Leveraging Hardware Support For Transactional Execution To Address Correctness And Performance Challenges In Software

A DISSERTATION

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Ragavendra Natarajan

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Doctor of Philosophy

Antonia Zhai

May, 2015

© Ragavendra Natarajan 2015

ALL RIGHTS RESERVED

Acknowledgements

I'm a firm believer that most of what we accomplish in life is only in small part due to our efforts and largely due to the helping hands of others, as well as favorable circumstances. My PhD journey is no exception, and I'm indebted to far too many people who have helped me not only during my time as a graduate student but through various stages of my life preceding graduate school. Nonetheless, I will attempt to list the people who have had the biggest impact on me during my time here in Minnesota.

I am deeply grateful to my advisor Prof.Antonia Zhai for the opportunity and the helpful advice offered throughout my graduate career. I'm also indebted to Prof. Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, Prof. Pen-Chung Yew, and Prof. Wei Hsu for their fruitful collaborations at different stages during my PhD. I thank the fantastic support and facilities provided by the entire Computer Science department, and especially the systems staff and the office staff, throughout my time here.

Years from now when I look back at my time in graduate school I might not remember much of the research that I was part of during my time here. However, I'm certain I'll fondly recall the many friends I made and the times I shared with them. I am greatly thankful for the company and solace provided by my labmates, and the companions on my journey, Anup Holey, Vineeth Mekkat, Jieming Yin, Zhenman Fang, and Sanyam Mehta. I am grateful to my roommates with whom I have shared my life during my time here -

Raghuram Thiagarajan, Srinivas Rangarajan, and Venkat Durbha. Thank you for the wonderful company, the good times, and some of the best conversations I've had the pleasure of being part of. I'm also thankful for the thoroughly enjoyable company of the numerous friends I've made during my time here, of whom, I list only a few: Abhishek Sunku, Raveesh Shenoy, Renju Mohandas, Ramesh Raghupathy, Savio James, Prahaladh Iyer, Ranganathan Gopalakrishnan, Kandan Ramakrishnan, Sundar Eshwar, Ganesh Krishnamurthi, Suhail Ubaidar, Muthukumar Murugan, and Praveen Kumar. Of all the people I've befriended during the course of my life, my college friends, with whom I shared my formative years and who no doubt have had a significant impact in moulding me into the person I am today, are the dearest to me. In spite of the great distances between us in the intervening years, they have remained my warmest well wishers during my time as a PhD student. Vishal Chandrashekar, Shriram Jagannathan, Manuj Kumar, Sriharsha Niverty, Sivabalan Paneerselvam, Sathyajith Prabhu, Arun Radhakrishnan, and Srinivas Varadharajan - thank you for always rooting for me.

I hold my biggest gratitude to the most important people in my life; the people who have made numerous sacrifices in their lives just so I could strive for greater things in mine - my family. I will forever be grateful to my parents - Lalitha and Sivaram Natarajan - the two people who love me more than anyone else on the face of this earth. Even the few things I've accomplished in my life would not have been possible without you. And lastly, I thank the person who is singularly responsible for me embarking upon the PhD journey - my dearest sister Purnima Natarajan. Thanks Purni, for noticing my ability when I myself did not, for pushing me to aim higher and work harder, and for making sure I learnt from your life experiences. I owe all of this to you.

Dedication

Dedicated to the loving memory of Hiran Mayukh - the friendliest and most intelligent person I've known. You are gravely missed.

Abstract

Improvements in semiconductor technology and computer architecture have led to the proliferation of multicore and many-core processors. In order to improve the performance of multithreaded applications on multicore processors, hardware vendors have recently included support for transactional execution in the form of Hardware Transactional Memory (HTM) and Hardware Lock Elision (HLE). Under transactional execution, threads can speculatively execute in parallel and rely on runtime hardware to detect memory conflicts and rollback/replay execution if required. If an application does not encounter frequent memory conflicts among threads, then transactional execution can result in better performance, as compared to using mutex locks, due to the increased parallelism. Although primarily intended to improve multithreaded software performance, the introduction of hardware support for transactional execution presents exciting new avenues for addressing crucial research problems in a wider range of software. This thesis presents two novel applications of transactional execution to address performance and correctness challenges in software.

Most state-of-the-art processors implement relaxed memory consistency models in an attempt to extract more program performance. Different processor vendors implement different memory consistency models with varying memory ordering guarantees. The discrepancy among memory consistency models of different instruction set architectures (ISAs) presents a correctness problem in a cross-ISA system emulation environment. It is possible for the host system to re-order memory instructions in the guest application in a way that violates the guest memory consistency model. In order to guarantee correct emulation, a system emulator must insert special memory fence instructions as required. Transactional execution ensures that memory instructions within concurrent transactions appear to execute atomically and in isolation. Consequently, transactional semantics offers an alternative means of ordering instructions at a coarse-grained transaction level, and the implementation of hardware support for transactional execution provides an alternative to memory fences. This thesis tackles the correctness problem of memory consistency model emulation in system emulators by leveraging transactional execution support.

Extracting sufficient parallelism from sequential applications is paramount to improve their performance on multicore processors. Unfortunately, automatic parallelizing compilers are ineffective on a large class of sequential applications with ambiguous memory dependences. In the past, Thread-Level Speculation (TLS) has been proposed as a solution to speculatively parallelize sequential applications. TLS allows code segments from a sequential application to speculatively execute in parallel, and relies on runtime hardware support to detect memory conflicts and rollback/replay execution. No current processor implements hardware support required for TLS, however, the transactional execution support available in recent processors provides some of the features required to implement TLS. In this thesis, we propose software techniques to realize TLS by leveraging transactional execution support available on multicore processors. We evaluate the proposed TLS design and show that TLS improves the overall performance of a set of sequential applications, which cannot be parallelized by traditional means, by up to 11% as compared to their sequential versions.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computer technology has seen a tremendous improvement since the invention of the first electric computer. Advances in semiconductor technology coupled with improvements in computer architecture have resulted in powerful multicore and many-core processors since the turn of the century. Multicore processors aim to extract more performance, while consuming less power, compared to single core processors. In order to extract the power-efficient performance promised by modern multicore processors, it is crucial to extract sufficient parallelism from applications. Myriad programming and compiler tools have been proposed in order to help developers deploy multithreaded applications capable of harnessing the parallelism of modern multicore processors. Recently, multiple processor vendors have introduced hardware support for speculative transactional execution ,in the form of Hardware Transactional Memory (HTM) and Hardware Lock Elision (HLE), to further promote the development of multithreaded programs.

Transactional execution can be used as an alternative to traditional mutex locks in multithreaded programs. Transactional execution permits threads to speculatively execute in parallel, and relies on runtime hardware support to detect memory conflicts and rollback speculative execution if necessary. Therefore, transactional execution can extract more parallelism than traditional mutex locks when used in multithreaded applications which do not suffer from frequent memory conflicts among threads. Since its inception, transactional execution has been incorporated into a variety of multithreaded applications. Although intended to improve the performance of multithreaded programs, the advent of hardware support for speculative transactional execution presents new avenues of research. In particular, transactional execution semantics can be used to tackle problems in a broader range of software. This dissertation explores the application of hardware support of transactional execution to address correctness and performance challenges in software.

Most modern multicore processors implement relaxed memory consistency models in order to extract more performance. Consequently, current processors can re-order instructions in a concurrent application in a way that can produce results that are divergent from what the programmer intends. Hence, processor vendors provide special memory fence instructions that can be used by the programmer to prohibit the hardware from re-ordering instructions, and thereby guarantee correct execution of the program. Transactional execution enforces an implicit ordering among instructions at a coarse-grained transaction level by ensuring that memory operations within concurrent transactions appear to execute atomically and in isolation. Therefore, prior works have proposed transactional execution semantics as an alternative to memory fence instructions to enforce ordering of memory instructions in relaxed memory consistency model processors. The implementation of hardware support for transactional execution provides an alternate means to ensure correctness in software. In this dissertation, we employ transactional execution to address an important correctness challenge in cross-ISA system emulation software.

In order to take advantage of the processing power of multicore processors, it is crucial for sequential applications to extract sufficient parallelism. However, automatic parallelization of single-threaded applications with ambiguous data dependences remains a significant challenge. Prior research has proposed Thread-Level Speculation (TLS) as a solution for parallelizing sequential applications by relying on hardware support for speculative execution of threads. Although hardware support for TLS has not yet been widely adopted by the processor industry, recently implemented transactional execution support offers features similar to those required by TLS. Therefore, in this dissertation we explore the possibility of realizing TLS execution using HTM support that is available on existing microprocessors.

1.1 Challenges Addressed in this Dissertation

This dissertation explores novel applications of hardware support for transactional execution to address both *correctness* challenges in parallel system software, and *performance* challenges in sequential application software. We elaborate on the challenges addressed in this dissertation below.

1.1.1 Enforcing Correctness in Cross-ISA Emulators[1]

System emulation, or system virtualization, is a key technology that is widely used in today's computers. Data centers reduce costs be employing virtualization in order to utilize computational resources more efficiently. Virtualization also provides strong isolation between different applications running on the same hardware, thereby resulting in better security and reliability in the cloud. System emulation has numerous applications beyond cloud computing as well. Emulation is widely used as a safe way to examine malware. Emulation also facilitates execution migration of applications across different platforms and devices.

Support for emulation across processors with different instruction set architectures (ISA) can open up further opportunities in many different applications of system emulation. Cross-ISA emulation can help data centers to consolidate workloads over a wider range of processors. It can also enable new processor architectures to be deployed easily in data centers without any changes to existing applications. For example, x86-based applications can take advantage of servers built with emerging low-power processors with

[1]This work is set to appear in ACM Transactions on Architecture and Code Optimization [47].

different ISAs. Cross-ISA emulation also has potential applications beyond the data center. It can facilitate the execution of incompatible applications on desktop and mobile phones, as well as allow application execution to migrate between different devices seamlessly. For example, it can allow applications developed for ARM mobile processors to run on x86 mobile processors (and vice-versa). Cross-ISA system emulation can enable wider adoption of ubiquitous computing, which harnesses the cloud to run mobile applications, by supporting virtual execution of mobile applications on cloud servers with different ISAs.

Recent advances in semiconductor technology have resulted in multicore and heterogeneous multicore processors that drive systems from servers to mobile phones. In response to this trend developers are exploiting parallelism in applications. With parallel applications becoming more ubiquitous, cross-ISA virtualization of multithreaded programs is crucial. Although a large body of research exists on system virtualization, relatively few of the prior works address the challenges unique to multithreaded applications. One of the key challenges of virtualizing multithreaded applications across ISAs is ensuring that a program written for the guest system is executed correctly on the host system when the memory consistency models of the two ISAs differ.

The memory consistency model of a processor defines how the results of memory accesses in a program will appear to the programmer. The most intuitive memory consistency model is the *sequential consistency* (SC) model [37] which specifies that the memory operations from a processor appear to execute atomically and in the order they are specified in the program. Enforcing sequential consistency, however, prohibits a number of architecture optimizations crucial to high performance. Therefore, most modern processors choose to implement *relaxed* memory consistency models which are weaker than SC. However, they provide special memory fence instructions as a means to enforce SC. Table 1.1 shows the ordering constraints enforced in some modern processor architectures compiled from previous studies [1, 49, 42]. Different architectures vary in the ordering constraints they

Relaxation	W → R order	W → W order	R → RW order
SC			
x86-TSO	✓		
SPARC-TSO	✓		
SPARC-PSO	✓	✓	
SPARC-RMO	✓	✓	✓
POWER	✓	✓	✓
ARM	✓	✓	✓

Table 1.1: Relaxed memory consistency models of modern processors compared to SC. A ✓indicates the corresponding constraint is relaxed.

relax compared to SC. If the guest and host systems in a virtual environment have different memory consistency models, then it can lead to an incorrect execution of the guest application [58]. An emulated execution is considered incorrect if the order of memory operations that occurred during the actual execution on the host, could *not* have occurred on the guest system. Specifically, if the guest system has a stronger memory consistency model than the host system, it can result in the host machine reordering accesses in a way that is illegal on the guest ISA (e.g x86 on POWER).

Existing cross-ISA system emulators [3, 41] circumvent this issue by executing multithreaded programs sequentially - by emulating a multicore guest system through time-sharing using a single core on the host system. However, such emulators do not harness the power of multicore processors since they are not parallel. Recently proposed parallel emulators use multiple cores on the host system to emulate multicore guest systems [66, 16]. Consequently, they are much faster than sequential emulators. However, they only support emulation of guest and host systems with the same ISA (e.g. x86 on x86), or a guest system with a weaker memory consistency model than the host system (e.g. ARM on x86).

In order to ensure correct emulation when the guest system has a stronger memory consistency model than the host system, the ordering constraints of the guest memory model must be enforced on the host system by the emulator. One possible solution to the problem of memory consistency model emulation is runtime fence insertion in the translated host code. Transactional execution ensures that memory operations in concurrent transactions appear to execute atomically and in isolation. Therefore, it ensures an implicit ordering among instructions at a coarse-grained transaction level. The implicit ordering of instructions enforced by transactional execution can be used as an alternative approach to enforce an ordering among memory instructions in the translated host code. This dissertation explores the problem of supporting memory consistency model emulation in parallel system emulators in detail. We discuss the issues involved in supporting memory consistency model emulation, evaluate the tradeoffs between using the two alternate approaches, and propose a novel solution to address the problem.

1.1.2 Improving Sequential Application Performance Through Speculative Parallelization

Recent advances in semiconductor technology have resulted in powerful multicore and many-core processors. One way for single-threaded applications to benefit from this trend is to extract sufficient parallelism. Unfortunately, for a large class of applications with ambiguous dependences, automatic parallelization remains a significant challenge for software developers. Thread-Level Speculation (TLS) has been proposed as a solution to automatically exploit parallelism from sequential applications; and this technique has been studied extensively [23, 26, 2, 61, 43, 35, 21, 48, 59, 19, 12]. Under TLS, threads are speculatively executed in parallel. At runtime, data dependence violations are detected and speculative execution can be rolled back if necessary. A parallelizing compiler can leverage TLS support to speculatively parallelize sequential applications which contain ambiguous memory dependences [69, 68, 62, 17, 34, 33, 39, 64, 44]. No current processor implements hardware support for TLS, however, transactional execution support implemented in the form of Hardware Transactional Memory (HTM) [30, 55, 24, 27] in recent processors offers features similar to that of TLS.

Fundamentally, both HTM and TLS require efficient mechanisms for memory conflict detection and the rollback/replay when speculation fails. However, there are also significant differences between HTM and TLS. In particular, previously proposed TLS work has pointed out that ordered commit and synchronized inter-thread data communication are key for speculative parallelization of sequential applications [53]:

Ordered Commit: When sequential programs are parallelized under TLS, code segments from different parts of the program are speculatively executed concurrently. One way to ensure the preservation of the sequential semantics intended by the programmer is to force all the threads to commit in the same order as in the sequential execution. For example, TLS allows a loop with potential inter-thread data dependences to be parallelized by executing different iterations of the loop in separate threads. To preserve the sequential semantics under TLS, we must ensure that these threads are committed in the same order as in the sequential execution. Previously proposed TLS hardware ensures such ordered commit, however HTM does not provide such a guarantee.

