techniques for well-typed C code.

Fernandes, et al. [FPS06] Recent work includes that of Fernandes, et al., who extend the Cornell Checkpointing Compiler (a C preprocessor and runtime library) [BMPS03], with the ability to take portable checkpoints for migration among heterogeneous computers. Their preprocessor must determine the type of every data object. It adds instrumentation code to save (at runtime) the address and size of every object when it comes into existence. At checkpoint/migration time, this information is used to create new memory sections (stack, heap, globals) one object at a time, doing any necessary translation during re-creation. Pointers are fixed by searching through the object descriptors on the source machine for the object pointed to by the address, and setting the pointer value to the address of that object on the target machine. The preprocessor also adds push and pop function calls before and after every call site in the original code. The program uses these to record where it is executing, so that the sequence of call sites leading up to the place in the code where a checkpoint is taken can be determined [BMPS03].

Karablieh, et al. [KBH01] They address the problem of CPR among heterogeneous machines with a different objective. Their goal is maximum heterogeneity, supporting new platforms without modification or special configuration; Therefore, they are willing to sacrifice some performance for portability. Their preprocessor inserts code to use goto statements, calls, and returns triggered at CPR time to unwind and re-create the stack and bring the program counter to the correct position. During the recovery of a stack frame, local variables are restored and then control is transferred to the appropriate call site where the next function in the activation history is called. To facilitate the checkpointing of stack-based variables, all the local variables in a function are wrapped in a struct. The program is instrumented to maintain a *state stack*, where every element contains a pointer to the struct of locals and a label to the location in the code of the call site of the next function called. As functions are entered and exited, nodes are created and destroyed on the state stack. To store data in an architecture-neutral format, *fprintf* and *fscanf* are used during CPR. Instead of allowing a program to use native pointers and then translating all pointers at migration time, they create an additional level of indirection—what they call the *Memory Refractor* (MR). The instrumented program keeps this structure up-to-date at runtime. The MR contains size and type information about every variable (whether global, local, or heap-based) and a pointer to the location of that variable. All pointers in the original code are transformed into architecture-independent offsets into the MR. As a consequence of this extra abstraction, all pointer dereferences require an additional memory access.

In a heterogeneous CMP environment, where state copy is not necessary because memory is shared, Karablieh, et al.'s method for transforming state (using *fprintf* and *fscanf*) would result in high migration overhead—something our migration technique avoids. Our solution to pointer correctness after migration is also different: they use an MR, which hurts runtime performance by adding an additional dereference to every pointer access; we keep pointed-to objects at fixed locations in memory so that no additional indirection is necessary.

Ramkumar and Strumpen [RS97, SR96, Str98] These papers propose application-level checkpointing (where the application, instead of the OS, directs CPR) using a C-to-C compiler that adds code to explicitly save and restore variables by name and unwind and rewind the stack. Pointers are converted to a machine-independent format and back to a native format at migration time. Data is converted to a machine-independent format (that they call Universal Checkpoint Format) during checkpointing and then to the appropriate machine-dependent format during recovery. Their compiler generates what they call *type metrics*, which are tree structures that describe the properties of and relationships among all the data types used in a program's code. These aid in data conversion, including handling differences in padding and alignment, and pointer conversion.

Ferrari, et al. [FCG00, Fer98] These papers describe a migration scheme very similar to that of Ramkumar and Strumpen. One notable difference is their use of a receiver-makes-right policy for data translation. Instead of converting all data to a universal format when taking a checkpoint and then converting that to a machine-specific format during restoration, the checkpointing machine does no data conversion and the restoring machine converts from the native format of the sender to its own native format. The advantage of this scheme is that if the two architectures are identical (i.e., if the migration is homogeneous) or if they are very similar, the process incurs less translation overhead. However, this leads to more complexity since each machine type must be able to translate from every other supported machine type; in contrast, with a universal intermediate format, each machine type only needs to be able to translate to and from the universal format. They use polling to initiate checkpointing. At certain points of execution—which they call poll points—a state variable is checked to determine if the program has been put in the checkpointing state. If it has, then variables are saved and the stack is unwound using a sequence of `return` statements. They support two kinds of poll points—mandatory and optional. Mandatory poll points are placed around every call site that may potentially lead to a point in the code where a checkpoint request is detected by another poll point. Optional poll points can be placed anywhere at the discretion of the programmer. They are usually placed in regions

of code that execute for long periods of time without making function calls so that the average latency to checkpoint request detection is low. The authors introduce some simple heuristics for the automatic placement of optional poll points. We use similar heuristics but evaluate their effectiveness using a new metric and more realistic benchmarks.

Smith and Hutchinson [SH96] Instead of modifying the application to conduct its own migration, Smith and Hutchinson offload most of that work onto the compiler and a pair of separate processes, `migrout` and `migrin`. They modify the ACK compiler to disable optimizations across call sites and to record extra information about the program, including details about the type and location of every variable at every call site. `migrout` uses this information to inspect the program's memory image and convert it to a machine-independent form; `migrin` uses the information to convert from the machine-independent form to a machine-specific form on the destination machine. This reduces runtime overhead; however, pointer translation during migration is a slow operation, especially if the program uses many pointers.

Chanchio and Sun [CS02] The technique of Chanchio and Sun is very similar to Ferrari's. One improvement is the checkpointing of only live local variables (instead of all locals) at each call site. Their technique also differs in when stack frames are restored. Instead of translating and restoring all stack frames before execution is resumed, stack frames are translated and restored lazily as functions are returned. This reduces migration overhead at the expense of runtime overhead.

Pointers in languages like C present a difficult problem for heterogeneous migration because they are machine-dependent, allowing direct access to any object in the address space of a process. No prior work has been able to efficiently deal with the problem of pointers. Either runtime performance is sacrificed ([KBH01]) or migration performance is ([FPS06, RS97, SR96, Str98, FCG00, Fer98, SH96, CS02]). One of the main contributions of our work is to demonstrate an effective method of dealing with pointers (by ensuring all pointed-to objects remain at fixed locations).

4.2.5 Other Languages

The techniques presented in this section contain valuable contributions to the topic of heterogeneous migration but do so by taking advantage of the strong typing and mobile features of other programming languages. In contrast, our technique is effective in type-unsafe languages like C.

Veldema and Philippsen [VP05] Their paper proposes a migration technique for Java code. The primary goal of their work is to reduce runtime overhead; to achieve this goal they utilize heavy compiler support. The compiler performs all high-level machine-independent optimization passes and then creates a Usage Descriptor String (UDS) for each live local variable. The UDS is a string representing the computations necessary to produce the value of the variable at a call site. Each local variable is uniquely identified by a UDS, which is the same across compilation to different target architectures. Associated with each UDS string representing a variable is the location of that variable when the call site is reached (e.g., on the stack or in a register). As the low-level optimization passes change the variable's location, the mapping is updated. With this mapping, they are able to migrate local variable values across architectures. The compiler creates two functions for every call site: one that knows how to checkpoint the local state at that call site and another that knows how to restore local state at that call site (in a machine-dependent way, based on the UDS-to-location mappings).

Steensgaard and Jul [SJ95] This work adds heterogeneous mobility to the Emerald distributed programming language. Emerald programs consist of a set of objects distributed among different machines. Accesses to objects (even if they reside on different machines) are transparent. Originally, Emerald only supported the distribution of objects among homogeneous machines; Steensgaard and Jul add support for distribution among heterogeneous machines, utilizing the compiler to keep track of where data values are kept in the activation records and registers. This allows them to translate a machine-specific activation record to a machine-independent activation record and then back to a machine-specific activation record

for a different machine. They discuss when, during execution, it is safe to migrate—they call these points *bus stops*. A bus stop exists at procedure calls, system calls, and at the bottom of loops. The compiler is free to optimize code between bus stops, but not across. They give a few examples of such optimizations that move code across bus stops and they propose some solutions, though they do not implement the solutions or measure their effectiveness. They identify code motion, strength reduction, and instruction selection as three optimizations that may take place across migration points. They propose that bridging code be introduced by the compiler and executed during migration to reverse the effects of code reordering.

4.2.6 Other Directions

The following works propose less traditional approaches to the inter-machine migration problem.

Ssu, et al. [SF98, SFJ03] In this work, migration cost is reduced by having a master process on one machine and slave processes on machines of different architectures. The master process executes the program and periodically sends updates (on some procedure entries) to the slave processes; the slave processes sit idle most of the time, occasionally receiving state updates from the master, which they translate into a native format. Checkpointing is done using traditional homogeneous checkpointing techniques. Since the master does not need to convert any data itself, it has low overhead. The data conversion latency and checkpointing latency happen in parallel with regular execution, hiding their effects. Their results show low runtime overhead and little processor utilization by the slave processes; however, the checkpoint interval is around 5 minutes. They do not create or modify a compiler to handle the code changes, instead modifying the source code by hand (often increasing code size by 35%). They only support basic data types and do not support structs.

Theimer and Hayes [TH91] They propose that, at migration time, modified versions of the functions in the activation history be created. This code needs to be recompiled (at migration time), then run on the target machine. The new code restores all the variables, jumps to the appropriate code position, and continues normal execution. They do not implement their idea and measure its overhead, but it is clear that such an approach will result in high migration overhead due to the cost of compilation.

4.2.7 Differences

One important difference between the problem of process migration among computers and process migration among cores on a single chip is the need for I/O. When a process is to be migrated among computers, state must either be saved to stable storage (checkpointed), transferred, and restored, or it must be transferred directly over a network of some kind. Migration within a single machine requires no such I/O transfer—process state is kept in the same memory banks. A common focus of the related work is on efficiently storing state, including distilling or compressing needed state. Because the I/O overheads dominate migration performance, the computational overhead is less important. Since the problem we address does not have an I/O component, the computational component of migration is critical for good performance. We, therefore, focus on reducing the processing needed for migration.

Another important consequence of the difference between the problems addressed in this chapter and prior work is the ease of extending migration to other architectures. When process migration in a cluster environment is considered, it may be desirable to trade some performance for greater generality; the ability to easily add migration support for a new type of machine is attractive. However, the priorities are different for on-chip heterogeneous migration. A chip multiprocessor is designed by a single company. They may license some of the design from other companies or partner with them, but the effort will be unified. The different machine types in a heterogeneous cluster may be made by different (often competing) companies with their own agendas. In contrast, the designers of chip multipro-

cessors will put significant effort into integrating the different core types, carefully selecting core types that complement one another and work well together (e.g., they won't combine big-endian with little-endian and likely will keep fundamental data types and sizes the same). Since it is reasonable to expect that significant effort goes into the design and integration of the core types found in a chip multiprocessor, it is also reasonable to expect that some extra effort (simple compiler support, at minimum) would be profitable in order to optimize migration among those cores. In this domain, achieving performance through specialization is more important than excess generality. Unlike in a heterogeneous cluster environment, no new core types can be added after the product ships.

Unfortunately, most of the related work lacks comprehensive empirical performance results. Some present no performance results or incomplete results (either showing only runtime overhead or only migration overhead). Most only present results for a select few programs (most of them toy programs). One of the contributions of this chapter is to give a more complete performance analysis of our technique using a set of common benchmark programs.

4.3 Overview of Migration

Though the operating system is responsible for coordinating migration, we do not restrict the impetus for execution migration to come from the operating system. It is possible for the process itself to request migration (based on a self-awareness of its own needs), for the operating system to request migration (based on the availability of resources and the needs of other processes), or for some external agent (like the user) to request migration (based on a desired level of service).