Synchronized Inter-Thread Data Communication : While TLS provides an efficient mechanism for handling infrequently occurring data dependences, frequent data dependences are better handled through explicit inter-thread data communication. Thus, prior TLS proposals have explored such hardware support for synchronizing data between speculative threads [2, 26, 23, 43]. Unfortunately, existing HTM implementations do not have provisions for data communication between speculative threads. Data synchronization support is crucial to reduce frequent speculation failures in TLS.

Given these key differences between the existing HTM support and the hardware support required for efficient implementation of TLS, it is not clear if TLS can be implemented on existing multicore processors. Therefore, in this dissertation we aim to determine whether it is possible to realize TLS execution using HTM support that is available on current microprocessors. Implementing TLS on current processors can improve the performance of a large class of sequential applications with ambiguous data dependences.

1.2 Dissertation Contributions

This dissertation presents two novel applications of hardware support for transactional execution in state-of-the-art multicore processors. The thesis leverages transactional execution to address crucial correctness and performance challenges in software:

1. We study the problem of supporting memory consistency model emulation in parallel emulators and evaluate using transactions as an alternative solution to memory fences. The tradeoffs involved in using memory fences and transactions in the Intel Haswell processor are discussed and characterized. We implement the two approaches on COREMU, a recently proposed parallel emulator, and highlight the implementation issues. A novel hybrid technique that minimizes overhead by switching between using fences and transactions depending on the application characteristics is proposed. The overhead of the two approaches and the hybrid technique is evaluated on a set of parallel applications from the PARSEC and SPECOMP benchmark suites.

2. We implement TLS execution in the Intel Haswell microprocessor using hardware support for transactional memory. We propose software mechanisms to: i) ensure that speculative threads are committed in a predetermined order on current HTM, and ii) enable efficient inter-thread data communication between speculative threads on current HTM support. A novel dynamic tuning mechanism to prevent performance degradation by automatically disabling TLS in applications which suffer from

frequent speculation failures is proposed. The performance of TLS is evaluated using a set of SPEC2006 applications that are not amenable to parallelization using existing parallelizing compilers. Our evaluation shows that TLS yields a performance improvement of up to 11% compared to the sequential version.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows:

1. Chapter 2 describes the transactional execution support available on the Intel Haswell architecture which we use for all the evaluation studies in this thesis.

2. Chapter 3 describes the correctness problem that arises due to a discrepancy between the guest and the host system memory models in a cross-ISA emulation environment. It also discusses the tradeoffs between using memory fences and transactional execution as two alternative solutions to this problem, and compares the overhead of ordering instructions using memory fences and transactional execution on Intel Haswell.

3. Chapter 4 outlines how memory fences and transactional execution support can be incorporated into a parallel system emulator in order to support memory consistency model emulation. The chapter discusses implementation issues, and presents a detailed analysis of the two techniques, as well as our proposed hybrid emulation technique.

4. Chapter 5 illustrates how transactional emulation support can be utilized in order to realize TLS. We describe our software mechanism for implementing TLS, and software optimizations to further improve the performance of TLS.

5. Chapter 6 presents a detailed evaluation of the performance of our proposed TLS mechanism on a set of SPEC2006 applications which cannot be parallelized using

traditional techniques.

6. Chapter 7 concludes this thesis, and presents recommendations for features in future HTM implementations based on our experiences. The chapter also outlines possible future directions of research in transactional memory and speculative parallelization.

Chapter 2

Transactional Execution Support On The Intel Haswell Architecture

The idea of hardware transactional memory was first described over 20 years ago [30]. Although support for transactional memory has been introduced in specialized processors such as IBM's BlueGene/Q[29] and the discontinued Sun Rock processor [13], it has taken more than 20 years for transactional execution support to be implemented in commodity processors. With it's new *Transactional Synchronization Extensions* (TSX) instruction set, Intel recently introduced hardware support for transactional execution in the Haswell architecture. Concurrently, transactional memory support has been introduced in other commodity architectures, such as the IBM z/Architecture [32], and the IBM POWER [9].

In this chapter, we describe the hardware support for transactional execution offered on the Intel Haswell architecture. All the experiments presented in this thesis leverage the Intel TSX support. We describe the TSX instruction set in Haswell, and the features supported by TSX. Since the HTM implementations in other architectures offer similar support for store buffering, conflict detection, and execution rollback as the Intel TSX, we believe that the work described in this dissertation will be valid on other HTM implementations as well.

2.1 Intel Transactional Synchronization Extensions

The Intel TSX instruction set provides support for transactional execution in two different interfaces: *Restricted Transactional Memory* (RTM) and *Hardware Lock Elision* (HLE). Both these extensions rely on the same underlying architecture support for transactional execution. However, there are differences in the instruction set interface and the features supported by them. We discuss both these extensions in this section.

2.1.1 Restricted Transactional Memory

Intel's RTM instruction set provides instructions to enable programmers to transactionalize the execution of multithreaded software. The instructions provided by RTM are simple to use. A transaction is initiated using the XBEGIN instruction. Any memory read or write made after the XBEGIN instruction is buffered, and the changed addresses are tracked using per-thread read and write sets. The memory addresses are tracked at the cache line granularity. Memory changes made inside a transaction are committed at the end of the transaction using the XEND instruction. Data written within a transaction is not visible to the other threads until the transaction is committed using the XEND instruction. The XBEGIN and XEND are valid instructions within a transaction, thereby permitting transaction nesting in TSX.

There are several conditions under which a transaction can abort. On an abort, all the changes made within the transaction are discarded and the execution jumps to a fallback handler specified as an argument to the XBEGIN instruction. The reason for the transaction abort is recorded in the EAX register using an 8-bit flag. There are five flags defined in the architecture:

XABORT : An XABORT instruction aborted the transaction. An XABORT instruction can be used to explicitly abort a transaction. Execution jumps to the fallback handler specified in the XBEGIN instruction when XABORT is executed. XABORT takes an 8-bit failure code which is accessible by the fallback handler.

Conflict : The transaction aborted since an address modified within the transaction was read or written by another transaction.

Overflow : The transaction aborted since the number of memory addresses tracked in the read/write buffer exceeded the hardware limit.

Debug : The transaction aborted since a debug breakpoint was encountered.

Nested : The transaction aborted since a nested transaction failed.

There are a few instructions that are *restricted*within a transaction in the Intel RTM. If a restricted operation is attempted then the transaction is aborted and the fallback handler is invoked. Fundamentally, any instruction that changes the processor state in a way that cannot be trivially reverted causes a transaction abort. The restricted instructions include the multimedia extensions (MMX), streaming SIMD extensions (SSE), and the advanced vector extensions (AVX) instruction sets. Instructions that halt the processor's execution, change the privilege level of the execution, and cause exceptions are not permitted within a transaction. Despite these restrictions, the features offered by RTM are sufficient for most multithreaded application software.

2.1.2 Hardware Lock Elision

Intel TSX provides HLE as a solution to improve the performance of legacy lock-based multithreaded programs. HLE is based on prior research which proposed speculative lock elision [54] as a way to avoid a thread acquiring a lock before entering a critical section in a program if acquiring the mutex lock is not necessary at runtime. HLE is implemented in the form of two backward-compatible instruction prefixes: XACQUIRE and XRELEASE. These prefixes accompany atomic memory operations, such as `compare-and-swap`, that are typically used to implement mutex locks. The XACQUIRE prefix accompanies an atomic memory operation that acquires a mutex lock in the program, while the XRELEASE prefix accompanies an atomic memory operation that releases the mutex lock. If the processor supports the XACQUIRE and XRELEASE prefixes, then HLE is invoked and the execution is transactionalized. However, if the processor does not support transactional execution, then these prefixes are simply ignored and the critical sections are executed normally.

When an XACQUIRE-prefixed atomic store is executed, the processor implicitly starts a transaction at the lock boundary and *elides* the actual store, treating it as a transactional read instead (i.e., placing the cache line address of the lock variable in the read set). Internally, however, the processor maintains an illusion that the lock was acquired. Therefore, if the transaction reads the lock, it sees the value stored locally. Upon executing an XRELEASE-prefixed atomic store, the transaction commits and the lock is restored back to its original state. If an HLE transaction aborts, the XACQUIRE-prefixed store is re-executed *non-transactionally.* Such a non-transactional store conflicts with every concurrent HLE transaction eliding the same lock, since every such transaction is guaranteed to have the lock variable's cache line in its read set. This aborts all the concurrent HLE transactions which are then re-executed non-speculatively. An HLE transaction can abort due to the same reasons as an RTM transaction mentioned in Section 2.1.1. In all these cases, the HLE transaction is re-executed non-speculatively. Therefore, an HLE transaction can abort at most once before re-executing non-speculatively.

Both HLE and RTM rely on the same underlying hardware support to detect conflicts, track memory addresses, and rollback/replay execution. However, there are two major differences between RTM and HLE transactions.

1. In the case of an HLE transaction failure, the reason for the transaction abort is not visible to the programmer as in the case of an RTM transaction failure.

2. When using RTM, the programmer must ensure forward progress of the program by providing a fallback handler code which is executed in case of a transaction failure. The fallback handler can point back to the transactional code segment. However,

doing so does not guarantee the forward progress of the program. In the case of an HLE transaction, no fallback handler is required since the same code segment is automatically re-executed non-speculatively. Therefore, forward progress is implicitly guaranteed by the hardware.

Chapter 3

Correctness Challenges In System Emulation Across Different Architectures

System emulation, or system virtualization, of applications across processors with different instruction set architectures (ISAs) has many potential uses. System emulation is widely used in data centers for workload consolidation. Emulation also has applications in a wide range of areas such as malware analysis, application migration across platforms and devices, ubiquitous computing, and cross-platform software development. With parallel applications becoming more ubiquitous, cross-ISA virtualization of multithreaded programs is crucial. One of the key challenges of virtualizing multithreaded applications across ISAs is ensuring that a program written for the guest system is executed correctly on the host system when the memory consistency models of the two ISAs differ [58]. An emulated execution is incorrect if the order of memory operations that occurred during the actual execution on the host, could *not* have occurred on the guest system. Existing cross-ISA system emulators [3, 41] circumvent this issue by emulating a multicore guest system through time-sharing using a single core on the host system. Recently proposed parallel emulators are much faster since they use multiple cores on the host system to emulate multicore guest systems [66, 16]. However, they only support emulation of guest and host systems with the same ISA, or a guest system with a weaker memory consistency model than the host system. This chapter investigates the problem of supporting memory consistency model emulation in parallel emulators in greater detail.

In this chapter we begin by elaborating on the need for memory consistency model emulation using a motivating example. We then discuss two solutions to support memory consistency model emulation in parallel emulators: (i) using memory fences, and (ii) using transactional execution support. We then discuss the tradeoffs involved in using memory fences and transactions for memory consistency model emulation by characterizing the overhead of the two approaches on a recent processor. Our characterization shows that transactional emulation is a viable alternative to using memory fences for memory consistency model emulation. Moreover, the results show that a hybrid technique that intelligently employs both memory fences and transactions depending on the application characteristics is likely to yield the best results.

In the next chapter we propose our novel hybrid scheme, and present detailed analysis and results.

3.1 A Motivating Example

Consider the pseudocode shown in Figure 3.1 involving two threads (`Thread 0` and `Thread 1`) and two shared variables (`x` and `y`). `Thread 0` reads the value of `x` into a local variable `r1`, and writes to `y`, while `Thread 1` reads the value of `y` into a local variable `r2`, and writes to `x`. All the variables have an initial value of 0. Assume that the program is executed on an emulated x86 machine running on a POWER host system. Note that the x86 and POWER memory consistency models differ (Table 1.1). Table 3.1 shows all the possible values of `r1` and `r2` at the end of the execution of the program. It also indicates

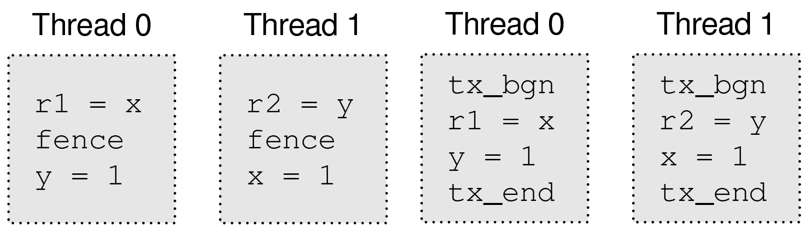
```
Thread 0      Thread 1

r1 = x        r2 = y
y = 1         x = 1
```

Figure 3.1: Pseudocode of an x86 guest application emulated on a POWER host system. All variables have an initial value of 0.

Result	x86-TSO	POWER
`r1` = 0, `r2` = 0	✓	✓
`r1` = 0, `r2` = 1	✓	✓
`r1` = 1, `r2` = 0	✓	✓
`r1` = 1, `r2` = 1	✗	✓

Table 3.1: Effect of the memory consistency model on the result of Figure 3.1

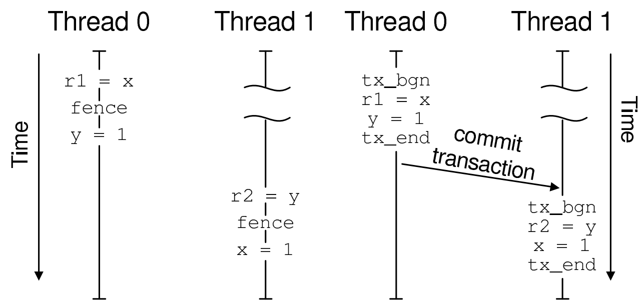
the outcomes that are valid under the x86 and the POWER memory consistency models. Under the x86 model, the final outcome of `r1` = 1 and `r2` = 1 is illegal since the outcome requires the stores to `x` and `y` to be reordered before the loads to `r1` and `r2` in both the threads, which is not possible since the x86 model ensures that stores are not reordered with preceding loads (R$\rightarrow$W order is not relaxed). However, all the possible outcomes are valid on POWER since the memory consistency model does not guarantee any ordering among the memory accesses. Hence, the virtualized x86 system can observe an illegal result (`r1` = 1 and `r2` = 1). Therefore, in a cross-ISA virtualized environment, if the guest system has a stronger memory consistency model than the host system, it can lead to an incorrect execution. However, if the guest system has a weaker memory consistency model than the host system (e.g. POWER on x86), then the execution is guaranteed to be correct.



```
Thread 0        Thread 1        Thread 0        Thread 1

r1 = x          r2 = y          tx_bgn          tx_bgn
fence           fence           r1 = x          r2 = y
y = 1           x = 1           y = 1           x = 1
                                tx_end          tx_end
```

(a) using fences (b) using transactions

Figure 3.2: Correct x86 emulation on the POWER host system.



(a) (b)

Figure 3.3: A correct execution of translated POWER host code (a) without the need for memory fences, (b) without any transaction aborts.

3.2 Memory Fences

In order to ensure correct emulation when the guest system has a stronger memory consistency model than the host system, an emulator must insert memory fences in the translated host code at runtime. For the example shown in Figure 3.1, a memory fence must be inserted between the load to `r1` (`r2`) and the store to `y` (`x`) in the translated POWER host code by the emulator. The memory fence ensures that the load and store in a thread do not get reordered. Figure 3.2(a) shows the pseudocode of the correct translated host code.

Finding a correct and efficient placement of memory fences for a program is a challenging task [8, 36, 20, 18]. Inserting fences conservatively results in redundant fences and degrades the performance of the program, while using too few fences can cause incorrect emulation.

Even if the number of fences inserted, and their placement, is optimal, previous studies show that a large fraction of the inserted memory fences are in fact unnecessary at runtime [65, 38]. Figure 3.3(a) shows an execution of the POWER host code translated using fences from Figure 3.2(a). Here `Thread 0` completes its accesses, and its effects are visible to `Thread 1`, before `Thread 1` executes its own accesses. In this execution, the final result is legal on the x86 guest even without any fences inserted in the translated code, since, even if the accesses made by both threads are reordered on the POWER host system it will not lead to a consistency violation.

3.3 Transactional Execution

Transactional execution support implemented in recent processors provides an alternative method of ensuring correct emulation of a guest system on a host system with a weaker memory consistency model without the use of memory fences. HTM or HLE can be used to group the accesses made by the translated host program into coarse-grained transactions. Hardware support ensures that all memory accesses within a transaction appear to execute atomically and in isolation. It also guarantees that all the transactions executed by the same thread are sequentially ordered. Therefore, transactional emulation guarantees sequential consistency at the coarse-grained transaction level. Consequently, all the memory accesses made by the guest application on the host system are also sequentially consistent. Enforcing sequential consistency on the host machine ensures that the emulated execution is guaranteed to be correct on any guest memory consistency model. Note that the granularity of the transactions does not affect correctness although it can impact performance, and that the accesses within a transaction can be reordered while still appearing to conform to sequential consistency.

Although transactional emulation enforces a stricter constraint than necessary, it can outperform emulation using memory fences under certain conditions. Unlike fences, which incur a fixed cost on every execution, the cost of a transaction varies depending on the abort rate. If there are no conflicts between the threads during execution, then all the transactions will commit without any aborts. Figure 3.3(b) shows a conflict-free execution of the POWER host code translated using transactions from Figure 3.2(b). In this execution, `Thread 0` commits its transaction before `Thread 1` begins executing its own transaction. Therefore, there is no conflict between the transactions and both commit without any aborts. In this execution, the accesses within a transaction can be reordered on the POWER host and the execution would still be correct. The transactional version of the translated code can outperform the fence version since it does not incur the overhead of executing fence instructions.

Transactional emulation can also result in poor performance under certain conditions. Small transactions cannot effectively amortize the overhead of starting and ending a transaction. Thus, they can result in poor performance. However, increasing the transaction size beyond a certain limit leads to diminishing returns. Large transactions can result in conflicts among memory accesses that are well separated in time and cannot lead to consistency violations in the guest application. Such false conflicts can increase the abort rate of the transactions, thereby resulting in poor performance.

3.4 Overhead Characterization

In this section we characterize the overhead and tradeoffs between using memory fence instructions and transactions on a recent processor. Our test system is a 4-core, 4-thread x86 Haswell processor with HTM and HLE support. The features of the processor and the transactional execution support are described in Chapter 2. Our evaluation does not characterize the hardware parameters of the transactional execution support implemented in Haswell since this has already been done by previous work [56]. We use HLE to implement our transactions (lock elided critical sections). We begin by comparing the overhead of memory fences and transactional execution in the absence of aborts using a simple single threaded micro-benchmark. We then evaluate both the correctness and the performance

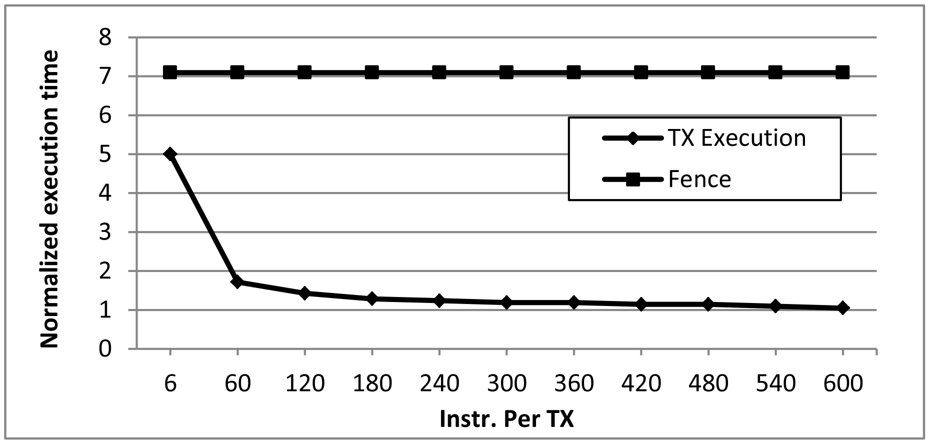


Figure 3.4: Execution times of the fence and transactional versions of a sequential micro-benchmark normalized to that of a no-fence and no-transaction baseline across different transaction sizes.

tradeoffs of memory fences and transactional execution using a a set of concurrent, lock-free algorithms.

Overhead: Fences vs. Transactions

We use a single-threaded micro-benchmark to compare the overhead of memory fences and conflict-free transactional execution on Haswell. The micro-benchmark consists of a single loop that iterates 100 million times. Each iteration of the loop performs a store to a memory location, followed by a load from a different memory location. Therefore, the store and load in this micro-benchmark might be executed out of order on x86. We design two versions of the micro-benchmark where this re-ordering is prevented. In the fence version we insert a fence between the store and the load, while the transactional version executes each iteration of the loop within a transaction. Note that since the micro-benchmark is sequential, there are no aborts due to memory conflicts in the transactional version. Since the loop accesses only a few cache lines, the transactional version does not experience any aborts due to buffer overflows either. We vary the size of each transaction in the transactional version by varying the number of loop iterations executed within each transaction, while keeping the total number of loop iterations constant.

Figure 3.4 shows the execution time of the fence and the transactional versions of the micro-benchmark normalized to the baseline which does not enforce any ordering. The data is shown for various transaction sizes. Each iteration of the loop contains 6 instructions and we vary the number of iterations within a transaction in steps of 10. The results show that the overhead of memory fences on x86 is considerably high. The overhead of transactional execution, on the other hand, varies depending on the transaction size. When the transaction size is small, the overhead of transactional execution is considerable. However, even at a small transaction size the overhead of using memory fences is much higher. As the transaction size increases, the overhead of transactional execution is amortized and performance improves. Once a large enough transaction size is reached, the overhead of transactional execution is negligible and the performance is comparable to sequential execution. Increasing the transaction size beyond this optimal size does not lead to any performance benefit. These results demonstrate that memory fences are expensive instructions on x86. They also highlight that using transactional execution to enforce memory ordering, instead of memory fences, can lead to substantial performance benefits if the abort rate is low and the transactional overhead is amortized.

Concurrent Micro-Benchmark Results

In order to evaluate both correctness and the performance tradeoffs, we use a set of concurrent, lock-free algorithms which are written assuming SC. Thus, these micro-benchmarks require memory fences for correct execution on x86 machines. All these algorithms enforce mutual exclusion among threads in a multi-threaded program, using only shared memory variables for communication. Each algorithm describes an entry region, which is executed by a thread prior to entering the critical section, and an exit region, which is executed by a thread once it exits the critical section. We briefly describe the kernels below.

- **Peterson's algorithm**: A well known algorithm [52] for enforcing mutual exclusion in a multi-threaded program. The algorithm requires 1 fence in the entry region code for correct execution on the x86 ISA.

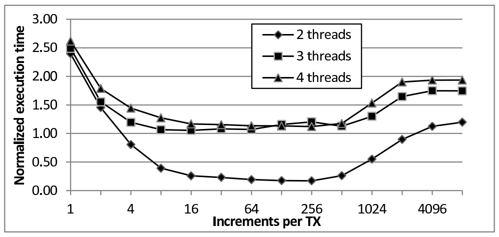
- **Big-Reader lock algorithm (BR-lock)**: A reader-writer lock implementation [7] originally proposed and used in the Linux kernel. The algorithm requires 2 fences, both in the entry region code, for correct execution on the x86 ISA.

- **Byte-lock algorithm**: Another reader-writer lock implementation proposed in [14]. The algorithm requires 2 fences, both in the entry region code, for correct execution on the x86 ISA.

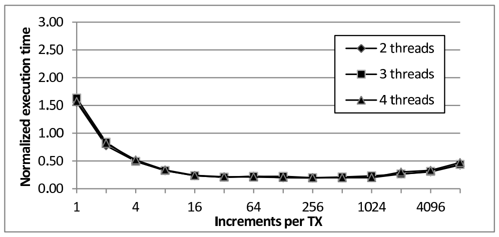
- **Dekker's algorithm**: A well known algorithm [15] for enforcing mutual exclusion among 2 threads. It requires 2 fences in the entry region code for correct execution on the x86 ISA.

Each kernel is a simple program where multiple threads compete simultaneously to increment a shared variable using a mutex lock implementation listed above. Each thread increments the shared variable a fixed number of times in a loop. One iteration of the main loop involves executing the entry region code, incrementing the shared variable, and executing the exit region code. The threads do not wait between successive increments and therefore, these programs have high contention. We check for correctness by testing the value of the shared variable at the end of program execution to confirm that there were no violations of mutual exclusion. Two versions are implemented for each kernel: a fence version that uses memory fences, and a transactional version (with no fences). In the transactional version of the program, each iteration of the main loop is performed as a single transaction by a thread. We vary the size of a transaction by varying the number of iterations executed within a transaction, while keeping the total number of iterations constant. The number of iterations is varied by unrolling the main loop as many times within each transaction. Note that the entry and exit region codes are executed as many times as the number of increments of the shared variable in each transaction. Although the kernels are not representative of real-world applications, they are useful in order to simulate the conditions under which transactional execution can outperform fences on a real machine.

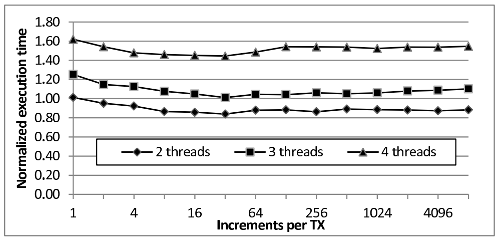
Effect of transaction size: Figure 3.5 (a, c, e, g) shows the execution time of the transactional version of each program normalized to the fence version, across different transaction sizes. The data is shown for 2, 3, and 4 threads. Only two thread results are shown for Dekker's algorithm since it cannot be implemented for more than 2 threads. We choose the fence version as the baseline in order to compare the relative performance of memory fences and transactional execution. Since the micro-benchmarks encounter a livelock in the absence of memory fences, we do not choose micro-benchmarks without fences as the baseline. For all the programs, as the transaction size increases, the performance improves until an optimal size and then begins to drop. Larger transactions amortize the overhead of starting and ending a transaction thereby resulting in better performance. However, very large transactions also increase the possibility of conflicts between the threads, which in turn increases the abort rate of the transactions. A large transaction can also fail if the number of unique cache lines accessed within the transaction exceeds a hardware specific maximum read/write size [56]. However, this phenomenon is not observed in the evaluated kernels since each of them accesses just a few unique cache lines within a transaction. Peterson's, BR-lock, Byte-lock and Dekker's kernels access 3, 3, 2 and 3 unique cache lines within a transaction, respectively. Therefore, the transactions in these kernels abort only due to data conflicts resulting from the increase in the number of instructions (loads/stores) per transaction. Some of the drop in the performance at very large transaction sizes is also due to the aggressive loop unrolling necessary to increase the transaction size. The 2 thread results show that the transactional version, even with a suboptimal transaction size, is faster than the fence version. The execution time of the optimal transactional version of Dekker's, Peterson's, BR-lock, and Byte-lock, with 2 threads, is 0.05, 0.17, 0.88, and 0.83 times the execution time of the fence version, respectively.



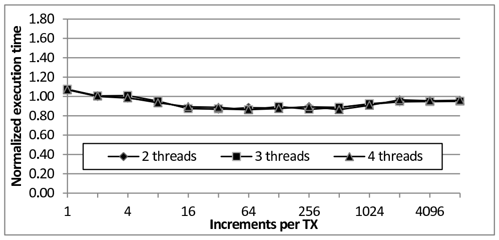
(a) Peterson's algorithm with high contention



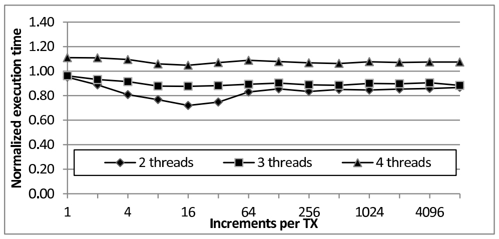
(b) Peterson's algorithm with no contention



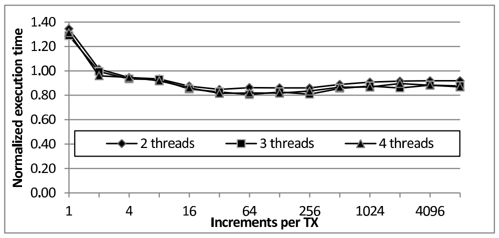
(c) BR-lock algorithm with high contention



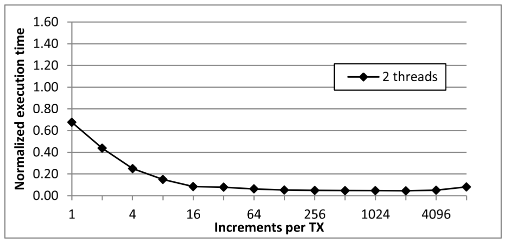
(d) BR-lock algorithm with no contention



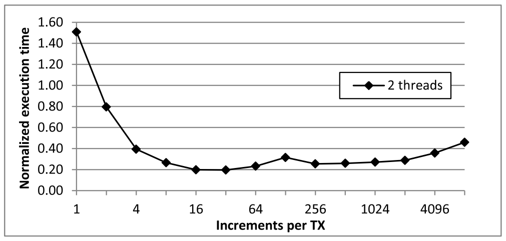
(e) Byte-lock algorithm with high contention



(f) Byte-lock algorithm with no contention



(g) Dekker's algorithm with high contention



(h) Dekker's algorithm with no contention

Figure 3.5: Execution time of the kernels using transactions normalized to execution time using memory fences under low and high contention for different transaction sizes.