By whichever means migration is requested, the operating system must perform three actions to facilitate migration. It must reschedule the process on another core, it must change page table mappings to facilitate access to the code for the migrated-to core, and it must spawn a process to transform memory for state migration. The first two tasks—process scheduling and page table manipulation—

are common operating system responsibilities and require no further discussion here. The final task—state transformation—is unique. The transformation process can either be a part of the operating system itself or it can exist as a privileged user process that is simply spawned by the operating system. Either way, the operation of the transforming process is the same. We describe this process in Section 4.5.

Before we continue with our overview of migration, we first state the assumptions that this chapter makes and describe the ISAs we use in this study.

Assumptions We assume that there is no architecture-specific code in the program to be migrated. We focus on migrating C code, free of inline assembly code. To support programs with custom assembly code, assembly code would need to be provided for all the architectures on which the process may run. We also assume that externally-linked libraries are compiled for migration. For libraries that are tuned to specific architectures, this will require some code changes. The C library, for example, because it contains a significant amount of architecture-specific code, will require significant refactoring to be migration-safe. For the testing of our migration technique on programs which are linked to C library code, we restrict migration to occur only within non-library code. Finally, we assume some similarities in the ISAs: the same endianness, fundamental data size, and floating point format. Otherwise, significant portions of memory would have to be transformed during migration.

ISAs For this study, we model a small, low-power ARM core and a large, high-performance MIPS core. ARM and MIPS are both 32-bit little-endian RISC ISAs with IEEE 754 floating-point support. Despite their similarities, they also represent a measure of diversity, both in design and application. In terms of design, ARM has many features that MIPS lacks: an abundance of predicated instructions (almost every instruction in the ISA has support for predication), load-multiple and store-multiple instructions, and a program counter accessible as a general-purpose register. MIPS, on the other hand, has double the number of accessible integer registers as ARM. In terms of application, ARM is most commonly used in highly power-constrained devices. MIPS cores, while also commonly used in power-

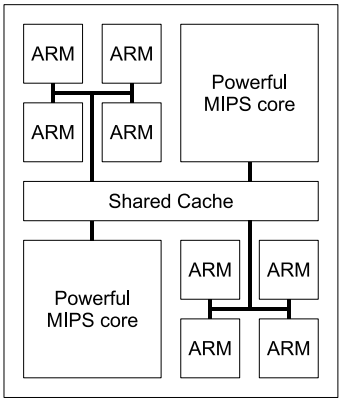

Figure 4.1: Heterogeneous-ISA CMP with two powerful MIPS cores and eight energy-efficient ARM cores.

constrained devices, have also been used in more performance-focused devices (like the Origin series of SGI supercomputers).

Figure 4.1 is a diagram of the type of CMP we model. The number of cores of each type is irrelevant in this study, as we propose the migration mechanism, not the policy (the former need only consider the migrated-from core and the migrated-to core; the latter must consider all cores). The detailed architectural model of our CMP is described in Section 4.6.2.

We experiment with only two ISAs, but our migration technique should be feasible for CMPs with more than two core types. The greatest difficulty in adapting the technique to more than two architectures is dealing with additional compiler complexity. A migration-aware compiler should be designed in a very modular fashion to easily accommodate generating compatible code for additional architectures. As we describe in Section 4.5, the key to state transformation involves creating mappings of objects from source-level/common names (e.g., local variable "foo") to architecture-specific locations (e.g., ARM register 5). For each additional architecture compiled for, an additional mapping would need to be created. Because mappings are made to a common representation, the number of mappings the compiler must create is linear with the number of supported architectures.

Fast migration requires minimal state transformation. And this hinges on

memory image consistency—the memory image at a point P in the execution of a program on ISA A should be nearly identical to the memory image at P on ISA B so that very little state must be transformed to a machine-dependent form during migration. This is the heart of our migration strategy. The next section describes how memory image consistency is achieved. Then Section 4.5 describes the migration itself.

4.4 Memory Image Consistency

In order to facilitate fast execution migration, each binary representation of a program (compiled for different ISAs) should expect to find each item of program data at the same virtual address. This makes it possible to perform the migration without having to rearrange data items in memory, reducing the latency of the migration process. Contrast this with traditional inter-machine heterogeneous migration where state copy dominates migration overhead and the cost of reordering and transforming memory is marginal. Migration on a CMP does not involve memory state copy, so keeping transformation costs low is critical to good performance.

To achieve memory image consistency, first the overall structure of the program sections for each ISA needs to be consistent: the types of sections, their composition from the sections of various object files, and their alignments must match. Also, within each section, the number of objects, their sizes, their relative order in memory, and their alignment and padding rules must be identical in order for their virtual addresses to be consistent across ISAs. This not only applies to data sections, but also to sections containing code. This is necessary so that function pointers will be accurate after a migration. In this case, the "objects" within the section that must be consistently placed are function definitions. All functions specified for each ISA must start at the same virtual addresses. The dynamic portions of memory—the heap and the stack—also need to be consistent.

4.4.1 Overall Section Structure

By specifying a consistent placement of the major program sections in the virtual address space for each ISA, we take the first steps toward a common data and code layout. We accomplish consistent section placement by modifying the linker scripts of the GNU linker, *ld*. Consistent ordering of and alignment between similar sections aggregated from multiple source files is achieved with modifications to both the linker and the assembler.

ISAs may utilize entirely different sections for the same data. This is the case for the *small* data sections in MIPS: the *.sbss*, *.sdata*, and *.scommon* sections. Data objects below a certain size are placed in the small data sections; this allows for faster access to these data objects since it takes fewer instructions to reach these items from a base pointer (like the global pointer). This is a consequence of the limited amount of space (16 bits) in a MIPS instruction to specify the offset from the base pointer. In order that data does not have to be copied in or out of small data sections at migration time, we add support for small data sections in ARM. However, at this point we do not change the compiler to take advantage of the small data section by generating ARM code to access data in these sections with fewer instructions—so the performance of ARM code is not improved, but is not degraded either.

4.4.2 Code Section Consistency

One program section that needs some work to ease migration is the code (*.text*) section. This is because function pointers pose a potential problem to fast, efficient migration. If a pointer is taken to a function, *foo*, and saved somewhere in memory (either on the stack, on the heap, or in a global variable) and later in time a migration occurs, that pointer may no longer be correct. To remedy this, function bodies, like data objects, should be consistently placed in memory across each ISA. This will allow all function pointers to work correctly after migration without the the need to find and fix all the function pointers in memory.

Before function bodies can be placed at consistent memory addresses, the compiler must make consistent decisions regarding *which* functions to produce code

for. If care is not taken, a function may be inlined away as it is compiled for one ISA but not another. For example, if a C function is declared static and everywhere it is called a copy of it is inlined, the original function definition may be eliminated altogether. If different inlining decisions are made as the program is compiled for different ISAs, a different set of function definitions may appear for each. To create a common set of functions for which code is generated, the compiler is modified to keep definitions for all functions even if they are inlined at every call site. This only marginally increases code size and we measure no effect on performance.

The order of function definitions in memory must be identical, otherwise function pointers will be incorrect after migration. To ensure this, the compiler emits function definitions in the order they are encountered in the source files.

Finally, to make the start addresses of each function identical, the code size of each function must be identical for every ISA. The assembler adds NOP instructions, when necessary, to pad functions to the appropriate size. As a result, each function compiled for each ISA matches in size and start address. While this method increases code size a little, we measure no performance loss.

4.4.3 Heap Consistency

Code that accesses the heap must be consistent—we call this *heap consistency*. Heap consistency is not necessary so that pointers to the heap will work correctly after migration. Rather, it is needed to ensure that after migration the program has an accurate record of what heap memory is allocated and what is free. This requires that the same implementation of *malloc* be used for all ISAs. Then at migration time, because all *malloc*'s internal data structures are preserved, a consistent view of the heap will be maintained. The same principle applies to any memory management library a program uses.

In the Linux system that we model, *malloc* acquires memory on behalf of the caller in two ways: through the *brk* system call, which grows the heap, and through the *mmap* system call. The *mmap* system call does not return heap memory, but pages of virtual memory that the operating system has allocated to the process. Since a single operating system instance governs all the cores, a common page table

is used, and page allocations allocations will not change, despite the migration.

4.4.4 Stack Consistency

The stack is especially difficult to make consistent without sacrificing too much performance because stack interaction is carefully optimized for each ISA. For example, in an ISA with a large number of registers, many function arguments may be passed through registers to avoid loads and stores to stack memory. But for an ISA with a small number of registers, most function arguments must be passed on the stack. Our goal is to find the right balance between good runtime performance and low migration overhead. We must change stack memory as little as possible at migration time without eliminating performance-critical ISA-specific stack optimizations. This section describes what should be changed (and what should not be changed) about the stack organization to strike this balance. First, we review the major sub-components of a frame. Then we describe changes that apply to the frame as a whole. Finally, we describe changes that apply to each sub-component.

Components of a Frame Each stack frame is composed of different parts. The exact number of parts and their relative order depends on (1) the needs of the ISA, (2) the type of code being generated (e.g., position independent code may require additional space to save a pointer into a global offset table), and (3) the needs of the function being compiled (e.g., non-leaf functions may place outgoing arguments in their frames). Common components of a frame include arguments, call-saved register spill slots, and local variables. Arguments include both incoming arguments and outgoing arguments (only if the function calls another function). Usually, incoming stack-based arguments to a function are considered part of the caller's frame. Call-saved registers can be of two types: callee-saved and caller-saved. A calling function that wishes to preserve the values in some of its caller-saved registers across a function call will save those registers to its own frame. A called function that intends to overwrite callee-saved registers of its caller will first spill those register values to slots in its frame. Finally, some functions will

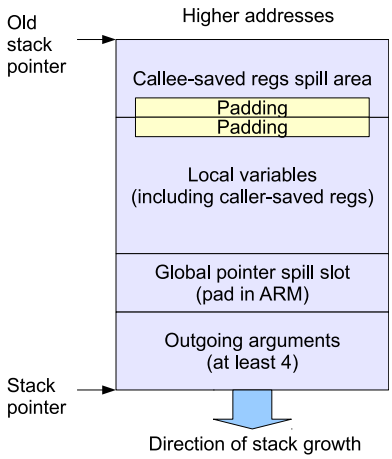

Figure 4.2: The organization of a non-leaf stack frame in a migration-capable program.

require stack space for local variables. This includes any compiler-created locals and excludes any compiler-eliminated locals and any locals bound to registers. For ISAs with moderate numbers of registers and functions with only a few locals, it's common to see the locals section of the stack frame omitted.

We now describe all of the general stack layout changes to avoid costly stack transformations during migration, including (1) direction of growth, (2) size, (3) ordering, and (4) alignment. If any of the following changes are not made, *all* of the data on the stack will need to be repositioned during migration and all pointers to stack-based objects will need to be fixed.

First, the stack's direction of growth should be the same: in both ARM and MIPS, the stack grows downward. Second, we compile the code in each function such that, when it executes, it will make the stack frame the same size for each ISA (by adding padding), making the overall stack size consistent.

It is also necessary to have a consistent ordering of regions within each frame. In our implementation, a callee-saved register spill region comes first (at the highest addresses of the frame), followed by space for locals and space for outgoing arguments (if the function is not a leaf). MIPS requires an additional region to save the global pointer register. This region is placed between the locals area and the outgoing arguments area. To account for this small space in ARM,

we modify the compiler to add additional padding to stack frames when compiling ARM code. The cross-architectural layout of a non-leaf stack frame is shown in Figure 4.2.

In order to achieve identical frame size, it is also necessary to maintain consistent alignment and padding rules between regions. In our implementation, each region is 8-byte aligned.