Effect of conflict rate: All these programs have a high conflict rate between the threads, and as we increase the number of threads it further increases the possibility of a conflict. A high conflict rate increases the abort rate of the transactions, thereby leading to poor performance. Figure 3.5 (a, c, e, g), shows that the performance of the transactional version drops significantly compared to the fence version for 3 and 4 threads. The execution time of the optimal transactional version of Peterson's, BR-lock, and Byte-lock, with 4 threads, is 1.12, 1.45, and 1.05 times the execution time of the fence version, respectively. In order to see the performance of transactional execution when there are no conflicts, we modified the kernels (both the fence and the transaction versions) such that each thread operates on a private lock and increments a private variable. Since the transaction support on Haswell tracks dependencies at the cache block level, false sharing among threads can also result in conflicts. Therefore, we take care to place all the private locks and variables on different cache blocks so as to eliminate any false sharing. Figure 3.5 (b, d, f, h) summarizes the results for all the kernels with 2, 3 and 4 threads. The results show that the performance of the transactional version gets better as the transaction size increases. However, under no contention, there is no drop in the performance of the transactional version at large transaction sizes. The dip in performance observed in the kernels at very large transaction sizes is due to the aggressive loop unrolling required to generate large transactions. Moreover, the performance does not vary with the number of threads when there is no contention. The execution time of the optimal transactional version of Dekker's, Peterson's, BR-lock, and Byte-lock, with 2 threads, is 0.2, 0.2, 0.88, and 0.85 times the execution time of the fence version, respectively. The corresponding numbers with 4 threads for Peterson's, BR-lock, and Byte-lock are 0.2, 0.86, and 0.82, respectively. Even as the number of threads increases, the transactional version is faster than the fence version.

These results show that transactional execution is a viable alternative to using fences in order to emulate a stronger guest memory consistency model on a host with a weaker memory consistency model. If the transaction sizes are large enough to amortize the transaction overhead, and the conflict rate among the threads is low, then transactions can outperform fences. However, if the transactions are too small, or if the program has a high conflict rate, then emulation using memory fences can result in better performance. Therefore, a hybrid technique that can intelligently employ transactions or memory fences for emulation depending on the application characteristics will likely yield the best performance. These characterization results lead us to propose a novel hybrid memory consistency model emulation technique that uses both memory fences and transactions in Chapter 4.

Chapter 4

Leveraging Transactional Execution For Memory Consistency Model Emulation

Chapter 3 described the problem of memory consistency model emulation support in parallel emulators in detail, and proposed transactional emulation as an alternative to run-time fence insertion as a solution to the problem. The characterization in Chapter 3 also demonstrated the conditions under which transactional emulation is a viable alternative to memory fence insertion.

This chapter delves into the issues involved in incorporating the two approaches in order to support memory consistency model emulation on a parallel system emulator. Based on the characterization results in Chapter 3, we propose a novel hybrid emulation technique that uses both fences and transactions, depending on the characteristics of the emulated application, in order to minimize the overhead . The three approaches are evaluated on COREMU, a recently proposed parallel emulator, using a set of real world parallel applications from the PARSEC and the SPECOMP benchmark suites, and a detailed analysis of the results is presented.

4.1 System Emulation Using Dynamic Binary Translation

System emulators commonly use dynamic binary translation to convert guest assembly instructions to host instructions. The guest code is translated on-the-fly, one basic block at a time. Once a basic block has been translated, it is executed on the host system and the emulator then begins translating the subsequent basic block. Emulators use a *translation cache* to store recently translated *translation blocks*. When translating a guest basic block, the emulator first searches for a corresponding translation block in the translation cache. On a cache miss, the guest block is translated and inserted into the translation cache before execution. Emulators also link translation blocks that are frequently executed in succession, thereby forming *traces*. Traces allow execution to directly jump from one translation block to the next without having to switch from the translation cache to the emulator code in between, thereby speeding up emulation.

4.2 Emulation Using Memory Fences

Automatic insertion of fence instructions in parallel programs to eliminate memory consistency violations is a well known problem. Prior works propose compiler techniques that automatically insert fences, or tools that provide the programmer with information about possible memory consistency violation bugs in the program [8, 36, 20, 18]. These techniques rely on static or dynamic program analysis, memory model descriptions or program inputs. Unfortunately, such high level information is inaccessible to a system emulator at translation time. Moreover, these techniques have a high cost in terms of computation time and therefore are not suitable for integration in a system emulator where dynamic binary translation must be fast. During binary translation the emulator does not have access to information that can help decide whether an access to a memory address is to a private or a shared variable. It also does not have information about the semantics of the guest application that is being translated. Therefore, the emulator must be conservative and insert a memory fence after *every* guest application memory operation in order to ensure correctness [58]. Depending on the number of memory operations in an application, this can lead to a considerable slowdown.

Fences must be selectively inserted only to bridge the gap between the guest and the host memory consistency models. Therefore, certain optimizations can be used to reduce the number of memory fences inserted depending on the guest and the host system. For example, if the guest system is an x86 machine emulated on a POWER host system, then the emulator needs to enforce only R→R, R→W and W→W order on the host system (Table 1.1). Therefore, the emulator must insert a fence after every read operation. A fence is required only *between* two write operations. While inserting fences only after a specific type (read/write) of memory access can be easily implemented in an emulator, inserting fences only *between* specific types of memory accesses is harder. For example, it might not be possible to insert a fence between the last write in a translation block and the first write in the successive block. This is because there might be multiple translation blocks that could potentially be executed after a given translation block. Therefore, the last write in a translation block can be followed by a read or a write in a successive block. Moreover, translation blocks that are executed successively might be translated at different times depending on when they are inserted into the translation cache and hence, it might not be possible to infer the first memory operation in a successive translation block at translation time. Therefore, in order to guarantee correctness the emulator must conservatively insert a fence at the end of a translation block if the last memory access is a write, thus negating most of the performance gain due to the optimization. In practice, we find that using simple optimizations such as inserting a fence only after a specific type of memory operation, is just as effective.

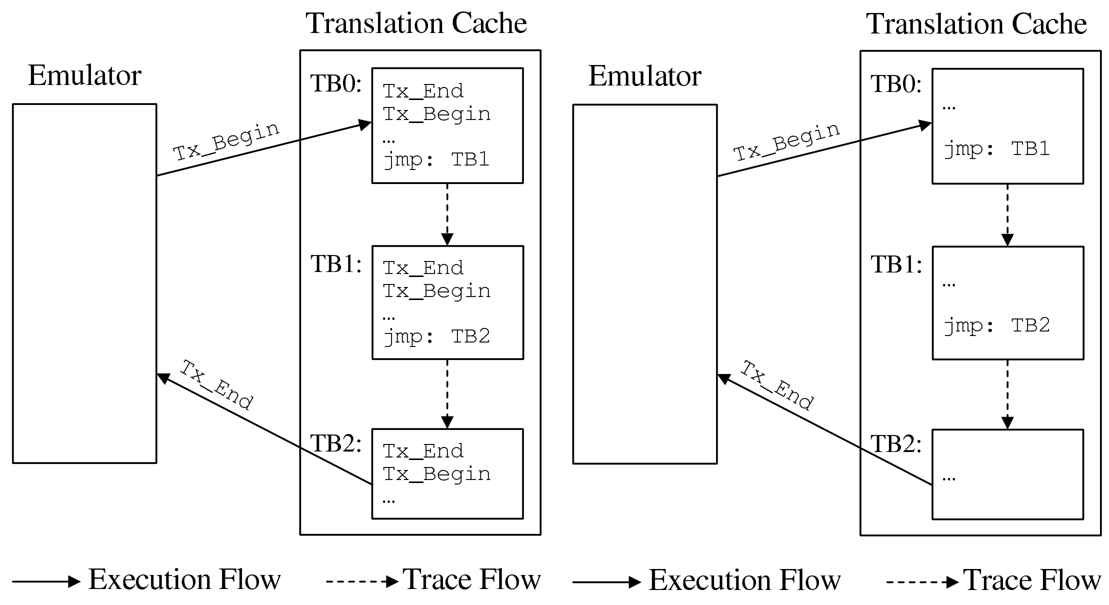
4.3 Emulation Using Transactions

An emulator can also use transactions for memory consistency model emulation. The guest code can be partitioned into chunks and executed as transactions on the host system. The hardware will detect any conflicts among the transactions that are executed simultaneously and re-execute them. Since the emulator cannot be certain if a memory access is to a private or a shared variable, it must enclose *every* memory access in the guest application within a transaction. Therefore, emulation using transactions is equally as conservative as emulation using fence instructions.

The simplest way to form transactions is at the translation block level. However, translation blocks are typically very small and contain only a few instructions. Therefore, executing each translation block as a separate transaction can incur a significant overhead. Executing entire traces as transactions can greatly reduce this overhead since traces typically contain tens of instructions. However, the transaction length is limited by the trace length, which can vary depending on the application.

Figure 4.1(a) illustrates how the guest code can be partitioned into transactions at the translation block boundaries. The emulator inserts *Tx_begin* and *Tx_end* instructions around each translation block at translation time. If the emulator uses HLE to implement the transactions then it must insert lock-elided lock and unlock instructions instead. A simple approach is to begin every translation block with a {*Tx_end, Tx_begin*} prologue that ends the previous block's transaction and begins the next one. *Tx_begin* and *Tx_end* instructions must be inserted when execution jumps to, and from, the translation cache in order to form complete transactions. Note that transactional execution ensures that there is an implicit fence between the translation blocks.

Forming transactions at the trace level involves a very small change. The emulator inserts *Tx_begin* and *Tx_end* instructions only when execution jumps to, and from, the translation cache but not around every translation block, as shown in Figure 4.1(b). Transactions must be started at every entry point, and terminated at every exit point, to the



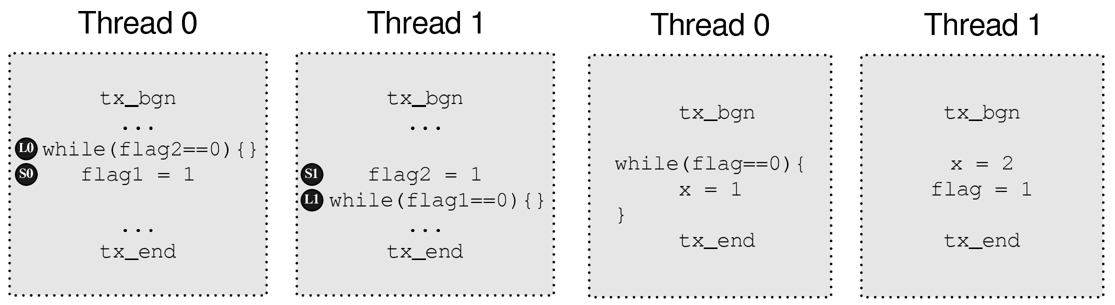
(a) Forming transactions at the translation block level

(b) Forming transactions at the trace level

Figure 4.1: Forming transactions at the translation block and trace level in an emulator.

translation cache. Although it might be beneficial to form transactions that are larger than the trace size it is not be possible to do this in an emulator environment. All the instructions executed within a trace correspond to the translated guest application. However, when the execution jumps out of the translation cache at the end of a trace, the executed instructions correspond to the emulator code itself. Since only the translated guest code must be executed inside a transaction, a transaction *must* be started and terminated at the beginning and end of a trace, respectively. Thus, any optimization that increases the trace length of an application will also increase the size of the transactions formed.

The emulator must generate code differently depending on the hardware support used to implement the transactions. If the transactions are implemented using HLE, then the emulator must start and end each transaction with lock-elided lock and unlock instructions. HLE, which is currently available only on Intel Haswell processors, automatically ensures



(a) Guest application with conditional synchronization emulated using transactions at the trace level. `flag1` and `flag2` are set to 0 initially.

(b) Guest application with conditional synchronization emulated using transactions at the translation block level. `flag` and `x` are set to 0 initially.

Figure 4.2: Forward progress issues in transactional emulation.

forward progress on an abort by re-executing the transactions as regular critical sections guarded by atomic locks [31]. Note that the emulator must use the same global lock to guard all the critical sections generated in the code. If HTM is used, then each transaction must start and end with the hardware specific *Tx_begin* and *Tx_end* instructions. Some HTM implementations, such as IBM z/Architecture, provide support to automatically ensure forward progress of aborted transactions [32]. However, other implementations, such as Intel Haswell and IBM POWER, require the programmer to ensure forward progress by explicitly specifying *fallback* code which is executed on a transaction abort [31, 9]. In such cases the emulator must generate the fallback code at run-time. The fallback code can point to the original transaction, however this might lead to the program not making any forward progress. Therefore, the emulator must be able to identify when there is no forward progress being made by the program (based on a timeout period or by monitoring the transaction abort rate), and re-translate the code using fences.

Forward progress issues can also arise if the guest application has conditional synchronization. Figure 4.2(a) shows a guest application with conditional synchronization that has been translated using transactions at the trace level. The variables `flag1` and `flag2` are set to 0 initially. The transactions span multiple basic blocks as shown in the figure. Note that the original guest code might contain fence instructions for the example shown in Figure 4.2(a), however, the emulator eliminates all fence instructions during translation since the code is emulated using transactions. For correct execution of the program, statement `S1` must complete before loop `L0`, and statement `S0` before loop `L1`. The introduction of transactions, however, requires that `L0` and `S0` execute atomically before `S1` and `L1`, or vice versa. Since the transactions shown in Figure 4.2(a) are not serializable, this either leads to a live lock or a dead lock. Live locks are possible even when the translated code has transactions at the translation block boundaries. Figure 4.2(b) shows an example guest application that has been translated with transactions formed at the translation block boundaries. Both transactions in Figure 4.2(b) span a single basic block as shown. In this example, it is possible that the store to `x` by `Thread 0` continuously aborts the transaction in `Thread 1`, thereby leading to a live lock. Such forward progress issues are not unique to transactional emulation, and are possible with any application that contains ill-formed transactions as demonstrated by previous studies [5]. The emulator must handle such cases by re-translating the code using fences as described previously. Prior papers which employ transactional execution in a binary translation environment propose a similar solution for detecting when a program is not making any forward progress [11].

The guest application may contain user-defined transactions and critical sections. Transaction support implemented in recent processors automatically handles nested transactions by subsuming the inner transaction. One of the advantages of using transactions is that the same approach can work on any host system, as long as it supports transactional execution, since it does not rely on fences. This makes it attractive for emulation where the guest-host configurations can vary.

4.4 Hybrid Emulation Using Memory Fences and Transactions

As characterized in Chapter 3.4, the overhead of using memory fences and transactions depends on the number of fences inserted at runtime, the conflict rate among threads in the application being emulated, and the size of the transactions formed at runtime. Therefore, a hybrid technique that uses both fences and transactions, and automatically chooses the best approach based on these factors, is likely to provide the best performance.

Such a hybrid technique must estimate the overhead of emulation using transactions and memory fences at runtime. We propose using hardware performance counters to measure the execution time of the translated host code in order to compare the overhead of the two techniques. By measuring the number of host cycles elapsed, the execution time of both versions of the translated host code can be measured accurately. The emulator profiles the overhead of using fences and transactions periodically, and then applies the best policy for emulating the application. Both the policies are profiled for a fixed number of trace executions. During the profiling phase of the fence policy the emulator measures the execution time of the host code translated using memory fences. Once the overhead of fence emulation has been measured, the overhead of transactional emulation is measured similarly. The emulator then makes its decision and applies the best policy for emulation until the next profiling phase.

The main overhead of dynamic profiling is due to translation cache invalidations. Before beginning a profiling phase, the emulator has to invalidate previously translated code and begin translation using the technique being profiled. The translation cache must be invalidated again when the policy being profiled changes. Similarly, once both fence and transaction profiling phases have been completed, the translation cache must be invalidated in order to translate the guest code using the best technique (this can be avoided if the best policy is the same as the policy that is profiled last). However, the overhead of translation cache invalidations is small since each translation block must be translated just once before it is inserted into the translation cache again. The overhead of measuring execution time using hardware performance counters is also negligible. Therefore, the overhead of the dynamic profiling technique is low. The proposed hybrid scheme is simplistic and switches between fence and transactional emulation at a coarse-grained level. A fine-grained hybrid technique that switches between fence and transactional emulation at a per-trace or per-translation block level might yield better performance. However, comparing the execution times of fence and transactional emulation at a fine-grained granularity also requires fine-grained book-keeping operations. The lack of light-weight hardware performance counters makes the overhead of fine-grained book-keeping operations prohibitively large. The design of an alternate light-weight fine-grained hybrid technique is challenging. A comprehensive treatment of this subject is beyond the scope of this thesis.

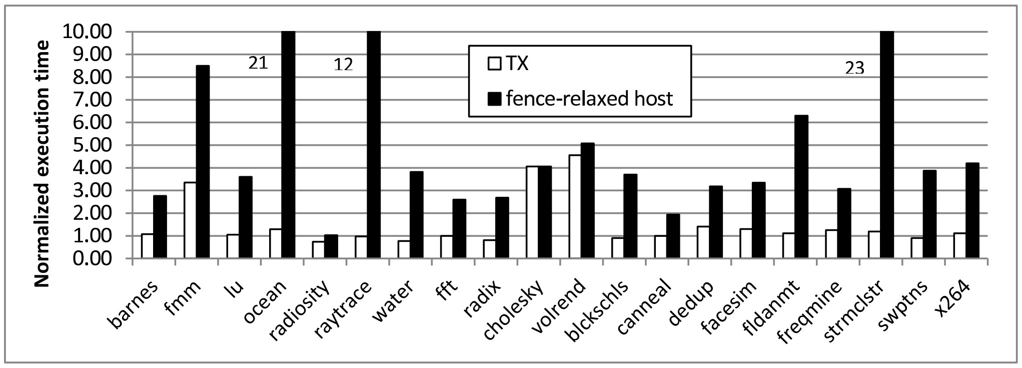
4.5 Evaluation

We use COREMU [66], a recently proposed parallel emulator for our study. Since COREMU supports only x86 hosts, we use a 4-core, 4-thread, Haswell architecture based, x86 Xeon E3-1225 v3 processor with transaction support as our host system. The processor speed is 3.2GHz and does not support simultaneous multithreading (SMT). Consequently, each core runs a single thread. Each core has a 32KB private L1 data cache and a 256KB L2 unified cache. All the cores share an 8MB L3 cache. The line size of all the caches is 64 bytes.

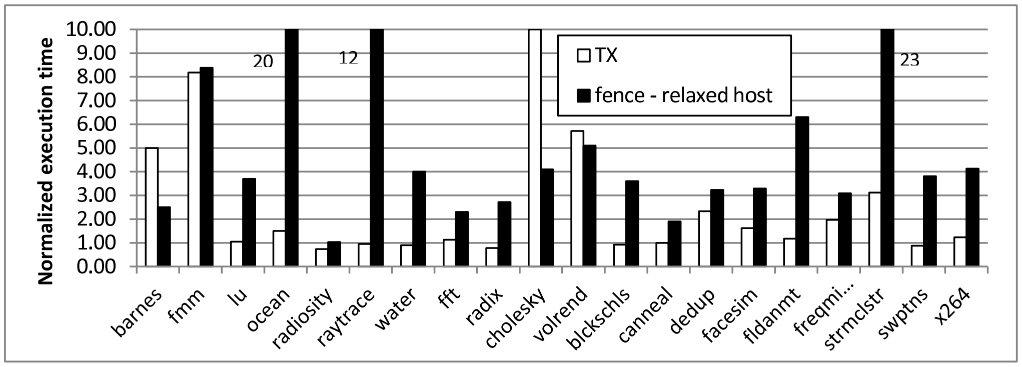
No modern processor implements a memory model stronger than the x86 memory model. Therefore, in order to simulate a guest system with a stronger memory model we assume a hypothetical sequential consistency guest system with the x86 ISA. We form guest applications for the sequential consistency guest system by taking existing x86 programs and removing all the fence instructions from them. We verify that the sequential consistency guest applications produce incorrect results when emulated on the x86 host system using the unmodified COREMU emulator. We discuss our results in the context of a real cross-ISA system in Section 4.5.4.

We use two sets of multithreaded guest applications to check for correctness and performance, respectively. The kernels described in Chapter 3.4 are used to verify correctness. We use eleven (entire set) applications from SPLASH-2 [67] and nine applications from PARSEC [4] in order to evaluate the performance overhead of the two techniques. We use the updated input sets from SPLASH-2x and PARSEC-3.0 for our evaluation. Since RAYTRACE is common to both SPLASH-2 and PARSEC we include it just once. We omit BODYTRACK, FERRET and VIPS from PARSEC due to difficulties encountered when running them on COREMU.

We modified COREMU to enforce sequential consistency (the guest memory model) on the host by automatically inserting memory fences after every store instruction in the guest application. Although a memory fence is required only between a store and a load in order to guarantee sequential consistency on an x86 system (to enforce the W→R constraint), such an optimization does not benefit much (Section 4.2). In practice, we find that inserting a fence after every store is a simple and effective solution. No fences are inserted after loads since it is not required on an x86 host system. In order to get a rough estimate of the overhead of fence emulation on a host system with a relaxed memory consistency model (such as POWER), we assumed that the x86 host has a relaxed memory model, and modified COREMU to insert a memory fence after every load and store instruction in the guest application. We also modified COREMU to execute the guest code as transactions using HLE support available on the host system. This simplifies our implementation since we do not have to generate fallback code or handle forward progress issues that might arise from using HTM support instead (Section 4.3). We implement transactional support at both the translation block and the trace level in order to evaluate them. We handle guest applications with conditional synchronization that can lead to a livelock or deadlock when emulated using transactions by monitoring the transaction abort rate using hardware



(a) 2 threads



(b) 4 threads

Figure 4.3: Execution time of the applications on the virtual machine using transactions (trace level), and memory fences inserted assuming a relaxed host system, normalized to execution time with fences inserted only after a store instruction.

performance counters and re-translating the guest application using memory fences if the abort rate if very high. We do not encounter such behavior with the evaluated real-world applications, and the kernels used to verify correctness, since they do not have such conditional synchronization constructs.

4.5.1 Performance Comparison

Figure 4.3(a) compares the performance of emulation using memory fences and transactions. The figure shows the execution times of the applications on the virtual machine, emulated with transactions formed at the trace boundaries, normalized to the execution times when emulated by inserting memory fences only after a store instruction. The figure also shows the execution times of the applications emulated using memory fences assuming that the x86 host has a weak memory consistency model (by inserting a memory fence after every store *and* load instruction), normalized to the same baseline. Therefore, in the *fence - relaxed host* configuration, a memory fence is inserted after every memory operation in the guest application, while in the baseline system a fence is inserted only after every store instruction in the guest application. Figure 4.3(b) shows the same data for 4-thread applications.

The transactional execution results demonstrate that there is a variation in the behavior of different applications. Transactional emulation is faster than the baseline for 2-thread applications such as RADIOSITY (0.74), RAYTRACE (0.97), WATER (0.77), RADIX (0.81), BLACKSCHOLES (0.9) and SWAPTIONS (0.9). However, the baseline fence emulation is faster for BARNES (1.08), FMM (3.35), OCEAN (1.29), LU (1.05), CHOLESKY (4.06), VOLREND(4.56), DEDUP (1.41), FACESIM(1.30), FLUIDANIMATE (1.11), FREQMINE (1.25), STREAMCLUSTER (1.19) and X264 (1.11). The trends are similar for most applications when run with 4 threads. Transactional emulation and emulation using the baseline fence configuration are comparable for FFT (1.00) with 2 threads; however the baseline is faster in the case of the 4-thread version. In the case of CANNEAL (1.00), the baseline and transactional emulation configurations are comparable for both 2 and 4 threads. These results show that the best technique depends on the characteristics of the emulated application.

The *fence - relaxed* emulation results show that, as expected, the execution times of most applications are much slower when a fence is inserted after every memory operation in the application. Moreover, unlike transactional emulation, the *fence - relaxed* execution times do not vary between the 2 and 4 thread applications; this is also expected since fence emulation overhead depends mainly on the number of memory operations per thread, rather than the number of threads in the application. The results show that transactional emulation is faster than *fence - relaxed* emulation for most of the applications in both the 2 and the 4 thread cases. Therefore, the *fence - relaxed* results suggest that transactional emulation can be more beneficial than fence emulation across a wide range of applications on a host system with a relaxed memory consistency model. We use the *fence - relaxed* results solely to illustrate the potential benefits of transactional emulation on a relaxed host system. Since inserting a fence after every memory operation is not required on an x86 host system, and doing so can artificially inflate the overhead of fence emulation, we do not include these results in the rest of the thesis. For the rest of this thesis, we refer to the baseline fence emulation configuration as simply fence emulation.

Table 4.1 lists the characteristics of the transactions formed in each application. Note that the transactions are formed at the trace boundaries. The table shows the average number of guest instructions, guest memory accesses and guest stores per transaction in the evaluated applications. It also shows the abort rate of the transactions for each application when run with 2 threads.

Transactional emulation results in poor performance in BARNES, FMM, OCEAN, CHOLESKY, VOLREND, DEDUP, FACESIM, FLUIDANIMATE and FREQMINE due to the high abort rate. Transactions abort in these applications due to data conflicts. Apart from true data dependency conflicts, false sharing in these applications also results in aborts since Haswell tracks dependencies at the cache block level. Transactional emulation in FFT and LU has a high abort rate, but its performance is comparable to emulation using fences since the overhead of fence emulation is also large due to the high number of stores per transaction in these applications. In the case of STREAMCLUSTER transactional emulation is slower than fence emulation, even with a fairly low abort rate, since the fence overhead is very low given the small number of stores per transaction.

Transactional emulation is faster than using fences in BLACKSCHOLES and SWAPTIONS since the abort rate in these applications is fairly low. Transactional emulation outperforms emulation using fences in RADIOSITY, WATER and RADIX because of two reasons. These

Application	Inst. per TX	(LD + ST) per TX	ST per TX	Abort rate (%)
BARNES	43.36	24.36	8.84	46.67
FMM	202.67	36.38	4.33	89.52
LU	8778.67	3445.00	984.33	99.52
OCEAN	185.81	66.04	0.65	90.41
RADIOSITY	38.02	4.63	4.51	2.77
RAYTRACE	17.45	6.82	0.55	1.67
WATER	69.28	27.66	7.24	11.38
FFT	387.38	128.50	45.88	92.66
RADIX	82.68	13.71	4.94	22.73
CHOLESKY	313.00	119.00	29.33	96.55
VOLREND	69.92	23.13	4.50	85.41
BLACKSCHOLES	22.69	6.89	1.96	3.06
CANNEAL	16.69	6.61	3.27	2.00
DEDUP	45.91	20.35	6.36	53.97
FACESIM	49.65	21.90	6.55	55.50
FLUIDANIMATE	32.92	11.15	1.76	25.99
FREQMINE	46.59	21.04	6.85	37.88
STREAMCLUSTER	18.40	7.73	0.25	10.57
SWAPTIONS	26.87	10.30	2.66	24.71
x264	28.32	10.13	2.46	18.99

Table 4.1: Characteristics of the transactions formed during emulation using transactions. LD stands for number of load instructions, ST stands for number of store instructions, and TX stands for transaction.

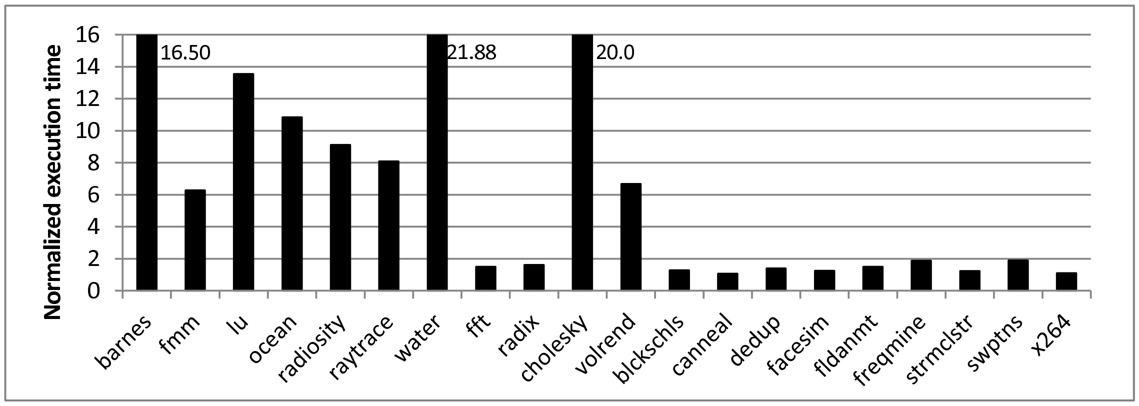


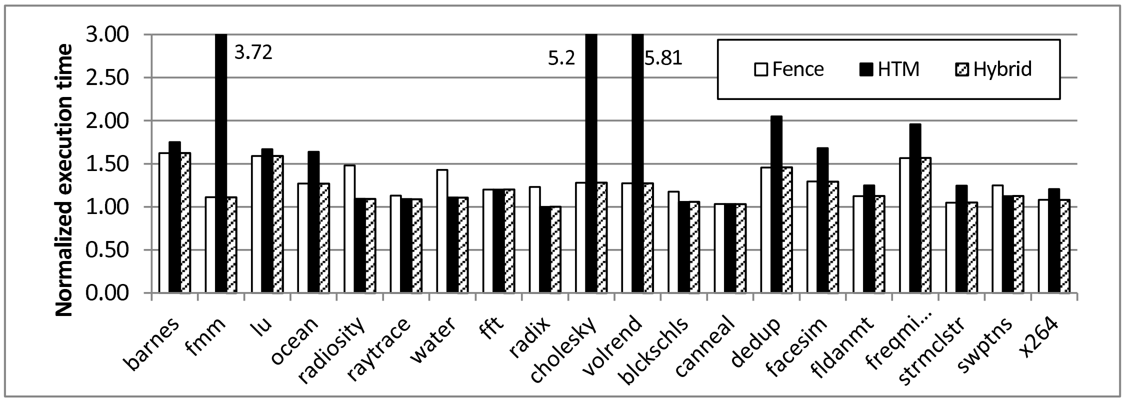
Figure 4.4: Execution time of the applications on the virtual machine with transactions formed at the translation block boundaries normalized to the execution time with transactions formed at trace boundaries.

programs have very low abort rates leading to a low overhead and, the number of stores per transaction in these applications is also fairly large resulting in a high overhead when using memory fences. Transactional emulation is only marginally faster in RAYTRACE, although it has a low abort rate, since the number of stores per transaction in the program is small thereby resulting in a low overhead when emulating using fences. In the case of CANNEAL, the two approaches are comparable since the execution time of the emulated application is dominated by the initial phase where the main thread reads the input data.