We now discuss specific changes to each major frame sub-component to avoid costly stack transformations during migration. The major frame sub-components are:

1. the function arguments region
2. the callee-saved register spill area
3. slots for local variables

Function Arguments

After migration, the program must be able to locate arguments to open functions without requiring stack transformation during migration. First, we review argument-passing conventions in MIPS and ARM. Then we describe how problematic differences can be reconciled. Finally, we consider a special case of argument passing: functions that accept a variable number of arguments.

Arguments may be passed to a function in two ways: through registers or through stack memory. Each ISA has its own calling conventions specifying how arguments should be passed. To enable migration, it is necessary to modify some of these conventions so that functions can find their arguments after migration. Changes are also necessary to ensure consistent frame size because stack-bound arguments contribute to overall frame size. Forcing arguments on the stack that would otherwise have been passed through registers can hurt performance, so we avoid this.

For MIPS, the convention is for non-leaf functions to allocate space in their own frames for a minimum of four outbound arguments. However, there are four registers in which the first four arguments to a function are passed. If the called

function needs to overwrite any of the argument registers, it may first spill those registers to the pre-allocated argument slots on the stack. For ARM, the convention is to only allocate space on the stack for arguments beyond the first four. The first four arguments are always passed through registers.

To reconcile these differences (so that arguments may be located after migration) without sacrificing performance, we modify how the compiler generates ARM code to handle function arguments. The compiler allocates stack space for the first four arguments, though the caller does not fill these slots. The called function is made aware of this region as a possible spill location for the argument resisters if it needs to reuse those registers. Performance is not affected, as the same number of arguments are passed through registers instead of going to memory. Memory usage is increased by about 16 bytes per frame. This additional memory consumption has no noticeable effect on performance.

Variadic functions—functions that take a variable number of arguments (like C's *printf* function)—create an additional challenge to consistent calling conventions. If the conventions for handling variadic functions differ among the ISAs on the CMP, arguments to variadic functions will not be locatable after migration. We briefly describe how variadic functions work and the conventions followed by MIPS and ARM; then we outline the changes that must be made to enable migration in variadic functions.

The protocol for handling arguments to variadic functions is as follows. All of the anonymous (unnamed) arguments are required to be laid out contiguously on the stack. Initially, a local variable is created that is a pointer to the stack location of the first anonymous argument. It is up to the called function to determine the number of arguments passed. The function iteratively accesses each anonymous argument in order. It gets the value for the next anonymous argument by following the pointer just described. After the argument has been read, the pointer is incremented so that it points to the next anonymous argument.

The details of how the original GCC compiler handled variadic functions differ for each ISA. For MIPS, in non-variadic functions, the first four argument stack locations are allocated but may never be filled. For variadic functions, if

the first anonymous argument is within the first four arguments, then some of those argument slots must be filled, otherwise the pointer to the next anonymous argument will point to an empty slot. Therefore, when called, the first thing a variadic function does is spill some of its argument registers to the already-allocated stack locations in the parent's frame (assuming that the first anonymous argument lies within the first four arguments).

In ARM, since the first four arguments are passed through registers, a similar problem arises when the first anonymous argument lies within the first four arguments. But, unlike in MIPS, the convention in ARM is to not allocate stack space for the first four arguments. So on entry to a variadic function, the first action taken is to allocate N stack slots (where N is four minus the number of named arguments) in the called function's frame and spill N argument registers to them. We modify the behavior of GCC when it compiles variadic functions in ARM to (1) not allocate space for incoming anonymous arguments in its own stack space, (2) spill argument registers for anonymous arguments into the already-allocated slots in the caller's frame, and (3) correctly set the pointer to the first anonymous argument in the caller's frame.

It is important to note that all of these changes to argument passing rules require the recompilation of external libraries (primarily *libc*). This is necessary because the interface between callee and caller functions is changed and all calls back and forth between library code and user code must respect the new calling convention.

Callee-saved Register Spills

For a given function, the number of callee-saved registers that need to be overwritten depends heavily on (1) the number of registers the ISA supports (affecting register pressure), (2) constraints placed on ISA-specific special-purpose registers, and (3) low-level code transformations. Because all of these factors are architecture-specific, the size of the callee-saved register spill area differs greatly across ISAs. We therefore modify the compiler to add padding, as necessary, to the callee-saved register spill area of each stack frame. This change has no noticeable

effect on performance.

Local Variables

The last frame sub-component that requires modification is the area reserved for local variables. There are two reasons why changes to this frame area are necessary. First, pointers to stack-based variables should be correct after migration (and fixing them during migration is too costly). This means that pointed-to variables must be placed at identical addresses by each ISA. Second, the size of this area must be the same across ISAs, so that resizing (and therefore copying) is not necessary during migration. This section describes the necessary modifications to this part of the frame. These changes ensure (1) consistent ordering and (2) identical size (requiring padding).

The direction of stack-based local variable allocation within the locals region of the stack frame differs between architectures. For MIPS, GCC allocates variables in this region from low address to high; for ARM from high address to low. Large aggregate objects (like structs and arrays) and objects whose addresses are taken are allocated to the stack first. The different allocation directions cause these objects to be allocated at different addresses on the stack. For pointers to these objects to work after migration, we require all of these objects to be allocated to the same addresses across ISAs. To enforce this requirement we change the allocation direction; when GCC compiles ARM code it now allocates stack objects from low address to high. This change results in the identical placement of stack-bound variables to which there may be pointers.

Like the other regions of a stack frame described above, this region may require padding so that the size across architectures is identical. This is due to the fact the the number and choice of local variables to allocate to the stack is different across architectures.

Function Inlining

GCC makes function inlining decisions in its high-level tree optimization passes. Most high-level passes are architecture-independent, so the optimization

decisions that are made are identical whether compiling for ARM or MIPS. However, the function inlining passes use code size estimates, which are architecture-specific. To facilitate an identical view of the function activation records on the call stack at migration time, we force the compiler to make a common set of inlining decisions during compilation. We modify GCC to use only ARM code size estimates to guide inlining decisions. This has no noticeable effect on runtime performance.

4.5 Migration Process

In the last section we described the key to fast migration: obviating the need to transform most of the memory image during migration by keeping the memory image in an architecture-neutral state. The last section described what must be done at compile time to prepare for migration; this section describes the migration itself.

Migration, as we define it, is the continuation of a process' execution on a different core—a core with a different architecture and ISA. The source and target cores share the same memory. The migration process involves

1. halting execution of a migratable process on the source core,
2. performing minimal transformations on the process' memory image to make it ready for execution under a different ISA,
3. mapping the code section of the migrating process for the target ISA into the process' virtual address space (replacing the code section for the source ISA),
4. resuming execution on the target core.

All of these activities are under the control of the operating system. Most modern operating systems already have the underlying support necessary for steps 1, 3, and 4. Step 2 (memory image transformation) is the most critical to migration

latency. Because this step is the heart of heterogeneous migration, we focus on its characteristics and behavior in this section.

When all of the compiler/assembler/linker changes described in the previous section are applied, all of the program sections, except for some portions of the stack, are migratable without any transformation. We observe that keeping the stack in a state of complete architecture-neutrality would eliminate the need for any memory transformation at migration time, but would significantly hurt runtime performance. Architectures that rely on large numbers of registers for good performance would be affected more severely—data that would normally reside only in registers would be forced to the stack at every function call.

To facilitate the transformation of a process' memory image (specifically, the stack portion) we introduce a small program called the Stack Transformer (ST). The ST has three jobs:

1. creating the register state for the migrated-to core.
2. fixing all the return addresses on the stack.
3. repositioning the values of local variables in open function activations. The ST must ensure that the value of every stack-based variable is at the address where code for the migrated-to core expects it.

The ST may either be a part of the kernel or it may be user code that is invoked by the kernel during migration. Before we describe the operation of the ST, we will discuss how its prerequisite data is generated by the compiler.

4.5.1 Stack Transformation Prerequisites

The Stack Transformer (ST) needs detailed information about the compilation of the program from source code in order to prepare the stack for execution on the migrated-to core. We modify GCC to record the necessary information during compilation. This information includes

1. the frame layout for each function,

2. details about function call sites,

3. the locations of local variables,

4. the sets of spilled caller-saved and callee-saved registers.

Determining Frame Layout

The common frame layout is shown in Figure 4.2. We modify GCC to record, for every function, the sizes of every region within that function's frame. The ST reads in this information to determine where one frame ends and the next begins and where to look for data in the different regions within the frame.

Collecting Call Site Information

The ability to identify function call sites is essential for migration for two reasons: to determine where it is safe to migrate and to translate stack data during migration. Safe execution migration may only occur at a call site that exists in the code for both architectures because these call sites represent points of execution equivalence. Call site identification is important for stack transformation because the contents of the stack represent execution state at a sequence of function call sites. In order for the ST to reposition data within each frame, it must be able to identify and interpret stack frames.

Although we arrange function definitions to begin at the same address across code compiled for different ISAs (to avoid fixing function pointers during migration), the address of call sites are not the same. When translating state from ISA A to ISA B, the ST, given the sequence of return addresses on the stack for A, must find the corresponding call site locations in the machine code of B. This is necessary because the ST must transform return addresses on the stack. Also, it must look up the meta-data produced by the compiler for each call site—information like:

- the set of live registers at the call site (because the values of spilled registers may need to be moved),

- the names of the calling and called functions (so that frame size and layout may be determined).

In order to map between the two different code addresses of a call site and to look up additional call site information, we need a common key—a unique, architecture-independent label that is the same across compilations for different architectures. One option is to use a tuple of caller name, callee name, source file name, and source line number. However, there are many instances where multiple calls to the same function occur on the same line. This is exacerbated by the use of macros that may expand into a large body of code containing multiple function calls to the same function (all marked by the compiler as residing on a single line). Including column numbers in addition to line numbers is not always possible because many compilers (like GCC) do not keep column number information throughout the entire compilation.

A better solution is to assign each call site a unique identification number. These UIDs are assigned during the parsing stage of compilation because this stage is shared when compiling for different ISAs, so the numbering will be consistent. We modify GCC to create these UIDs and carry them along with each call site throughout compilation, even as different intermediate representations are used. During code generation, special code labels are created and inserted at each call site. The labels encode the source file name and call site UID. When the code is assembled and linked, these labels appear as symbols in the symbol table and are associated with the virtual address of each call site. The ST can inspect the symbol table to find the mappings between code addresses and call site UIDs. Also, the call site UIDs are referenced in the meta-data produced by the compiler, giving more information about each call site, including caller and callee names and live variable information.

Since call sites are given UIDs during the parsing stage, special care must be taken if call instructions are copied at some later point during compilation. Instructions may be duplicated for a variety of reasons by a number of optimization passes. Two examples are the function inlining passes and the basic-block reordering pass. To avoid ambiguity among copies, each cloned call site is given

a UID when it is created. Furthermore, the association to the original call site is reflected in the UID—some of the bits of the UID contain the UID of the original and some of the bits contain the copy number. We divide call site clones into two categories: clones created in the early compiler passes and clones created in the late compiler passes. Because the early passes are shared when compiling for multiple ISAs, clones created in these passes will be the same. Clones created in early passes tend to have an aspect of context sensitivity: the context of the cloned call site may be different. For example, when a call site is cloned due to function inlining (that is, when a function containing a call site is inlined into a caller and a copy of the call instruction is made) the cloned call resides in a different function. Because the context has changed, it would not be correct, upon return from the called function, to return to the original call site; it is only correct to return to the cloned call site.