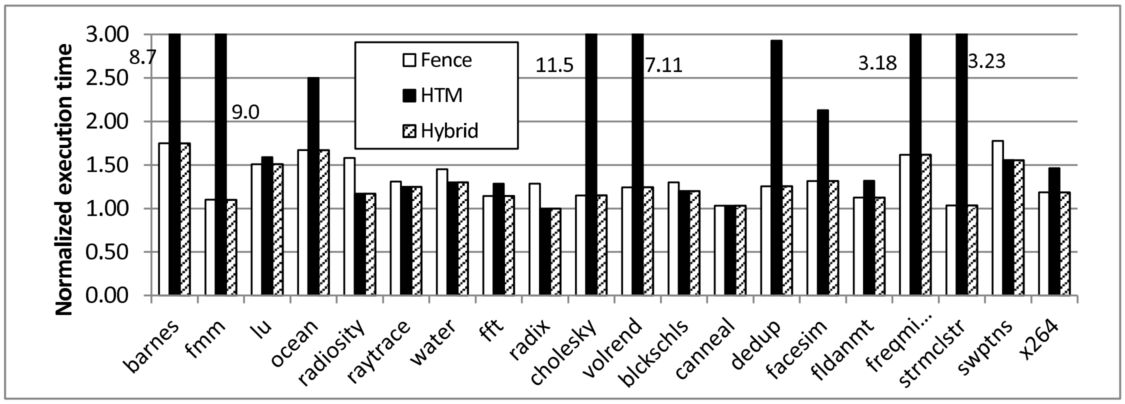
Figure 4.4 shows the effect of the transaction size on emulation. It shows the execution time of the applications on the virtual machine when emulated with transactions formed at the translation block boundaries normalized to execution time with transactions formed at the trace boundaries. The results are shown for 2-thread applications. The results show that emulation with transactions formed at translation block boundaries is significantly slower with as much as 20x slowdown (WATER). This is because in most of the applications translation blocks are just a few instructions in length, and transactions at the translation block boundaries are not large enough to amortize the overhead of starting and ending a transaction. There is a marked difference between the SPLASH-2 and the PARSEC applications. In the PARSEC applications, the difference in the number of instructions per trace and per translation block is not as large as in the SPLASH-2 applications. However, transactional emulation at the translation block level is still slower than emulation at the trace level in the PARSEC applications with as much as 1.89x slowdown (FREQMINE and SWAPTIONS). Since emulation with transactions formed at the translation block boundaries results in poor performance, in the rest of this thesis we focus only on transactional execution with transactions formed at the trace level.

4.5.2 Hybrid Emulation Using Fences and Transactions

Figure 4.5 shows the execution time of the applications on the virtual machine using memory fences, transactions, and our hybrid technique, all normalized to the execution time of the applications without any support. Although emulating an application without any support can lead to an incorrect emulation, we choose it as the baseline to illustrate the overhead of each emulation technique. The data is shown for both 2-thread and 4-thread applications. The results show that the hybrid technique chooses the best approach for all the applications. Most of the evaluated applications exhibit bipolar behavior with one technique resulting in much better performance than the other. Therefore, the proposed simple profiling technique is sufficient in order to choose the best policy. The profiling overhead for the hybrid technique is less than 1% and does not result in a slowdown. The average overhead for emulation using fences, transactions, and the hybrid technique, compared to the incorrect baseline emulation, is 27.1%, 60.8%, and 20.8% for 2-thread applications. The corresponding numbers for 4-thread applications are 32.3%, 128.4%, and 26.3%, respectively. Memory consistency model emulation using the proposed hybrid technique is 4.9% faster than emulation using fences and 24.9% faster than emulation using transactions, on average, for 2-thread applications. The corresponding numbers for 4-thread applications are 4.5% and 44.7%, respectively.



(a) 2 threads

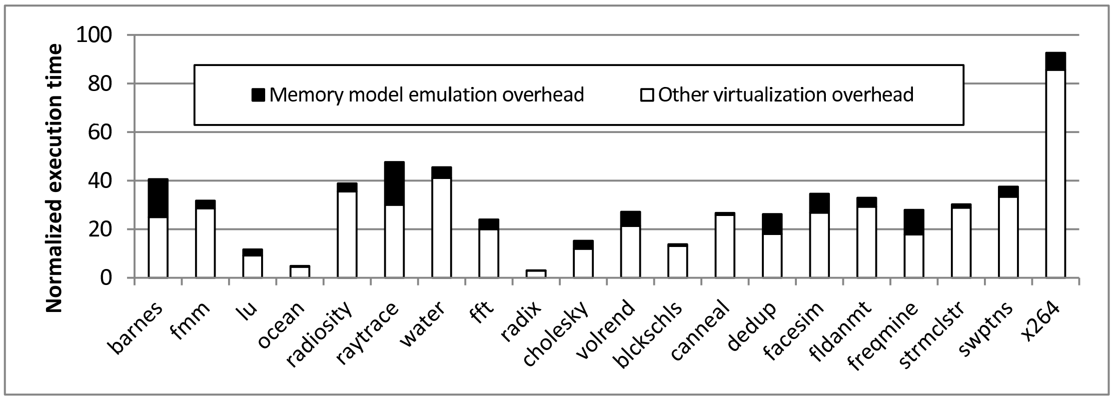


(b) 4 threads

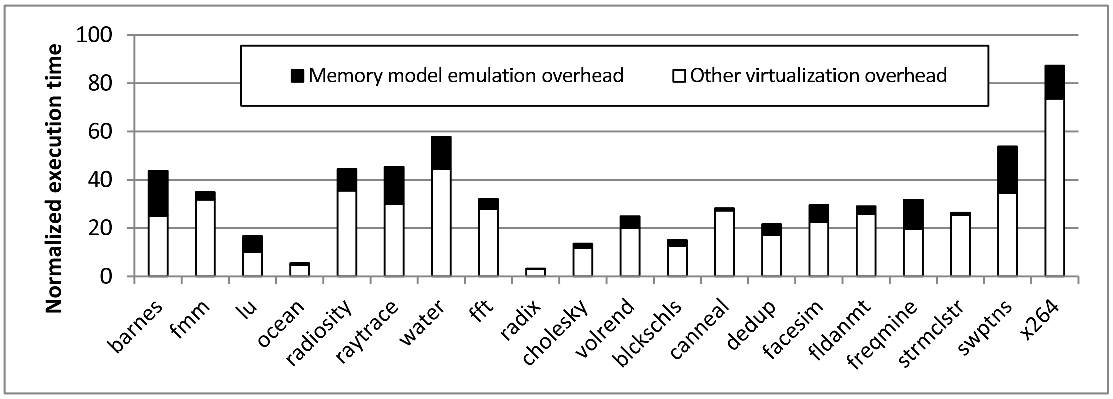
Figure 4.5: Execution time of the applications on the virtual machine using transactions (trace level), memory fences, and hybrid techniques normalized to execution time without any support (incorrect emulation).

4.5.3 Overhead of Memory Consistency Model Emulation

Figure 4.6 shows the execution time of the applications on the virtual machine, emulated using the hybrid technique, normalized to the native execution time. The normalized time is split to show the contribution of the overhead of memory consistency model emulation to the total overhead of system virtualization. The data is shown for both 2- thread and 4-thread applications. On average, the total virtualization overhead using the hybrid



(a) 2 threads



(b) 4 threads

Figure 4.6: Execution time of the applications on the virtual machine using the hybrid technique normalized to the native execution time. The normalized time is split to show the contribution of the overhead of memory consistency model emulation to the total virtualization overhead.

technique is 24.45x for 2-thread applications and 25.78x for 4-thread applications. The results show that in most applications the overhead of memory consistency model emulation is a small, but non-trivial fraction of the total overhead of system virtualization. On average, memory consistency model emulation contributes 11.3% and 13.9% of the total system virtualization overhead for 2-thread and 4-thread applications, respectively. The overhead of memory consistency model emulation can be decreased by selectively applying the emulation technique to only shared variable accesses in the application. However, in order to filter the accesses to private data, the emulator needs access to high level program semantic information. Incorporating program semantic information in emulators, using compiler or binary analysis, is left as future work.

4.5.4 Discussion

Our evaluation illustrates the validity of memory consistency model emulation using fences and transactions, and highlights the performance tradeoffs between the two approaches. It also shows that the hybrid technique proposed in this work can correctly choose the approach with the lowest overhead. Thus, although our evaluation uses a guest-host pair that differ only in their memory consistency models, our proposed technique and the performance tradeoffs between fence and transaction emulation are valid on a real cross-ISA system where both the instruction set and the memory consistency models of the guest-host pair differ.

The overhead of transactions formed at the trace level depends mainly on the transaction abort rate since they can effectively hide the overhead of starting and ending a transaction. Since the transaction abort rate is an application characteristic, we expect the overhead of emulating an application using transactions to be similar to the results shown in this thesis in a real cross-ISA system. The overhead of fence emulation, on the other hand, depends on the number of memory operations, which is an application characteristic, as well as the placement of the fences in the translated code, which depends on the guest and host memory consistency models. Hence, the overhead of emulating an application using fences might vary from the results shown in this thesis depending on the host and guest ISA pair. Although the technique with the lowest overhead for an application might change in a different guest-host ISA pair, the proposed hybrid technique would still be able to correctly identify it.

The total overhead of system virtualization is likely to increase in a real cross-ISA system due to the increased instruction translation time. The overhead of memory consistency model emulation in a real cross-ISA system can increase in cases where the hybrid technique employs fence emulation, but it would be similar for applications where transactional emulation is chosen by the hybrid technique. Thus, we expect the contribution of the overhead of memory consistency model emulation to the total overhead of system virtualization to be similar to the results shown in Figure 4.6 in a real cross-ISA system.

4.6 Related Work

Previous works have explored system virtualization of multithreaded applications. Sequential system emulators, which emulate multithreaded applications by time-sharing emulated threads on a single physical core on the host system, have been proposed previously [3, 41, 6]. In such emulators the memory consistency model of the guest system is inconsequential since only one thread is emulated at a time on the host system. Therefore, sequential emulators can emulate any guest-host memory consistency model pair. However, they suffer in performance since they do not utilize the resources available on current multicore systems. Parallel system emulators, which run multiple emulated threads simultaneously on multiple physical cores on the host system, greatly increase emulation speed [66, 16]. But such emulators only support same-ISA guest-host pairs or support only guest systems that have weaker memory consistency models than the host systems. The techniques proposed in this thesis are orthogonal to these works. They can be applied to existing parallel system emulators to extend them to support a wider range of guest-host pairs.

Techniques for automatic placement of fences in parallel applications running on relaxed memory systems have been explored in previous work. The delay set analysis algorithm is used widely for inferring the placement of memory fences in parallel applications on relaxed memory systems [57]. Various compiler techniques and automated tools for inserting fences based on the delay set algorithm have been proposed [8, 36, 20, 18]. However, such techniques are aimed at helping developers write concurrent programs for relaxed memory consistency models, and rely on static or dynamic program analysis, memory consistency model descriptions or program inputs. The limited availability of program semantic information at runtime, and the high cost of these techniques makes them unsuitable for use in emulators. The memory fence insertion techniques discussed in this thesis are simple, fast, and low cost techniques suitable for runtime systems.

The idea of executing memory accesses as coarse-grained, sequentially consistent chunks has been proposed as a solution for enforcing sequential consistency on modern processors without sacrificing performance [10, 22, 25, 28]. These prior works focus on the problem of enforcing sequential consistency on modern processors while our work focuses on memory consistency model emulation. We do not propose any hardware changes and instead leverage existing hardware on processors.

Using transactional memory has been previously proposed as a solution for thread-safe dynamic binary translation of multi-threaded applications [11]. The authors propose using transactional memory to eliminate data races among metadata maintained by dynamic binary translation tools in multithreaded applications. In contrast, our work proposes using transactional execution as a solution for memory consistency model emulation.

4.7 Summary

In this work we focus on the problem of memory consistency model emulation in virtual machines where the memory consistency models of the guest and the host systems differ. We compare using memory fences and transactions in order to support memory consistency model emulation. We discuss the tradeoffs involved in using memory fences and transactions for correct emulation, characterize the overhead of using fences and transactions on a recent processor, and show that transactions are a viable alternative to using memory fences for correct emulation. We implement the two approaches on COREMU, a recently proposed parallel emulator, and highlight the implementation issues. We propose a hybrid technique that switches between the two approaches depending on the application characteristics in order to minimize the overhead. We evaluate the overhead of the two approaches and our hybrid technique on a set of real-world parallel applications. The results show that, on average, the proposed hybrid technique is 4.9% faster than emulation using fences and 24.9% faster than emulation using transactions for 2-thread applications. The corresponding numbers for 4-thread applications are 4.5% and 44.7%, respectively.

Chapter 5

Implementing Speculative Parallelization Using Transactional Execution

With multicore processors becoming ubiquitous, it is crucial for sequential applications to extract sufficient parallelism in order to benefit from the trend. Although there has been extensive research on automatic parallelization of sequential applications, they have been effective mainly on certain classes of scientific applications. For a large class of sequential applications with ambiguous memory dependences, automatic parallelization remains a significant challenge. Prior research has proposed Thread-Level Speculation (TLS) as solution to automatically parallelize sequential applications. Although TLS has been demonstrated to show performance improvement in simulated environments, hardware support for TLS is yet to be adopted by the processor industry. However, hardware support for transactional execution introduced in recent multicore processors, in the form of Hardware Transactional Memory (HTM), guarantees some of the features required to realize TLS.

Both HTM and TLS require efficient mechanisms for memory conflict detection and the rollback/replay when speculation fails. However, there are also significant differences between HTM and TLS that affect how sequential applications are speculatively parallelized. When a sequential program is parallelized using TLS, code segments from different parts of the program are speculatively executed in parallel. In order to preserve the sequential semantics and maintain correctness, these parallel code segments must be completed in the same order as in the sequential execution. Prior TLS works assume hardware support for such ordered commit, however existing HTM implementations do not allow transactions to commit in a pre-determined order. For efficient TLS performance it is crucial to reduce the number of speculation failures. Therefore, previous works have explored hardware optimizations for TLS that provide the means to synchronize and forward values between speculative threads [2, 26, 23, 43]. Unfortunately, existing HTM implementations do not provide hardware support for data synchronization or data forwarding among transactions. Existing HTM offerings also do not support other advanced hardware optimizations assumed by many prior TLS proposals, such as word level conflict detection. Given these key differences between the existing HTM support and the hardware support required for efficient implementation of TLS, it is not clear if TLS can be implemented on existing multicore processors. If successfully implemented on current processors, TLS can improve the performance of a large class of existing and emerging sequential applications.