Call sites cloned in the late passes, on the other hand, have an aspect of context-insensitivity. There is only one late pass that creates call site clones—the basic-block reordering pass. This pass performs software trace caching, arranging (and in some cases duplicating) basic-blocks to avoid branch mispredictions and instruction cache misses. The control flow is maintained; only the layout of blocks in memory is altered (based on performance-related predictions). It is therefore safe, upon return from a called function after migration, to return to any of the copies of that call site (though returning to the site of the same copy is likely to result in slightly better performance). This property of context-insensitivity is important in the late passes because clones made at this stage of compilation are not necessarily made consistently when compiling for different architectures. Thus, as long as we can track the association of clones made here to their originals, we can safely migrate at a cloned call site that may not exist for the architecture to which we are migrating.

Cross-Jumping Not only can call site identification be impaired by call site cloning, but it may also be hindered by optimizations that combine multiple call sites into one. The cross-jumping optimization is one such optimization. Automatically enabled at optimization level -O2 and above in GCC, this optimization

is designed for size efficiency, not performance. It combines common sequences of code; sometimes this includes instructions used for function calls—this happens frequently in MIPS code. A call to a function defined in another source file is implemented in two steps. First the address of the function is loaded into a register; then a *jalr* (jump and link register) instruction is executed to call the function whose address is in the given register. The cross-jumping optimization may combine multiple *jalr* instances into one, separating the function-address-loading instructions from the actual call instruction (adding unconditional jumps to the shared call instruction). This makes uniquely identifying these call sites difficult and makes it impossible for the ST to determine the true call site given just a return address (since the same return address would be shared by multiple call sites). We disable the cross-jumping optimization since it has virtually no effect on runtime performance in our experiments.

Locating Automatic Variables

In order for the ST to move the local variables of open functions to the correct stack locations for execution on the migrated-to core, it needs to know how to interpret the stack contents. It must be able to map high-level program variables (which are named identically among cross-platform compilations) to architecture-specific, compilation-dependent locations (either stack addresses or registers). All local variables located in the stack memory image must be located and identified by their high-level variable names. This section describes how the compiler gathers this information for the ST. The next section includes a brief discussion of optimization passes that make it difficult to locate stack variables that must be moved during migration.

The Effect of Inlining on Variable Names Normally, programming language rules make it unambiguous which object a variable name refers to in a given scope. However, as the compiler inlines functions it may create ambiguities if variables are identified by name alone. For example, if function *foo* contains a variable with name *var* and a call to function *bar*, which also contains a variable with name *var*,

then if the call to *bar* is inlined, there will be two variables in *foo* with the same name. GCC is able to differentiate between the two variables, but the ST, which only sees the variable names, could get confused. To solve this problem, every time we copy a variable declaration for inlining, we append a unique ID to the end of the name. This unique ID is the same across compilations for different ISAs, so the same variable is named consistently.

A local variable for an open function activation can be in a number of places. It may be allocated to a stack slot, allocated to a scratch register that is caller-saved, or kept in a callee-saved register. If a variable is allocated to a callee-saved register, then, at migration time, it could still be in the register (if no subsequent function activation uses the register), or it could be spilled in one of the records of a subsequent function activation (the first one that reuses the register).

In our implementation, this information is gathered and recorded by the compiler, and later supplied to the ST during a migration. The compiler records the location (either on the stack or in a register) of each variable, labeled by its source-level name. For every function, it records all the variables directly allocated to stack slots. It also records all the variables allocated to scratch registers that are caller-saved and the location on the stack where each caller-saved register is spilled. Finally, for every call site in every function, it records what variable is bound to each of the live callee-saved registers at that call site.

A single stack slot may contain the values of different variables at different times during the lifetime of a function. There are two reasons for this. First, nested lexical scopes within a function may contain variable declarations. Thus, two variables in different sub-scopes may occupy the same place in the stack at different times. The second reason for stack slot cohabitation is different caller-saved registers. If any scratch registers need to be preserved across a call site, they are spilled on the stack. At different call sites within a function, the same stack slot for caller-saves may be used to hold the values of different variables.

Since execution migration is possible at call sites, the ST needs to know precisely which variables are expected to be in which stack slots at every call site so that it can move the values of those variables to their correct, architecture-specific

locations. To handle multiple caller-saves to the same stack location, we modify the compiler to record the set of live scratch registers that are spilled into caller-saved stack slots at each function call site. Later, the ST consults this record to determine the contents of all the caller-saved stack slots in each activation record, in case any of these values need to be moved. Ambiguity resulting from stack slot cohabitation due to nested lexical scoping is handled by recording the scopes of variables that are allocated to the stack as well as the scopes of function call sites. During migration, the ST uses this scope information to disambiguate the local variables on the stack.

Just as it is necessary to track variables in caller-saved registers at every call site, it is also necessary to track variables in callee-saved registers across every call site because variables may be bound to different callee-saved registers across different function calls within the same calling function. We only need to track the variables that are live at each call site because these are the only variables the program expects to be valid when the process returns from the function call. If a migration took place in the interim (between the function call and its return), the process will be running different machine code under a different ISA that may expect to find those live variables in different locations.

Therefore, we add a new compiler pass to find the live callee-saved registers (and the source-level variable names bound to them) at every call site. The pass operates by first finding all the call sites by scanning the RTL representation of the program for call instructions. It then uses results of the data-flow analysis to identify the set of callee-saved registers that are live at the point in the code immediately following the call (i.e., the return point). GCC's data-flow analysis records the live registers at the end of each basic-block, not after every instruction. So we start with the set of live registers at the end of the basic-block containing the call instruction and scan backwards through every instruction up to the call site, updating the live register set along the way.

Optimizations that Interfere with Variable Location

A few late optimization passes in GCC expose new live registers across call sites. They do so in such a way that these new live registers do not always correspond to a high-level variable name; moreover, the transformation is not always applied uniformly across all architectures, resulting in architecture-specific state that cannot be translated at migration time. The incompatibility of these passes with migration is not inherent to the optimization algorithms, but an artifact of their implementation in GCC. To make these optimizations compatible with migration, one of two things may be done. First, the optimizations can be converted from RTL passes to tree passes because the tree intermediate representation is architecture-independent. Optimization can be applied consistently during compilation for both architectures and intermediate state that is exposed can be assigned labels that are carried in the intermediate representation until register assignment. Second, the optimization passes can be modified to avoid applying the optimization across call sites, as we do for the common subexpression elimination pass. How this is to be done depends on the operation of the optimization. If the optimization moves code, then code motion across call sites should be avoided. If the optimization deletes instructions, it should avoid deleting an instruction that exposes a new live register at any call site.

Both solutions—converting RTL passes to tree passes and introducing optimization barriers at call sites—may result in less than ideal performance improvement over the current implementations. But given the performance degradations with these optimizations completely disabled, we expect the performance trade-off to be minimal and acceptable. Either solution would required a significant refactoring of the code; so for this study, we have chosen to disable the problematic optimization passes. The optimization passes (all are RTL passes) are loop invariant motion, global common subexpression elimination, forward propagation, post-reload instruction scheduling, and tail call elimination. Disabling most of these passes results in negligible performance loss. For the benchmarks we use in this study, we observe some performance degradation when the following passes are disabled: global common subexpression elimination (less than 1%), forward

propagation (less than 1%), post-reload instruction scheduling (less than 0.5%), and temporary expression elimination (less than 1%). Turning off all these passes results in a performance loss of 3.1% in ARM code and 1.6% in MIPS code.

Special Register Contents

At call sites, live registers usually contain the values of variables, either variables named in the source code or temporary variables created in the early passes of the compiler. But occasionally a live register will contain something else. It may contain the value of a constant that is frequently used in the function or it may contain the address of a variable that is frequently referenced in the function. If it is an address, it may be the address of a global variable (residing in the global data section) or the address of a stack-based variable (a local variable residing in the stack frame of the active function). Whether the register contains a constant or an address, it may either contain the whole value or only the upper half of it, because sometimes large values must be loaded in two instructions, one that loads the high half and another that shifts the high half and adds the low half. We modify the compiler to report live registers containing constants and variable addresses, both complete values and upper halves. The compiler finds these live register values in the same way it finds live registers containing variable values—by traversing the data-flow graph and extracting register meta-data from RTL instructions. Information about live registers containing constants and variable addresses is later used by the ST when preparing register contents for the target ISA—either at that call site or the stack frame of a descendant function invocation that spills that register.

Locating Spilled Registers

Some local variables are not allocated to slots in the locals region of a function's frame. Instead, they are allocated to callee-saved registers. These registers are guaranteed to be preserved across function calls. To support this guarantee, if a function ever needs to write to a callee-saved register, it must first save the previous contents of that register and then restore those contents before returning.

Callee-saved registers that are used are spilled in a special region of the frame. The ST must find the location of every live variable in every activation record in case those values need to be moved—this includes variables in callee-saved registers that have been spilled to the stack in more recent activation records. Therefore, the ST needs information about which callee-saved registers are spilled by each function and at what offsets. Along with local variables bound to callee-saved registers, other important registers, like the return address register, may be spilled in the callee-saved register spill region. The ST needs to know the stack address where the return address is kept to determine which function the next stack frame corresponds to. Then it is able to correctly decode the next frame. We modify the compiler to record for the ST the set of callee-saved registers and their locations in every function's frame.

4.5.2 Operation of the Stack Transformer

Now that we have described the prerequisites to stack transformation, we describe the algorithm of the Stack Transformer, which is relatively simple once all the prerequisites have been met. The job of the stack transformer is to transform the architecture-specific program state (mainly stack data, but also register state) from ISA A to ISA B, so that the program running on ISA B will find all of its data after migration where it expects it. From a high level, the ST performs two quick passes over the call stack. The first pass goes from innermost frame (the frame in which execution was stopped) to outermost frame and finds values for caller-saved spill locations and simple stack-bound variables. The second pass works in the reverse direction—from outermost frame back to innermost frame—finding values for callee-saved spill locations and determining final register state. We describe each of these passes in turn.

The first pass examines and transforms each stack frame, from the most recently opened frame to the first frame created when the program was begun. When the ST begins, it is given the PC of the old core when it stopped at a call site. The ST looks up information about the call site (information recorded during compilation) using the PC. From this record it can tell in which function the call

site resides. The ST then looks up information on the function (also recorded during compilation) based on the function name and source file origin. The ST also looks up records for the same call site (using the call site UID) and function for the other ISA (the ISA of the core to which execution is migrating). With these four records—call site information and function information for each ISA—the ST goes to work transforming the stack frame.

First, from the call site record of the destination ISA, the ST processes the list of registers that are live across the call site. It looks for the values of these registers (by variable name) in three places in the records of the source ISA: (1) the list of live registers in the call site record, (2) the list of caller-saved spills in the call site record, and (3) the list of stack-bound variables in the function record. When checking the list of stack-bound variables for the current function, associated scope information for each variable may be used; the scope of the call site must be contained within the scope that the stack-bound variable is defined in for it to be a match. Once the ST has located the value that the register should have at the current call site, it copies the value to a list of live register values for the current function. It will keep such a list of live register values for every frame encountered, to be used later in the reverse call stack traversal. In some special cases, the value for the live register will be a constant, which would have been recorded during compilation, or the address of a global variable, which can be looked up in the program's symbol table.

Next, the values of variables in the caller-saved spill slots are found. The names and relative stack addresses of these variables are listed in the call site record for the destination ISA. To find the values for these variables, the ST checks the same three sources that it used to find values for the live registers on the destination core. Once found, the value of the caller-saved variable is copied to the appropriate stack slot.

Finally, the values of stack-bound variables listed in the function record for the destination ISA are found. The same three sources as before are checked, and when the value has been located, it is copied to the appropriate stack slot. It should be noted that only small variables whose addresses are never taken have

to be moved. Large variables on the stack (like arrays and structs) and variables whose addresses are taken are given identical stack positions by the compiler, so no copying of these values is needed and pointers to these variables will remain valid after migration.

When the ST is done processing a frame it will determine the live register state on the source core as it existed immediately before the current function was entered. It will carry over and use this register state when it transforms the next frame. It updates its view of the register state on the source core by reading from the callee-saved register spill locations in the current frame. The list of registers that are spilled (and the relative addresses to which they are spilled) on entry to the current function instance is found in the function record for the source ISA.

Since the ST knows the size of the frame from the function record, it can determine where the stack pointer would have been just before entry to the function. Also, since it knows where the return address was saved on the stack, it can read from this stack location and determine the address of the call site that led to the current function activation. With this information, the ST repeats the transformation procedure for the next frame up on the call stack, looking up the call site records and function records for the new call site and function context.

After the ST has passed over all the stack frames from innermost to outermost, it has enough information about the live register state at each call site to determine the values of the important callee-saved spill locations. The ST starts with live registers in the outermost frame (which it recorded in the previous pass) and moves to the next (inner) frame. For this frame, it consults the list of callee-saved registers and, for each one, copies the value from its view of the current register state to the appropriate location on the stack. Then it updates the current register state with the live register values at the next call site (which it recorded in the previous pass). The ST moves to the next frame and repeats the process until all the important callee-saved spill slots are full. The register state at the end of this process is the register state that should be instated on the destination core when the ST is done and the core is ready for native execution of the program.

There are two important things to note about this process. First, it handles the case where a live register at a particular call site is not saved immediately by the callee (because the callee doesn't recycle that register), but is instead saved in some distant frame or is never saved—with the value remaining in the register at the time of migration. This would not be possible with only one pass from innermost to outermost frame. Second, if the callee spills a register that is not live at the call site, the ST may not have a value to place in that spill slot; but this does not cause any problems after migration because when the called function is returned and the register is filled, it is not live, so it will not be read before being overwritten.

4.6 Experimental Methodology

This section describes our methodology. We begin by describing how we use two traditional compilers to model a multi-ISA compiler (a compiler with a unified front-end and multiple back-ends to simultaneously generate code for different ISAs). Then we describe how we measure performance. Finally, we describe how we measure migration overhead.

4.6.1 High-Level Pass Modifications

Ideally, when compiling for heterogeneous execution migration, the compiler should run all of its high-level, architecture-independent optimization passes once and then pass the high-level intermediate representation off to each architecture-specific back-end for further (architecture-specific) optimization and code generation. This model of compilation is both more efficient than separate compilation and necessary for correctness in migration. Unfortunately, the compiler we utilize, GCC, is not designed for such modular compilation. Rather than completely re-engineering GCC to conform to our ideal compilation model, we utilize two instances of GCC, each targeted to a different architecture, and work to harmonize the front-end of the compiler. Our goal is to rid the front-end of as much architecture-dependent code as possible so that the intermediate representation

of a given program right before the transition to the back-end is the same. In this way, we mimic the ideal compilation model, only sacrificing some efficiency during compilation. Making the front-ends identical is necessary (1) to ensure the same set of source-level variables make it to the back-end for assignment to architecture-specific locations and (2) to ensure consistent code motion across call sites.

Code motion across call sites occurs when computation is rescheduled across a call site. For migration at call sites to work, it is expected that identical units of computation would be done before the call site (before migration) on both core types. The form of the computation will differ (different machine instructions will be used), but the function (i.e., the semantic meaning) must be the same. Therefore, code motion is only permissible if it is applied consistently. We allow the front-end to apply code motion operations as it compiles for both architectures as long as those operations are the same. In the back-end of the compiler, only code motion between call sites, not across call sites, is permitted.

We modify GCC in a number of ways to enforce an identical intermediate representation (IR). Here we highlight some of the more important changes, including (1) making node numbering consistent to ensure identical compiler-created variable names, (2) using preprocessed source files to avoid architecture-specific code in macro expansions, and (3) disabling front-end optimizations that utilize architecture-specific information.

A significant source of difference between IRs is the numbering of nodes in the program's tree representation because the compiler uses these numbers to name new variables. Every tree node representing a declaration, type, or constant is given a unique ID number (from a simple counter) when it is created. When the compiler creates new variables, it bases their names on these UIDs. Thus, if more declarations, types, or constants are created when compiling for one ISA, then the counters that control the UIDs will be different and the compiler-created variables will be named differently. We make some modifications to GCC to ensure that these counters are consistent across compilations.

The use of identical preprocessed source files is also necessary to produce

identical IRs. Instead of starting with C files containing macros that may be expanded differently depending on the architecture-specific macro definitions in the C library header files and kernel header files that are included, we expand all the macros once and run the same preprocessed source files through each compiler.

GCC has a number of front-end compiler passes that peek at architecture-specific information to help guide decisions; this results in divergent intermediate representations when compiling for different architectures. One early optimization pass that relies heavily on architecture-specific information is the loop invariant optimization pass. This pass performs optimizations like strength reduction, induction variable coalescing, and induction variable elimination. It makes use of cost functions that estimate dynamic instruction counts to perform various operations. This pass has many places where architecture-specific knowledge is used, making it difficult to remove the dependencies without crippling the pass. We disable this pass and find no performance degradation for our benchmarks.

After all the modifications to GCC to enable an identical intermediate representation (at optimization levels -O0 through -O2), the runtime performance of the compiled programs is reduced by less than 1%.

4.6.2 Measuring Runtime Performance

For testing migration and measuring the runtime performance effects of compilation for migration, we use the SPEC2000 Integer benchmarks written in C (i.e., all but the C++ benchmark, *eon*). From this set of benchmarks we exclude the *gcc* benchmark because it uses the non-standard *alloca* C library function to dynamically allocate memory on the stack instead of on the heap—at this time our migration technique does not support variable-size stack frames. The ten benchmarks we use are *bzip2*, *crafty*, *gap*, *gzip*, *mcf*, *parser*, *perlbmk*, *twolf*, *vortex*, and *vpr*. All benchmarks are compiled with GCC at optimization level -O2. In all simulations the reference inputs are used.

Since we use the GCC compiler, which has front-ends for a number of different programming languages (including Fortran, C++, and even Java), it is theoretically possible to adapt our migration technique to work with programs

Table 4.1: Architecture detail for ARM and MIPS cores

ARM core			
Frequency	833 MHz	I cache	32 KB, 4 way
Fetch/commit width	2	D cache	32 KB, 4 way
Branch predictor	local	L2 cache	2 MB, 8 way
MIPS core			
Frequency	2 GHz	I cache	64 KB, 4 way
Fetch/commit width	4	D cache	64 KB, 4 way
Branch predictor	tournament	L2 cache	4 MB, 8 way

written in those languages. However, we have not thoroughly tested migration of programs written in languages other than C, so we do not show any results here for non-C programs.

To test the effects of migration-aware compilation we perform two kinds of tests. First, we perform tests to ensure correct execution. In these tests we use the M5 processor simulator [BDH+06] (configured to execute ARM binaries and MIPS binaries) to run each benchmark to completion. We verify that the outputs match the expected outputs. Second, we test performance using M5's cycle-accurate simulation mode on representative models of ARM and MIPS cores. The architectural model of the ARM core is based on the low-power Cortex-A8 core, while the MIPS core is modeled with performance as the primary design objective. The details of each core are given in Table 4.1.

Because simulating each benchmark in cycle-accurate mode is very time consuming, we simulate a portion of execution for each benchmark. To ensure that we measure the performance of the modeled cores on the same unit of work even after benchmark recompilation, we insert two marks in the source code of each benchmark—one to indicate where detailed simulation should start and one to indicate where it should stop. The first mark is made at the point in the code after approximately one billion instructions have passed (to skip over initialization code) and the second mark is made after approximately 500 million more instructions have passed. So the simulation interval is approximately 500 million dynamic instructions, the exact number of dynamic instructions depending on the ISA and

compilation options.

4.6.3 Measuring Migration Overhead

To measure the cost of migration, we cross-compile the Stack Transformer for both ARM and MIPS and run it in the M5 simulator on sample migration points taken from the benchmarks. We collect 10 samples for each benchmark in each direction of migration. The samples are collected at intervals of 100 million dynamic instructions starting one billion instructions into execution (to pass over the initialization phase). Each sample is of the first call site following the given instruction count (e.g., the first call site after 1.0 billion instructions, then the first call site after 1.1 billion instructions, etc). In a few cases migration at the nearest call site is not possible due to issues like executing in migration-unsafe library code; so in those cases, a nearby sample that is suitable is used.

The Stack Transformer is written in C++. Designed as a proof-of-concept, there is still plenty of room for optimization, and the performance results for the ST presented in this chapter should be considered conservative estimates of potential performance.

4.7 Results

This section quantifies the cost of our migration technique. Two critical characteristics of a good migration strategy are low runtime performance overhead (i.e., performance when no migration is occurring should be minimally impacted) and low migration cost. The following two sections quantify these characteristics. Since the frequency of migration opportunities may be important in some applications, a third section quantifies this.

4.7.1 Runtime Performance of Migratable Code

A key goal of this work is to enable fast migration without compromising runtime performance—that is, performance when no migration is occurring.

Throughout all the toolchain changes to ensure a nearly identical memory image across architectures, care must be taken that performance is not compromised. Among our benchmarks no performance was lost due to changes to make the memory image consistent (e.g., padding). There is, however, some performance loss due to optimizations in GCC that are disabled because they inconsistently move code across call sites (that is, call-crossing code motion is applied in only one ISA) and/or they impair the association of variable names with locations (preventing the ST from being able to reposition some data values). The specific changes that result in performance loss are described in Section 4.5.1. On average, runtime performance only suffers by 3.1% in ARM code and 1.6% in MIPS code.

4.7.2 Migration Cost

The cost of performing a migration is a combination of overhead from the involvement of the operating system and the execution of the Stack Transformer. Because programs are compiled with support for migration, most of the memory image is already prepared for execution on the destination core. The global data sections, the heap, and all the large variables on the stack do not need any transformations applied. Also all pointers will be valid without any transformation.

Since we do not have an OS for a heterogeneous-ISA CMP, we focus only on the migration cost incurred by the Stack Transformer. However, we expect the OS overhead to be relatively small, since its role is to add the thread to the run queue of another core and change some page table entries. In the rest of this section when we refer to migration cost we mean state transformation, not OS thread-scheduling activities.