In this work we aim to study the implementation of TLS execution using HTM support that is available on existing microprocessors. We begin this chapter by describing speculative parallelization of an example sequential application using TLS. We then describe how HTM support on the Haswell processor can be used to speculatively parallelize the same sequential application. We conclude by describing software mechanisms to improve the performance of our proposed TLS design. We present a detailed analysis and evaluation of our proposed TLS design in Chapter 6.

```
 1  #define SIZE 10000
 2  #define NUM_ITERATIONS 1000000
 3  struct bucket *hash_table[SIZE];
 4
 5  ...
 6  for (i = 0; i < NUM_ITERATIONS; i++) {
 7          hash_table[rand() % SIZE]->field += ...;
 8          ... = hash_table[rand() % SIZE]->field;
 9  }
10  ...
```

Listing 5.1: A microbenchmark which updates and reads random entries in a hash table. The microbenchmark is a good candidate for speculative parallelization.

5.1 Thread-Level Speculation

Consider the sample microbenchmark shown in Listing 5.1. Lines 6-9 show the main loop of the microbenchmark which updates and reads elements in a hash table structure `hash_table`. Since the loop updates and reads a random element in `hash_table` in each iteration, it is possible that multiple iterations of the loop operate upon the same element. Therefore, although most of the iterations of the loop can be executed in parallel, a traditional parallelizing compiler will not automatically parallelize the loop as it cannot prove which iterations are independent at compile time. Therefore, this loop is a good candidate for speculative parallelization.

Figure 5.1 shows a sample execution of the loop in Listing 5.1 using traditional TLS hardware as described by previous works. The loop is speculatively parallelized using two threads with the iterations being divided equally among them. `Thread 0` executes the even numbered iterations (0, 2, 4 ...), and `Thread 1` executes the odd numbered iterations (1, 3, 5 ...). In the execution shown in Figure 5.1, iterations 0 and 1 both operate on the same

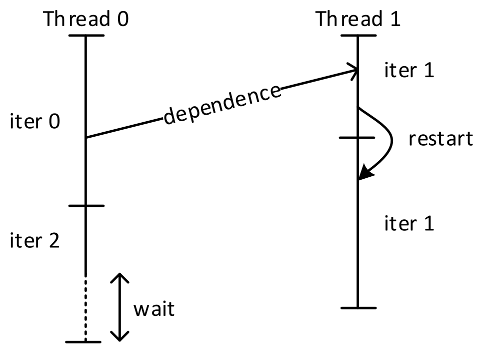


Figure 5.1: Sample execution of the microbenchmark in Listing 5.1 under TLS hardware.

element in `hash_table` and hence, there is a conflict between the two parallel threads. The hardware detects this conflict and aborts `Thread 1` since it is more speculative. `Thread 1` re-executes iteration 1 after the abort and succeeds the second time. Meanwhile, `Thread 0` executes iteration 2 in parallel, but has to wait until `Thread 1` commits iteration 1 in order to maintain the original sequential ordering. The hardware support for ordered transactions allows `Thread 0` to wait before committing.

TLS allows potentially dependent code segments from a sequential application to execute concurrently. If there are no runtime dependences, the speculative execution is successful thereby speeding up application execution. While TLS can still lead to an execution speedup in the presence of occasional speculation failures, frequent speculation failures can lead to poor performance. In fact, TLS performance can be worse than sequential performance if the application suffers from very frequent speculation failures. Therefore, not all sequential applications might be amenable to speculative parallelization.

5.2 Implementing Thread-Level Speculation Using Hardware Transactional Memory

We use the Intel RTM support, described in Chapter 2, for implementing transactions in this work. Listing 5.2 shows the microbenchmark in Listing 5.1 speculatively parallelized using HTM. `_xbegin`, `_xend` and `_xabort` are transactional memory intrinsics provided by the GCC compiler. `thread_function` shows the function executed by each thread.

The total number of loop iterations is equally divided between the threads. In our TLS implementation, each thread speculatively executes a block of iterations within each transaction. Executing a block of iterations within a transaction, rather than a single iteration, helps amortize the transaction overhead, as well as reduce false sharing between the threads. We elaborate on the benefits of executing a block of iterations within each transaction in more detail in Section 5.3.1. For the sake of clarity, the rest of this section assumes that each thread executes a single iteration within a transaction.

The TLS version of the loop is shown in Listing 5.2. The loop iterates over the iterations assigned to each thread (line 16). Each iteration is executed within a transaction. Since there is no hardware support for ordered transactions in Haswell, transactions are ordered through software synchronization. The threads use the shared variable `next_iter_to_commit` to track the next iteration that must be committed in sequential order. `next_iter_to_commit` is initially set to 0. Before beginning the execution of its current iteration, each thread checks to see if its iteration is the next that must be committed in sequential order (line 19). If it is not, then the thread starts speculative execution by beginning a new transaction using `_xbegin` (line 24). `_xbegin` returns the status _XBEGIN_STARTED when a transaction starts successfully. If the transaction later aborts, then the execution jumps back to `_xbegin` which then returns the appropriate error status. On an abort, the thread checks `next_iter_to_commit` again and re-executes the transaction (line 25). Each thread checks `next_iter_to_commit` before committing its transaction (line 29). If it cannot commit next, then it explicitly aborts the transaction using `_xabort` and re-executes it (line 30). Note that repeatedly checking the value of `next_iter_to_commit` within the transaction is not a good idea since the transaction will be automatically aborted when another thread updates `next_iter_to_commit`. After a thread successfully commits its current iteration, it updates `next_iter_to_commit` (line 33). The TLS version of the

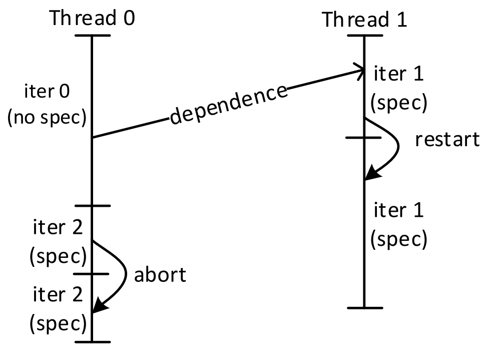


Figure 5.2: Sample execution of the microbenchmark in Listing 5.1 under TLS implemented using HTM.

loop requires memory fences at appropriate locations to ensure correct results on x86. We omit them for the sake of clarity.

If before starting a transaction a thread sees that its current iteration is the next that must be committed in sequential order, it executes the iteration non-speculatively (`spec_exec` is set to 0 in line 20). This guarantees that at least one thread eventually executes its current iteration non-speculatively, thereby ensuring forward progress.

Figure 5.2 shows a sample execution of the loop in Listing 5.2. `Thread 0` starts by executing iteration 0 non-speculatively since `next_iter_to_commit` is initially set to 0, while `Thread 1` begins by executing iteration 1 speculatively within a transaction. Since both iterations 0 and 1 update the same element in `hash_table`, a conflict is detected by the HTM, and `Thread 1`'s transaction is aborted and restarted. `Thread 1` successfully commits its transaction upon re-execution since `Thread 0` updates `next_iter_to_commit` to 1 after committing iteration 0. Although `Thread 0` successfully executes iteration 2, it explicitly aborts its transaction in order to maintain sequential ordering as `Thread 1` has not yet committed iteration 1. `Thread 0` eventually commits iteration 2 on its second execution. Using software synchronization and explicitly aborting transactions to enforce transaction ordering can lead to wasted CPU cycles. If there are frequent transaction aborts, either due to memory conflicts or due order inversion, then the performance of the speculatively parallelized version can in fact be worse than the sequential version.

5.3 Improving the Performance of Thread-Level Speculation

In this sub-section we propose software optimizations that can be applied on top of the TLS implementation discussed in Section 5. We begin by describing how the overhead of transactional execution can be effectively amortized. We then demonstrate how data synchronization can be implemented in order to improve the performance of the loops with frequent inter-loop dependences. We further propose a dynamic tuning mechanism in order to prevent the performance degradation of the applications which suffer from frequent speculation failures.

5.3.1 Amortizing Transaction Overhead

The size of a transaction can significantly affect the performance of transactional execution. Very small transactions cannot hide the overhead of starting and ending a transaction. Larger transactions can effectively amortize transaction overhead. Therefore, in our TLS implementation each thread speculatively executes a block of iterations within each transaction. Executing a block of iterations within a transaction also reduces false sharing between the threads, which in turn reduces the number of transaction aborts since conflicts are detected at the cache line granularity on Haswell. For example, if each iteration in a loop updates a 4-byte element of an array, then the *iteration block size* to eliminate false sharing on a 64 byte cache line must be 16 iterations (64/4). Note that transaction sizes cannot be increased indiscriminately. A very large transaction increases the possibility of a conflict between the threads. A large transaction can also fail if the number of unique cache lines accessed within it exceeds a hardware specific maximum read/write size. The iteration block size varies for each application depending upon its characteristics. We evaluate the effect of the iteration block size in detail in Chapter 6.

5.3.2 Improving TLS Performance Using Data Synchronization

In order to improve TLS performance it is crucial to reduce the number of transaction aborts due to memory conflicts. Speculative parallelization of a loop with frequent cross-iteration data dependences can result in poor performance due to the high number of transaction aborts because of memory conflicts. Therefore, previous works have explored hardware optimizations for TLS that provide the means to synchronize and forward values between speculative threads [2, 26, 23, 43]. Unfortunately, existing HTM implementations do not provide hardware support for data synchronization or data forwarding among transactions. However, data synchronization can be achieved using software techniques on current HTM implementations.

Consider the frequently executed loop in the SPEC2006 [60] benchmark HMMER, shown in Listing 5.3. The loop has two cross-iteration data dependences; line 4 where `dc[k]` is computed based on the value of `dc[k-1]`, and line 5 where `dc[k]` is computed based on the value of `mc[k-1]`. The rest of the loop is omitted for the sake of clarity. The data dependence of `dc[k]` on `dc[k-1]` and `mc[k-1]` leads to frequent speculation failure when the loop is speculatively parallelized. Synchronization of the frequently dependent value can help alleviate this problem.

Consider the loop in Listing 5.4, which is the same as in Listing 5.3, except that it has been speculatively parallelized using explicit synchronization. The calculation of `dc[k]` is serialized and ordered using a separate synchronization variable, and executed non-speculatively, in lines 3-5. The memory fences required for correct execution on x86 have been omitted. The rest of the original loop is speculatively executed using the approach described in the previous section. Since the frequent dependence is no longer executed within a transaction, the probability of a memory conflict is lower. Although the synchronization results in partial serialization of the loop, it can result in better performance by eliminating frequent speculation failures. However, it is important to limit the number of instructions serialized in order to achieve good performance. Although the loop in Listing 5.4 illustrates synchronization using a block size of 1 iteration, the approach can be easily extended to an arbitrary block size. We also note that the same technique can be used to synchronize scalar value communication between speculative threads.

Only certain loops are amenable to data synchronization. If there are too many frequently occurring data dependences, or if the number of instructions in the serial portion is large, then synchronization can lead to poor performance. Of the evaluated SPEC2006 benchmarks which contained frequent cross-iteration data dependences, only HMMER was amenable to synchronization. We find that using synchronization greatly reduces the number of memory conflicts in this application.

5.3.3 Dynamic Performance Tuning

Certain applications can suffer from frequent transaction aborts resulting either from memory conflicts or out of order transaction commits. If the transaction abort rate is very high, the TLS performance of such applications can in fact be worse than that of the sequential execution. However, this might not be the case for all the inputs to a given application. Therefore, a dynamic framework that can monitor the TLS performance and revert to sequential execution in case of frequent speculation failure is desirable. We propose a dynamic tuning framework that utilizes the hardware performance counters available on modern processors to automatically monitor TLS performance and disable speculative parallelization when necessary.

A possible heuristic to identify if TLS performance is worse than the sequential execution performance is the abort rate of the transactional execution. If the abort rate is very high then it is likely that the performance of the speculatively parallelized version of the application is worse than that of the sequential version. We find that sampling the transaction abort rate of a small fraction of the speculatively parallelized main loop of the application is sufficient in order to make the decision of enabling or disabling TLS in the application. Hardware performance counters available on modern processors provide a convenient means of deriving the transaction abort rate of a given piece of code. We maintain two versions of the main loop in each application - the original sequential version, and the modified speculatively parallelized version. When the loop is executed for the first time, we use the TLS version of the main loop and monitor the transaction abort rate of the first 5% of the total number of iteration blocks in the main loop. If the transaction abort rate is above a threshold, we execute the remaining iterations of the main loop sequentially. We use an aggressive threshold of 90% in our experiments (we discuss the reasons for choosing the aggressive threshold in Chapter 6). If the transaction abort rate is lower than the threshold, we continue to execute the TLS version of the main loop, however, we do not continue to monitor the transaction abort rate.

The overhead of sampling the transaction abort rate of the main loop using the hardware performance counters is negligible. Moreover, we sample the abort rate for only for a small fraction of the speculatively parallelized main loop. Instrumenting the code in order to monitor the transaction abort rate is straightforward. We measure the values of the appropriate performance counters once before starting the execution of the main loop, and once again after the required number of iteration blocks have been completed. The transaction abort rate is derived from the measured values. Note that this methodology precludes the tuning mechanism from adapting to time-varying phase behaviors of the main loop. In order to account for such behavior, the main loop would need to sampled periodically. However, we find that such a periodic sampling mechanism increases the overhead and offers relatively little performance benefit for the evaluated applications.

```
#define SIZE 10000
#define NUM_ITERATIONS 1000000
struct bucket *hash_table[SIZE];
volatile int next_iter_to_commit;


...
void *thread_function(void *targ)
{
  int start_index = *((int *) targ);
  int inc_to_next_iter = nthreads - 1;
  int cur_iter_index;
  int spec_exec;
  int status;

  for (i = start_index; i < NUM_ITERATIONS; i += inc_to_next_iter) {
    cur_iter_index = i;
try:
    if (cur_iter_index == next_iter_to_commit) {
      spec_exec = 0;
    }
    else {
      spec_exec = 1;
      if ((status = _xbegin()) != _XBEGIN_STARTED)
        goto try;
    }
        //lines 7-8 from original loop
    if (spec_exec) {
      if (cur_iter_index != next_iter_commit)
        _xabort(0xff);
      _xend();
    }
    next_iter_to_commit = cur_iter_index + 1;
  }
}
```

Listing 5.2: The microbenchmark in Listing 5.1 speculatively parallelized using HTM support. The listing illustrates how software synchronization can be used to enforce ordered transactions in HTM.

```
for (k = 1; k <= M; k++) {
 ...
 ...
 dc[k] = dc[k-1] + tpdd[k-1];
 if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
 ...
 ...
}
```

Listing 5.3: Frequently executed loop in HMMER.

```
for (k = 1; k <= M; k++) {
 //compute mc[k]
 //start serial & ordered
 //compute dc[k]
 //end serial & ordered

 //transaction start
 ...
 ...
 ...
 //transaction end
}
```

Listing 5.4: TLS version of frequently executed loop in HMMER using data synchronization. The frequent data dependence in the loop is ordered and executed non-speculatively using software synchronization.