For our benchmarks, state transformation (i.e., ST execution time) takes, on average, 234,651 dynamic MIPS instructions when migrating execution from ARM to MIPS and 239,426 ARM dynamic instructions when migrating execution from MIPS to ARM. Under our detailed architectural models for ARM and MIPS cores, the average migration times are 272 microseconds for migration from ARM to MIPS and 344 microseconds for migration from MIPS to ARM. Figure 4.3 shows the average migration costs for each benchmark for ARM to MIPS migration, and

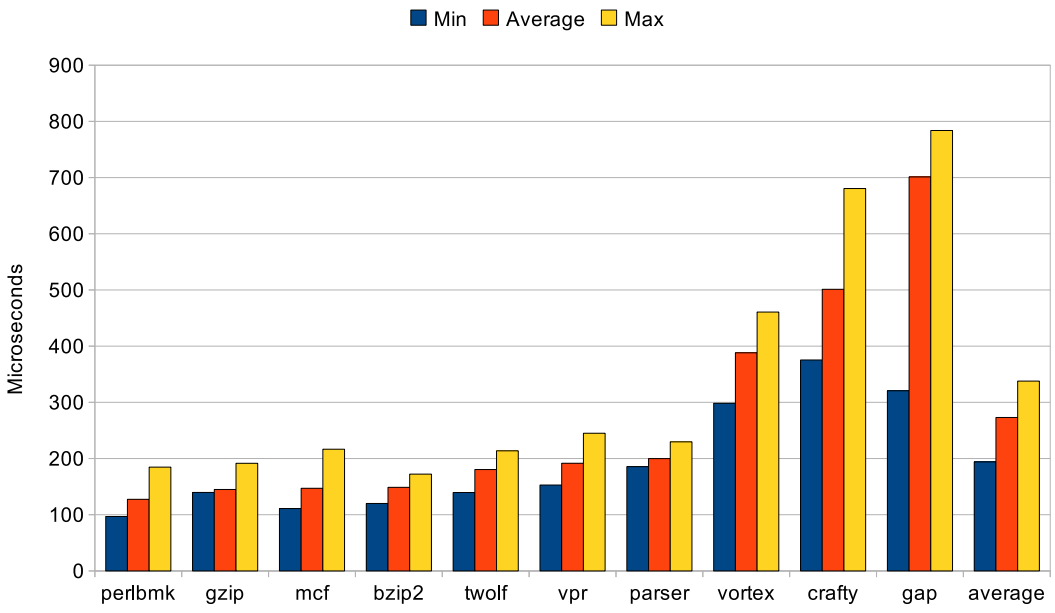

Figure 4.3: The average costs (measured in microseconds) of state transformation for migration from an ARM core to a MIPS core.

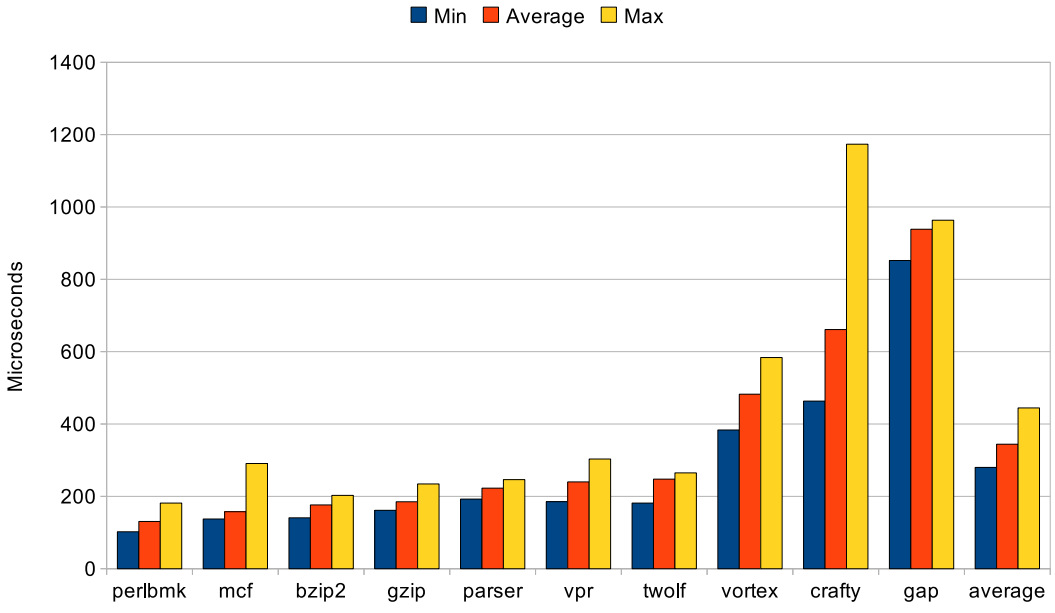

Figure 4.4: The average costs (measured in microseconds) of state transformation for migration from a MIPS core to an ARM core.

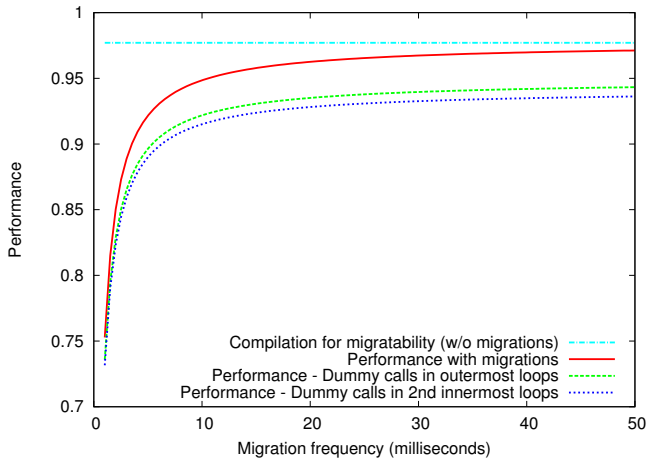

Figure 4.5: Performance vs. migration frequency.

Figure 4.4 shows the average migration costs for MIPS to ARM migration. The graphs also show the fastest and slowest migrations that we measure for each benchmark.

For migration in either direction, the migration overheads for the benchmarks *vortex*, *crafty*, and *gap* are the greatest. These three benchmarks also have the highest average stack depths. The average stack depth is the average number of frames on the call stack at any given point in execution. *Gap* has an average stack depth of just under 30. On the other end of the spectrum, *perl* has an average stack depth of three. The average stack depth across all the benchmarks was nine. Stack depth has the greatest influence on migration cost in our migration technique; the deeper the stack, the more state needs to be transformed. One consequence of this is that programs with highly recursive code have higher migration cost; highly iterative code is much less costly to migrate.

Figure 4.5 shows the relationship between performance and migration frequency. The performance results presented in this graph account for both performance costs due to compilation for migration (discussed in the previous section) and migration overhead. In this graph, migration frequency refers to how frequently migrations take place. For example, a frequency of 20 milliseconds means that every 20 milliseconds the program switches cores. We assume that we can

migrate at any time—that potential migration points (call sites) occur frequently. The straight line in the graph represents the performance if no migration occurs, and accounts only for the overhead of compilation for migratability. The line below that shows performance when migrations occur. When migrations happen every 10 milliseconds or less the effect on performance is significant. But above 10 milliseconds, performance remains above 95%, as compilation cost dominates migration overhead. The next two lines will be discussed in the next section.

4.7.3 Frequency of Potential Migration Points

Though our primary goals are fast migration and minimal performance impact (when not migrating), it is also sometimes desirable that when a migration is requested, the time until migration can begin is short, on average. This section quantifies the frequency of potential migration points.

For some use cases of migration, the expected amount of time to the next possible migration point is important—for example, if migration is triggered by frequent program phase changes or if a thermal emergency occurs and migration must take place before core failure. Our migration technique does not support instantaneous on-demand migration. Rather, migration is only possible at function call sites. To gain some insight into the expected time that a process would have to wait from the time migration is requested until the thread can migrate away from the current core, we measure the frequency of function calls in the benchmarks.

The distribution of function calls is highly irregular. During some phases of execution, function calls are frequent; in other phases, calls (and, therefore, migration opportunities) are much less frequent. Consequently, the average time between calls and the median time between calls are poor metrics for evaluating call frequency. Instead, we use as our metric the expected time to the next call. This is the average time (measured in dynamic instructions) until the next call site is reached from any randomly-selected point in execution.

The expected time to migration (ETTM) is computed as follows. Intuitively, if one considers the time from every discrete point in the program's execution (e.g., every dynamic instruction) to the next call and finds the average

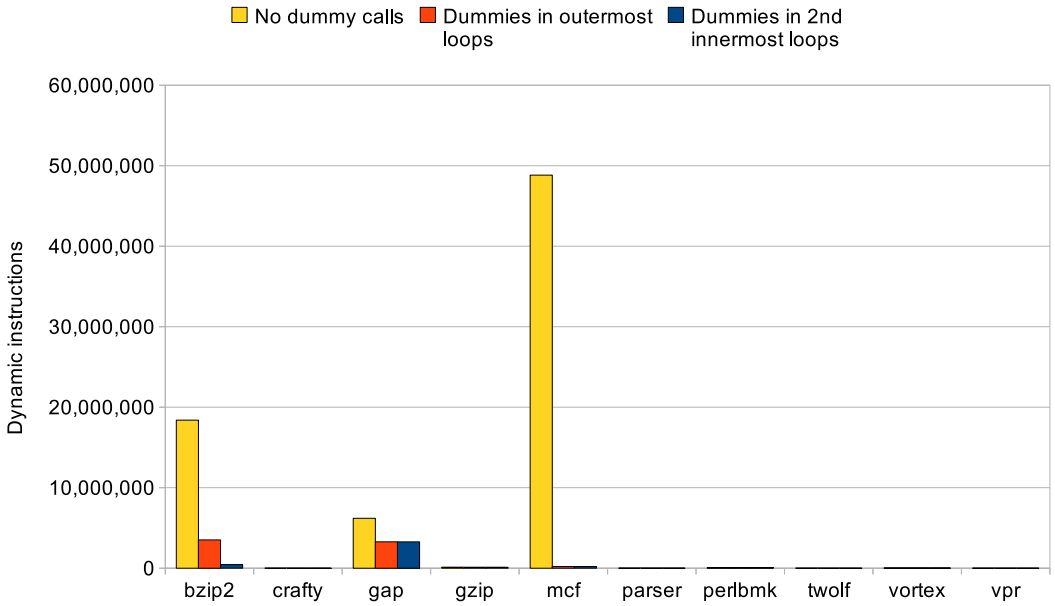

Figure 4.6: Among ARM binaries, the expected time to the next call, under three situations: no dummy calls have been added, dummy calls to outermost loops have been added, and dummy calls to second-innermost loops have been added.

of all of these times, this is the ETTM. More formally, it can be calculated in the following way. Let times $t_1, t_2, \ldots, t_{n-1}, t_n$ be the times between function call instances $c_1, c_2, \ldots, c_n, c_{n+1}$. For simplicity, assume that c_1 is not actually a call, but is the first instruction executed (since migration can easily take place here); likewise, assume that c_{n+1} is the last instruction of the program (since migration can easily take place here as well). Then let the program take time T to execute, where $T = \sum_{i=1}^{n} t_i$. If you randomly choose a point in time, $x : 0 \leq x < T$, then the ETTM is given by

$$ETTM = \frac{\sum_{i=1}^{n} \frac{t_i(t_i+1)}{2}}{\sum_{i=1}^{n} t_i} = \frac{T + \sum_{i=1}^{n} t_i^2}{2T} \tag{4.1}$$

Figures 4.6 and 4.7 (light bars) show ETTM for each benchmark as compiled for ARM and MIPS, respectively. From these graphs it is clear that call frequency varies dramatically across benchmarks. Among the ARM binaries, on one end of the spectrum is *vpr* where, on average, one must wait only 84 dynamic instructions until the next possible migration point. On the other end of the spectrum is *mcf*

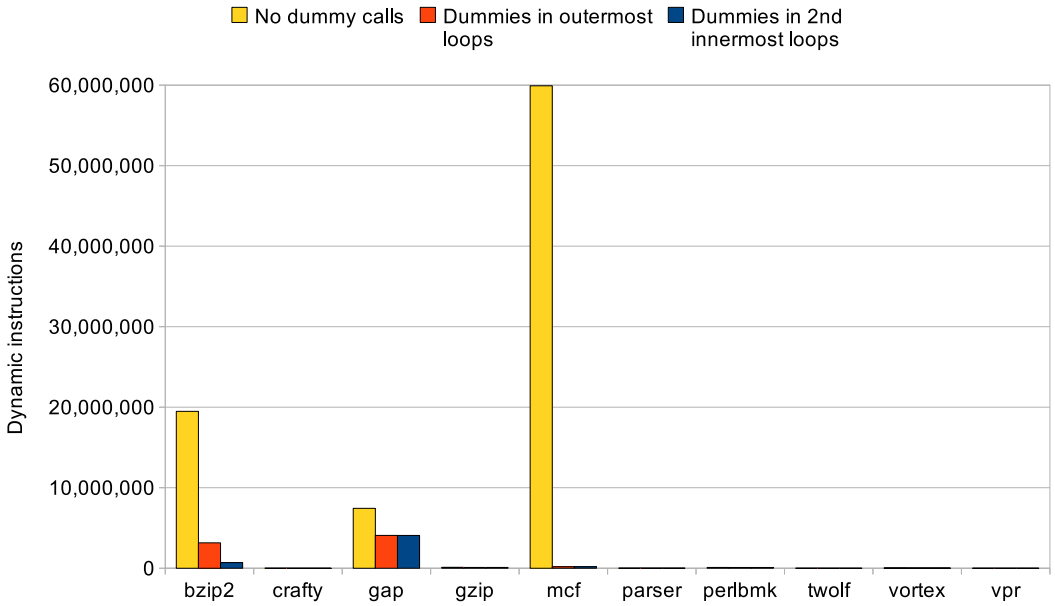

Figure 4.7: Among MIPS binaries, the expected time to the next call, under three situations: no dummy calls have been added, dummy calls to outermost loops have been added, and dummy calls to second-innermost loops have been added.

where, on average, one must wait over 48 million instructions until the next possible migration point. For seven of the ten benchmarks, the ETTM is under 130,000 dynamic instructions; but for *bzip*, *crafty*, and *mcf* the ETTM is much higher.

One way to increase the frequency of migration opportunities is to add more function calls. The long gaps between calls result from long-running loops that do not contain any function calls (or have had their function calls inlined). We modify GCC to inject dummy function calls (calls to an empty function that returns immediately) into loops. We experiment with two loop selection policies following two simple heuristics: inject dummy calls in outermost loops and inject dummy calls in second-innermost loops (parents of innermost loops). Neither policy selects innermost loops because the performance impact is too severe [FCG00].

The impact on ETTM of these policies is shown in the second and third bars in Figures 4.6 and 4.7. For the three benchmarks with the longest ETTM, inserting function calls in outermost loops dramatically improves call frequency. For example, in the MIPS binaries, the ETTM for *bzip* drops from over 19 million to

about 3.1 million dynamic instructions; for *gap*, the ETTM drops from about 7.4 million to about 4 million dynamic instructions; and *mcf* sees its ETTM reduced from almost 60 million to under 200,000 dynamic instructions. Using the second loop selection policy (second-innermost) results in only marginal gains. Most benchmarks only see a slight decrease in ETTM. *Bzip2* is an exception, dropping by a factor of 7.9 on ARM and 4.5 on MIPS.

The additional opportunities for migration that these changes bring comes at the cost of performance. Injecting dummy calls adds instructions that do not do any useful work and may interfere with some compiler optimizations. Performance drops 1.4% on ARM and 4.7% on MIPS when dummy calls are added to outermost loops. When calls are added to second-innermost loops, performance drops 2.3% on ARM and 5.4% on MIPS. For some programs (ones with infrequent calls) the performance degradation may be justified if the migration policy demands it.

The two lowest bars in Figure 4.5 show the performance of code compiled with dummy calls inserted. Migration overhead is the same, but due to the additional compilation costs, performance is lower at every migration frequency. If migration never occurs, performance remains below 95%.

4.8 Conclusion

In this chapter we present a new technique for execution migration in a heterogeneous-ISA CMP. This environment affords a unique opportunity for fast migration because migration overhead is not dominated by state copy since memory is shared and native to both cores. Our migration technique takes advantage of this opportunity by compiling programs to maintain memory state in a way that is nearly identical to its representation on every core type. As a result, migration requires only minimal transformation—only portions of the stack and register state need to be transformed. All pointers remain valid after migration without any transformation, eschewing the need for expensive pointer fixing during migration. We demonstrate that with strategic compiler changes, runtime performance does not have to be compromised for migratability—on average, non-migration perfor-

mance is reduced by 1.6% for MIPS and 3.1% for ARM. We show that the state transformation cost for migration is, on average, 272 microseconds for ARM to MIPS migration and 344 microseconds for MIPS to ARM migration.

Acknowledgments

The authors would like to thank Dean Tullsen for his guidance and support. Also, thanks to Ashish Venkat for his helpful comments.

Chapter 5

Conclusion

The shift in the microarchitecture industry to multicore architectures challenges us to consider how software will be able to capitalize on the increased hardware parallelism. Applications with inherent thread-level parallelism can be easily transformed into sets of homogeneous parallel threads and directly benefit from hardware parallelism. However, legacy binaries compiled for a single thread of execution cannot directly benefit from multicore systems, nor can applications that lack inherent TLP. We address the challenges to both of these types of applications in this dissertation.

To address the problem of legacy code achieving performance gains on multicore architectures we propose a new technique to automatically parallelize single-threaded code. Our solution transforms (dynamically) single-threaded binaries for which source code is not available. It incorporates a dynamic optimization framework to adaptively apply parallel transformations, selecting only beneficial transformations and dynamically tuning parallelization parameters. It leverages the optimistic concurrency of transactional memory to parallelize code that traditional parallelizing compiler cannot (or can, but not without serialization).

To address the problem of serial applications making efficient use of multicore resources, we look at how two forms of heterogeneity (workload-based and hardware-based) may be used to find the best core for each application. To support workload-based heterogeneity, we introduce a set of new scheduling algorithms for the complex topology of multithreaded multicore architectures. These architec-

tures pose an interesting challenge to thread schedulers, offering two dimensions of thread interaction: loose resource sharing among cores and tighter resource sharing among SMT contexts. The stakes for achieving efficiency and the number of potential schedules is greater than previous-generation single-core or simple multicore architectures.

Furthermore, we help pave the way for future schedulers on heterogeneous multicore architectures. We propose a technique for thread migration in a heterogeneous-ISA CMP. We exploit the fact that state copy is not necessary, since memory is shared, by compiling programs in such a way that very little state transformation has to be done to make them ready for execution on another core type.

In the next three sections we summarize our findings in each of these areas.

5.1 Runtime Parallelization

Chapter 2 presents a new runtime parallelization technique that leverages transactional memory and the runtime flexibility and efficiency of dynamic optimization. It has the following key features. It works on binaries with no source code available. It is completely automatic, requiring no assistance from the programmer or user. Parallelization at runtime combined with a dynamic optimization framework allows for parallel transformations to be applied adaptively. Poorly performing parallel code is discarded and parallelization parameters (like tile size) are tuned. Transactional memory facilitates the parallelization of code for which it cannot be proven that no memory aliasing exists—code that a traditional parallelizing compiler would be unable to parallelize without locks. It utilizes TM for register synchronization when no faster alternatives exist. No special-purpose hardware (hardware exclusively for parallelization) is required. We only assume a dynamic optimization framework (which has been shown to be useful for a variety of other optimizations [ZCT05]) and support for TM (which has more general use as a synchronization mechanism for parallel programs).

Our runtime parallelization results in a 36% performance improvement

across the NAS benchmarks and a 34% performance improvement across the SPEC2000 floating-point benchmarks, utilizing two-core parallelism. We show that a loop selection policy that considers only loops at a particular nesting level (e.g., innermost loops only) fails to achieve the highest performance. Nevertheless, the parallelization of innermost loops proves more profitable than the parallelization of outermost loops. We find that tile size, while important, plays less of a role in determining performance than other parameters, like TM granularity. We show that no single tile size is ideal for all loops; instead ideal tile size varies significantly among programs and also among individual loops. Finally, we evaluate the effectiveness of runtime parallelization at different TM granularities: word and cache line, showing that more significant gains can be achieved with word granularity.

5.2 Scheduling for a Multithreaded CMP

Chapter 3 introduces a new set of scheduling algorithms for CMPs with SMT cores, allowing applications that cannot be parallelized to benefit from multicore systems via unbalanced schedules. Multithreading multicore architectures are becoming increasingly common, and traditional schedulers fail to cope with the increased complexity. Traditional schedulers seek to balance load across available contexts. But in this chapter we show that unbalanced schedules, which a traditional scheduler would not even consider, are often best. Unbalanced schedules are especially necessary to achieve maximum efficiency in terms of performance *and* energy. We show, empirically, that most of the time the ideal schedule is an unbalanced one. It is necessary, therefore, for schedulers in these environments to consider both balanced and unbalanced schedules.

In a CMP of SMT cores, the ability to find a good schedule is critical to achieving high efficiency. But the search space of possible schedules is very large in this domain. Our scheduling algorithms intelligently navigate this huge search space, quickly converging on "good" schedules. The ability to quickly find good schedules means that they are able to adapt to changing program behavior, reacting to phase changes—traditional schedulers cannot do this.

The scheduling policies proposed in this chapter fall into two categories based on how they navigate the search space: sampling-based policies and electron policies (as we call them). The sampling-based policies work by trying out several proposed schedules at regular intervals and directly measuring power and performance during each trial. A number of different utility functions can be used to achieve the desired goal—whether it be performance, power, energy, or energy-delay product (EDP). The best sampling-based policy, *Prefer Last – Move*, results in an EDP savings of 7.8% with eight threads on a 4x4 architecture (four-core CMP, four SMT contexts per core).

Electron policies are based on the assumption that the previous schedule has some value. At regular intervals, the schedule is slightly modified as one core attracts a thread to itself and another core repels a thread away from itself. The criteria that a core uses to decide if it should attract or repel a thread depends on the goal function. The electron policy targeting EDP saving achieves a 10% savings in EDP with 12 threads on a 4x4 architecture.

5.3 Execution Migration

Serial applications can benefit even more from heterogeneous hardware, especially when that heterogeneity extends to the ISAs, as it is more likely that each application will have a core that is well-suited to its needs and executes it efficiently. But in order for scheduling on such architectures to be effective, running applications must be free to move among cores—to match phase behavior or environment changes (e.g., power state transitions).

Chapter 4 focuses on enabling dynamic scheduling in future heterogeneous architectures by introducing a technique for migrating execution among heterogeneous-ISA cores on a CMP. The migration technique relies on the build toolchain to ensure that program state is in a form that is as architecture-neutral as possible. The compiler, assembler, and linker work together to ensure that global data, heap data, and critical stack data does not need to be transformed for execution on a different core. In addition, function definitions begin at identical addresses and

pointed-to stack variables are assigned fixed addresses. Consequently, all pointers (whether they are to global variables, heap-based variables, stack-based variables, or functions) remain valid after migration. And not only is most state transformation unnecessary, but the copying of state is also unnecessary because the source and target of migration reside on the same chip and share the same memory—only some pagetable manipulation by the OS is necessary.

On average, state migration takes 272 microseconds for migration from an ARM core to a MIPS core and 344 microseconds for migration from a MIPS core to an ARM core. The major determining factor in migration time is stack depth—the deeper the stack, the more state needs transformation. It is important to note that stack variable size does not affect migration time because large variables on the stack (arrays, structs, etc.) are kept at fixed addresses and do not need transformation. Fast migration comes at a price though. As the compiler makes program state less architecture-specific and more generic, some runtime performance (performance when no migration is happening) must be sacrificed. Nevertheless, we demonstrate that with careful compiler modifications and by keeping performance-critical state in an architecture-specific form, performance loss is minimal: on average, 3.1% in ARM code and 1.6% in MIPS code.

Though the two primary goals of our migration technique are fast migration and minimal performance degradation, we also briefly evaluate the frequency of migration opportunities (which occur at function calls). This chapter introduces a new metric to fairly evaluate the frequency of migration opportunities. It measures the expected time to migration (ETTM) from a randomly-selected point in execution. We find that the ETTM varies significantly from one program to the next. Among our benchmarks, it is as high as 60 million dynamic instructions (for *mcf* on MIPS) and as low as 84 dynamic instructions (for *vpr* on ARM). To reduce the ETTM for benchmarks with high ETTM, we add dummy calls (which become migration opportunities) to loops. We evaluate two policies for selecting which loops to add the calls to: (1) outermost loops only and (2) second-innermost loops only. We show that the first policy is very effective at reducing ETTM in programs with high ETTM. For example, in *bzip* compiled for MIPS, the ETTM

is reduced by over a factor of six. We show that the second policy has a more limited impact—though for *bzip2* it is still helpful, reducing ETTM by a factor of 4.5 over the first policy.

Bibliography

[AAK+05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[BDB00] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 2000. ACM.

[BDH+06] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, July 2006.

[BGA03] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[BGW93] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1993.

[BMPS03] Greg Bronevetsky, Daniel Marques, Keshav Pingali, and Paul Stodghill. Automated application-level checkpointing of MPI programs. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 84–94, New York, NY, USA, 2003. ACM.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture*, June 2000.

[C-P01] C-Port Corp., 120 Water Street, N. Andover, MA 01845. *C-5 Network Processor Architecture Guide*, May 2001.

[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 *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 670–672, 2004.

[CLG00] W.K. Chen, S. Lerner, and R. Chaiken D.M. Gilles. Mojo: a dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization*, 2000.

[CMT00] Marcelo Cintra, José F. Martínez, and Josep Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 13–24, New York, NY, USA, 2000. ACM.

[CNV+06] Weihaw Chuang, Satish Narayanasamy, Ganesh Venkatesh, Jack Sampson, Michael Van Biesbrouck, Gilles Pokam, Brad Calder, and Osvaldo Colavin. Unbounded page-based transactional memory. *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, 41(11):347–358, 2006.

[CO03] Michael K. Chen and Kunle Olukotun. The Jrpm system for dynamically parallelizing java programs. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 434–446, New York, NY, USA, 2003. ACM.

[CS02] Kasidit Chanchio and Xian-He Sun. Data collection and restoration for heterogenenous process migration. *Software Practice and Experience*, 32(9):845–871, 2002.

[CTTC06] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[DJR01] S. Dutta, R. Jensen, and A. Rieckmann. Viper: A multiprocessor SoC for advanced set-top box and digital TV systems. *Design & Test of Computers, IEEE*, 18(5):21 –31, sep-oct 2001.

[DLMN09] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 157–168, New York, NY, USA, 2009. ACM.

[DRS89] F. B. Dubach, Robert M. Rutherford, and Charles M. Shub. Process-originated migration in a heterogeneous environment. In *CSC '89: Proceedings of the 17th conference on ACM Annual Computer Science Conference*, pages 98–102, New York, NY, USA, 1989. ACM.

[EMGAD06] Ali El-Moursy, Rajeev Garg, David H. Albonesi, and Sandhya Dwarkadas. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *Proceedings of the 20th international conference on Parallel and distributed processing*, pages 141–141, Washington, DC, USA, 2006. IEEE Computer Society.

[FCG00] Adam Ferrari, Steve J. Chapin, and Andrew Grimshaw. Heterogeneous process state capture and recovery through process introspection. *Cluster Computing*, 3(2):63–73, 2000.

[Fer98] Adam John Ferrari. *Process state capture and recovery in high-performance heterogeneous distributed computing systems*. PhD thesis, University of Virginia, Charlottesville, VA, USA, 1998. Adviser-Grimshaw, Andrew S.

[FPS06] Rohit Fernandes, Keshav Pingali, and Paul Stodghill. Mobile MPI programs in computational grids. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 22–31, New York, NY, USA, 2006. ACM.

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

[GG03] Soraya Ghiasi and Dirk Grunwald. Aide de camp: Asymmetric dual core design for power and energy reduction. Technical report, University of Colorado, Boulder, 2003.

[GPV04] Mohamed Gomaa, Michael D. Powell, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 260–270, New York, NY, USA, 2004. ACM.

[HD06] Paul H. Hargrove and Jason C. Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. *Journal of Physics: Conference Series*, 46(1):494, 2006.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[HM08] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33 –38, july 2008.

[HWC+04] L. Hammond, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, Honggo Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 102–113, June 2004.

[JSCT08] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 220–229, New York, NY, USA, 2008. ACM.

[KBH01] F. Karablieh, R. Bazzi, and M. Hicks. Compiler-assisted heterogeneous checkpointing. *Reliable Distributed Systems, IEEE Symposium on*, 0:0056, 2001.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589 –604, july 2005.

[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 *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[KT99] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, Sep. 1999.

[KTJR05] R. Kumar, Dean M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *Computer*, 38(11):32 – 38, November 2005.

[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 *International Symposium on Computer Architecture*, June 2004.

[LBK+10] Tong Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *HPCA '10: Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, pages 1 –12, Jan 2010.

[LBKH07] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 53:1–53:11, New York, NY, USA, 2007. ACM.

[LCYcH04] Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei chung Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6:2004, 2004.

[LTBL97] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of unix processes in the condor distributed processing system. Technical Report 1346, University of Wisconsin Madison, April 1997.

[MBM+06a] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *HPCA '06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265, 2006.

[MBM+06b] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 359–370, New York, NY, USA, 2006. ACM.

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

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation*, chapter 14. Morgan-Kaufmann Publishers, 1997.

[OYOB02] Kanemitsu Ootsu, Takashi Yokota, Takafumi Ono, and Takanobu Baba. Preliminary evaluation of a binary translation system for multithreaded processors. In *IWIA '02: Proceedings of the International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA'02)*, page 77, Washington, DC, USA, 2002. IEEE Computer Society.

[PBKL95] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: transparent checkpointing under unix. In *TCON'95: Proceedings of the USENIX 1995 Technical Conference Proceedings on USENIX 1995 Technical Conference Proceedings*, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association.

[PCT09] Leo Porter, Bumyong Choi, and Dean M. Tullsen. Mapping out a path from hardware transactional memory to speculative multithreading. In *PACT '09: Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2009. IEEE Computer Society.

[PE95] Bill Pottenger and Rudolf Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *ICS '95: Proceedings of the 9th international conference on Supercomputing*, pages 444–448, New York, NY, USA, 1995. ACM.

[PEL00] Sujay Parekh, Susan Eggers, and Henry Levy. Thread-sensitive scheduling for SMT processors. Technical report, University of Washington, April 2000.

[PL01] Sanjay J. Patel and Steven S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, 2001.

[PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*, pages 348–354, New York, NY, USA, 1984. ACM.

[RHL05] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.

[RS97] Balkrishna Ramkumar and Volker Strumpen. Portable checkpointing for heterogeneous archtitectures. *Fault-Tolerant Computing, International Symposium on*, 0:58, 1997.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 414–425, New York, NY, USA, 1995. ACM.

[SF98] K.-F. Ssu and W.K. Fuchs. Preaches - portable recovery and checkpointing in heterogeneous systems. *Fault-Tolerant Computing, International Symposium on*, 0:38, 1998.

[SFJ03] Kuo-Feng Ssu, W. Kent Fuchs, and Hewijin C. Jiau. Process recovery in heterogeneous systems. *IEEE Transactions on Computers*, 52(2):126–138, 2003.

[SH96] Peter Smith and Norman C. Hutchinson. Heterogeneous process migration: The Tui system. Technical report, University of British Columbia, Vancouver, BC, Canada, Canada, 1996.

[Shu90] Charles M. Shub. Native code process-originated migration in a heterogeneous environment. In *CSC '90: Proceedings of the 1990 ACM annual conference on Cooperation*, pages 266–270, New York, NY, USA, 1990. ACM.

[Sin08] R. Singhal. Inside intel next generation nehalem microarchitecture. In *Hot Chips 20*, Aug. 2008.

[SJ95] B. Steensgaard and E. Jul. Object and native code thread mobility among heterogeneous computers (includes sources). In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 68–77, New York, NY, USA, 1995. ACM.

[SKJC04] Alex Settle, Joshua L. Kihm, Andrew Janiszewski, and Daniel A. Connors. Architectural support for enhanced SMT job scheduling. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 63–73, 2004.

[SKTC05] Y. Sazeides, R. Kumar, D. M. Tullsen, and T. Constantinou. The danger of interval-based power efficiency metrics: When worst is best. In *Computer Architecture Letters, Vol 4*, January 2005.

[SPHC02a] T. Sherwood, E. Perelman, G. Hammerley, and B. Calder. Automatically characterizing large-scale program behavior. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[SPHC02b] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, New York, NY, USA, 2002. ACM.

[SR96] Volker Strumpen and Balkrishna Ramkumar. Portable checkpointing and recovery in heterogeneous environments. Technical Report 96-6-1, Dept. of Electrical and Computer Engineering, University of Iowa, June 1996.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[ST00] A. Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading architecture. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[STC00] John S. Seng, Dean M. Tullsen, and George Z.N. Cai. Power-sensitive multithreaded architecture. In *Proceedings of International Conference on Computer Design*, 2000.

[Str98] Volker Strumpen. Compiler technology for portable checkpoints. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology, 1998.

[STV02] Allan Snavely, Dean M. Tullsen, and Geoff Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIG-*

METRICS international conference on Measurement and modeling of computer systems, pages 66–76, 2002.

[Sun05] Sun Microsystems. Throughput computing faq:http://www.sun.com/processors/throughput/faqs.html, 2005.

[TB01] Dean M. Tullsen and J.A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *34th International Symposium on Microarchitecture*, December 2001.

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

[TEL95] Dean M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *22nd Annual International Symposium on Computer Architecture*, June 1995.

[Tex06] Texas Instruments Inc., PO Box 655303 Dallas, Texas 75265. *OMAP5912 Multimedia Processor Device Overview and Architecture Reference Guide*, February 2006.

[TH91] M.M. Theimer and B. Hayes. Heterogeneous process migration by recompilation. In *Distributed Computing Systems, 1991., 11th International Conference on*, pages 18–25, May 1991.

[THA+99] Jenn-Yaun Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[Tul96a] Dean M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[Tul96b] Dean M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *22nd Annual Computer Measurement Group Conference*, December 1996.

[vBSS94] David G. von Bank, Charles M. Shub, and Robert W. Sebesta. A unified model of pointwise equivalence of procedural computations. *ACM Transactions on Programming Languages and Systems*, 16(6):1842–1874, 1994.

[VP05] R. Veldema and M. Philippsen. Near overhead-free heterogeneous thread-migration. In *Cluster Computing, 2005. IEEE International*, pages 1–10, Sept. 2005.

[vPCC07] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–89, New York, NY, USA, 2007. ACM.

[VRR+07] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[YBM+07] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. Logtm-se: Decoupling hardware transactional memory from caches. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 261–272, Washington, DC, USA, 2007. IEEE Computer Society.

[YF08] E. Yardımcı and M. Franz. Dynamic Parallelization and Vectorization of Binary Executables on Hierarchical Platforms. *Journal of Instruction-Level Parallelism*, 10:1–24, 2008.

[ZCT05] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society.

[ZCT06] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 50–64, Washington, DC, USA, 2006. IEEE Computer Society.

[ZMLM08] Hongtao Zhong, Mojtaba Mehrara, Steve Lieberman, and Scott Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *In Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

[ZTC07] Weifeng Zhang, Dean M. Tullsen, and Brad Calder. Accelerating and adapting precomputation threads for effcient prefetching. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 85–95, Washington, DC, USA, 2007. IEEE Computer Society.