Chapter 6

Evaluating Thread-Level Speculation

Chapter 5 outlined our proposed mechanism to implement TLS using Haswell's HTM support. Chapter 5 also discussed software optimizations to improve the performance of our proposed TLS design. In this chapter, we present detailed analysis and results which demonstrate the performance of our TLS implementation.

6.1 Methodology

We use a 4-core, 4-thread x86 Intel Xeon E3-1225 v3 Haswell processor with TSX support to conduct our experiments. The processor speed is 3.2GHz and does not support simultaneous multithreading (SMT). Consequently, each core runs a single thread. Each core has a 32KB private L1 data cache and a 256KB L2 unified cache. All the cores share an 8MB L3 cache.

We begin with a simple micro-benchmark to demonstrate the overhead associated with transactional execution. We then evaluate the performance of TLS using a selected set of SPEC2006 [60] benchmarks. We choose these benchmarks since previous studies [50] have demonstrated that they are able to benefit from speculative parallelization in a simulated environment. In particular, we focus on MCF, MILC, HMMER, H264, LBM, and SPHINX3, since they were demonstrated to achieve more than a 50% performance improvement compared to their sequential versions when speculatively parallelized. It is worth pointing out that, previous studies on TLS using simulators [50, 68, 69, 40] often build on a TLS execution model with hardware support for ordered transactions, data forwarding between the threads, as well as word level conflict detection. These features are not available on the HTM support in Haswell. The lack of such support forbids us from achieving the previously claimed performance gain.

In each application, we focus on the set of loops that were demonstrated effective with speculative parallelization by previous work [50, 51], and manually parallelize the loops. We use the POSIX thread library to parallelize the applications. Automatically selecting loops for speculative parallelization is beyond the scope of this work. Previous work has proposed techniques for selecting suitable loops for speculative parallelization in sequential applications [40]. We use the transactional memory intrinsics provided in GCC (version 4.8) to implement our transactions. Compiler support for TSX is invoked with the `-mrtm` flag. Unless mentioned otherwise, we always use the optimal iteration block size for all the experiments in this paper. We use the Intel Performance Counter Monitor (PCM) library to monitor the hardware performance counters for our dynamic tuning mechanism. The SPEC reference inputs are used in all the experiments. Table 6.1 illustrates the application name, the location of the most frequently executed loop in the source code of the application, as well as the execution coverage of these loops. In other words, all the performance improvements presented in this paper are obtained on real hardware using real benchmarks.

Application	Loop	Coverage
mcf	pbeampp.c: 165	63%
milc	quark_stuff.c:1523	35%
hmmer	fast_algorithms.c:133	79%
h264	mv-search.c:982	17%
lbm	lbm.c:186	59%
sphinx3	vector.c:513	35%

Table 6.1: The evaluated SPEC2006 applications along with their frequently executed loops and the execution coverage of the loops.

6.2 Micro-benchmark Results

We use a simple micro-benchmark to evaluate the effect of the iteration block size on TLS performance, the overhead of transactional execution, and the overhead of the transaction aborts due to order inversion. The micro-benchmark iterates through an array of 16,384 (16K) integers incrementing each integer 1,000 times. The micro-benchmark is highly parallel and does not have any cross-iteration data dependences. We choose an array of size 16,384 so that the entire array occupies an integral number of cache lines ((16K * 4) / 64 = 1K cache lines) completely. We parallelize the outer-loop of the micro-benchmark using TLS. Although executing the iterations out of order does not affect the correctness of the micro-benchmark, we ensure that the iterations executed by the threads are completed in the same order as in the sequential version of the benchmark. In the TLS version, we use iteration block sizes which are multiples of 16 iterations so that each transaction operates on an integral number of cache lines. Consequently, there are no memory conflicts due to false sharing among parallel threads in the TLS version. We experiment with the speculatively parallelized version of the micro-benchmark with 1, 2 and 4 threads. The single thread TLS version executes the same code as the parallel versions except that is uses just one thread. Consequently, there are no transaction aborts due to memory conflicts or

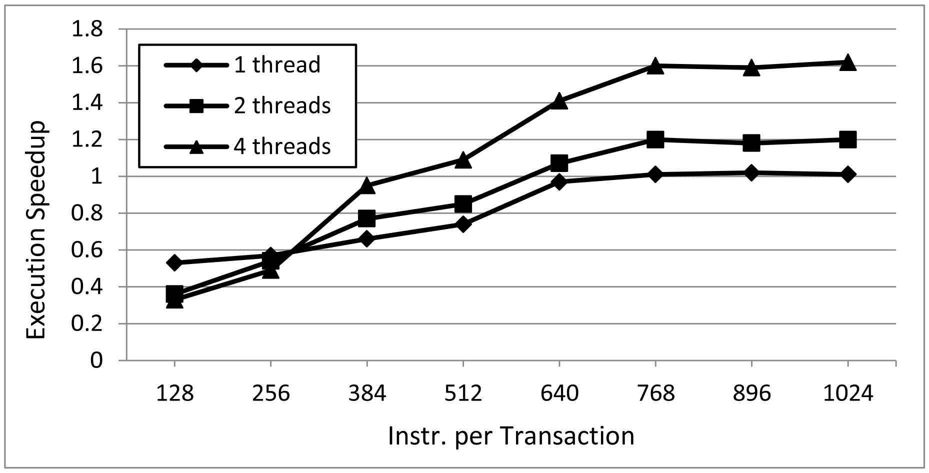


Figure 6.1: The execution time of the micro-benchmark speculatively parallelized using 1, 2 and 4 threads, normalized to its sequential execution time across various transaction sizes. The 1 thread version executes the same code as the 2 and 4 thread versions, except with just a single thread. Consequently, there are no aborts due to memory conflicts in the 1 thread version.

order inversion in the single thread version. We use the 1 thread version to highlight the overhead of transactional execution in the absence of aborts. Note that even the 2 and 4 thread TLS versions do not encounter any transaction aborts due to memory conflicts, however, the 2 and 4 thread TLS versions do suffer from transaction aborts due to order inversion.

Figure 6.1 shows the execution times of the speculatively parallelized versions of the micro-benchmark, normalized to the sequential execution time, for 1, 2 and 4 threads across various iteration block sizes shown as number of instructions per transaction. Each iteration of the main loop contains 8 instructions. The data highlights the impact of the transaction size on the performance of TLS using HTM. Focusing on the performance of the 1 thread version, we see that when the number of instructions per transaction is small it cannot effectively amortize the overhead of transactional execution even in the absence of the transaction aborts due to memory conflicts and order inversion. We see that the TLS performance gradually increases as the number of instructions per transaction increases, and once the optimal transaction size is reached the performance is close to sequential performance. At the optimal transaction size the overhead of transactional execution is negligible. Note that a larger transaction size increases the chances of a memory conflicts among the threads. However, since the micro-benchmark does not have any memory conflicts this effect is not manifested. Moreover, a very large transaction size can also increase the number of memory locations read and written within the transaction, which in turn can cause aborts due to read/write set buffer overflows. Since the micro-benchmark accesses only a few cache lines even at very large transaction sizes this effect is not visible. Once the optimal size is reached the performance of the 1 thread TLS version remains constant.

We see a similar trend in the 2 and the 4 thread versions. A small transaction size results in poor performance. In fact, the performance of the 2 and the 4 thread versions is worse than the 1 thread version at very small transaction sizes since the parallel versions suffer from transaction aborts due to transaction order inversion. However, at the optimal transaction size TLS improves the performance of the micro-benchmark by 1.2x with 2 threads and by 1.6x with 4 threads. The performance remains constant once the optimal transaction size has been reached. These results show that if the sequential application is amenable to speculative parallelization, then TLS implemented using the current HTM can result in a significant performance improvement.

Figure 6.2 shows the percentage of the aborted and successful transactions in the speculatively parallelized version of the micro-benchmark. The figure shows the percentage of successful transactions, and the percentage of transactions aborted due to memory conflicts, order inversion, read/write buffer overflows, and other unknown reasons. The data is shown for both the 2 and the 4 thread versions. When a transaction is aborted in Intel TSX, the reason for the abort is recorded in the EAX register using an opcode as described in Chapter 2. We use the opcode to track the reason for each transaction abort. Aborts due to memory conflicts and buffer overflows have pre-defined opcodes. In our experiments, an

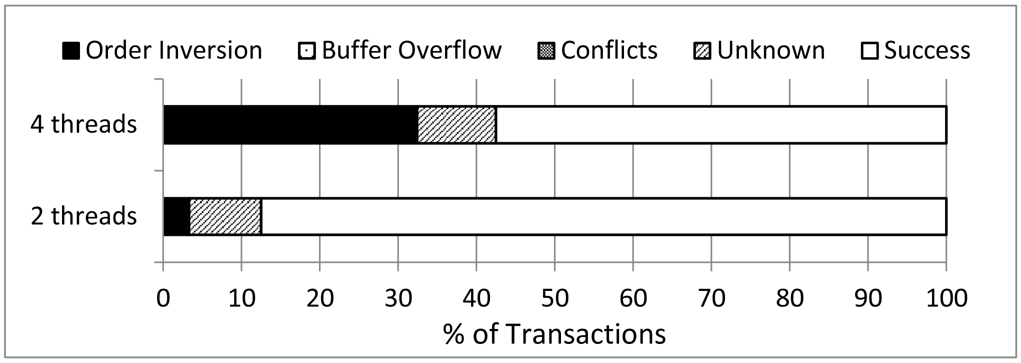


Figure 6.2: The percentage of transactions aborted in the speculatively parallelized version of the micro-benchmark for 2 and 4 threads. The transaction abort rate is split to show the fraction of the total abort rate caused due to different reasons.

explicit abort due to _xabort() is recorded using a special opcode which indicates that the transaction was aborted due to an order inversion (line 33 in Listing 5.2). The "Unknown" reason corresponds to the case when the processor does not record any opcode in the EAX register on a transaction abort. Although there are two other valid EAX opcodes for a transaction abort in Intel TSX ("Abort due to a debug trap" and "During nested transaction"), they did not occur in our experiments. From Figure 6.2, we can see that 10.5% of the transactions are aborted with 2 threads, and the abort rate increases to 42.5% with 4 threads. As expected, we do not see any transaction aborts due to memory conflicts in the micro-benchmark. Most of the aborts are because of order inversions which occur due to small timing differences among the threads. The 4 thread version shows an increase in the number of the transaction aborts due to order inversion since there is a higher chance of the transactions committing out of order with 4 threads running in parallel. These results show that the lack of ordered transaction support in the current HTM implementations results in a considerable amount of wasted cycles when a sequential application is parallelized using TLS.

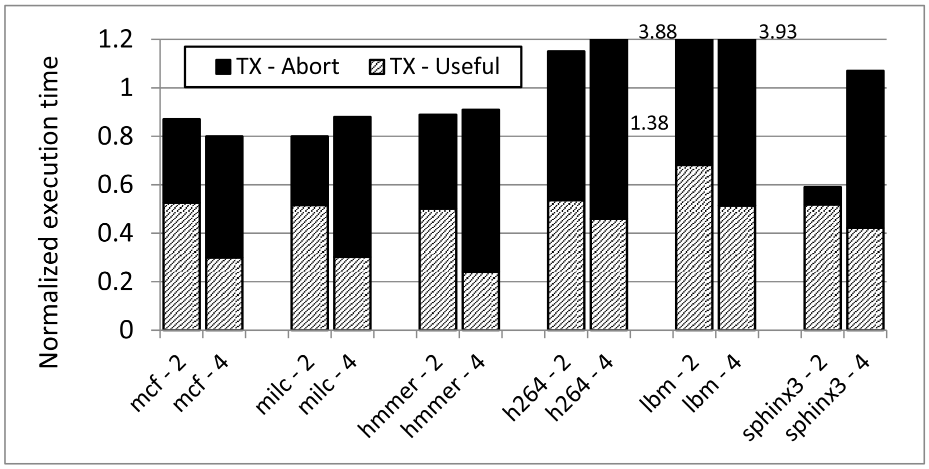


Figure 6.3: The parallel execution times of the main loop in the evaluated SPEC2006 applications normalized to their corresponding sequential execution times for 2 and 4 thread versions.

6.3 SPEC2006 Results

In this subsection we present a detailed analysis of the performance gain achieved by the selected SPEC2006 benchmarks using speculative parallelization. We begin by evaluating the performance improvement of the speculatively parallelized main loops in the evaluated applications. We then examine the causes for speculation failures in these applications in detail. Next, we analyze the effect of the iteration block size on TLS performance and demonstrate how choosing the appropriate iteration block size can effectively amortize the overhead of transactional execution. Finally, we study the impact of TLS on the overall performance of the SPEC2006 applications and the effectiveness of the proposed dynamic tuning policy.

6.3.1 Performance Improvement with TLS

Figure 6.3 compares the sequential and the parallel execution times of the speculatively parallelized loops in the SPEC2006 applications. The figure shows the parallel execution time of the main loop in each application normalized to the loop's sequential execution time, for the 2 and the 4 thread versions. The normalized execution time is split to show the fraction of the useful (TX-Useful) and the aborted (TX-Abort) transactional execution time. We see that TLS improves the performance for some applications, while frequent speculation failures lead to performance degradation in others. Speculative parallelization improves the execution time of the main loop for 2-thread applications such as MCF (0.87), MILC (0.8), HMMER (0.89), and SPHINX3 (0.59), while it degrades the performance of the main loop in H264 (1.15) and LBM (3.88). For the 4-thread versions, TLS improves the performance in the case of MCF (0.8), MILC (0.88), and HMMER (0.91), and degrades the performance for H264 (1.38), LBM (3.93), and SPHINX3 (1.07). The time taken to complete the useful work decreases as the number of threads increases due to the increased parallelism. However, we also see an increase in the fraction of the aborted transactional execution time as the number of threads increases. These results show that TLS implemented using HTM does not scale well as the number of threads increases.

6.3.2 Causes of Speculation Failure

Figure 6.4 shows fraction of the useful and aborted transactions in the parallelized SPEC2006 applications. The figure shows the percentage of the successful transactions, and the percentage of the transactions aborted due to memory conflicts, order inversions, read/write buffer overflows, and other unknown reasons. The data is shown for both the 2 and the 4 thread applications. Almost all the transactions are aborted in H264 and LBM, which results in the applications performing poorly with both 2 and 4 threads. MCF, MILC, HMMER, and SPHINX3 show a performance improvement with TLS due to their relatively low abort rates. However, they do not scale well since the transaction abort rate in these applications increases significantly when going from 2 to 4 threads.

Examining the reasons for the transaction aborts, we see that almost all of the aborts in the 2 thread applications, with the exception of MCF and MILC, are due to memory conflicts. Even in the case of MCF, the transaction aborts due to memory conflicts are

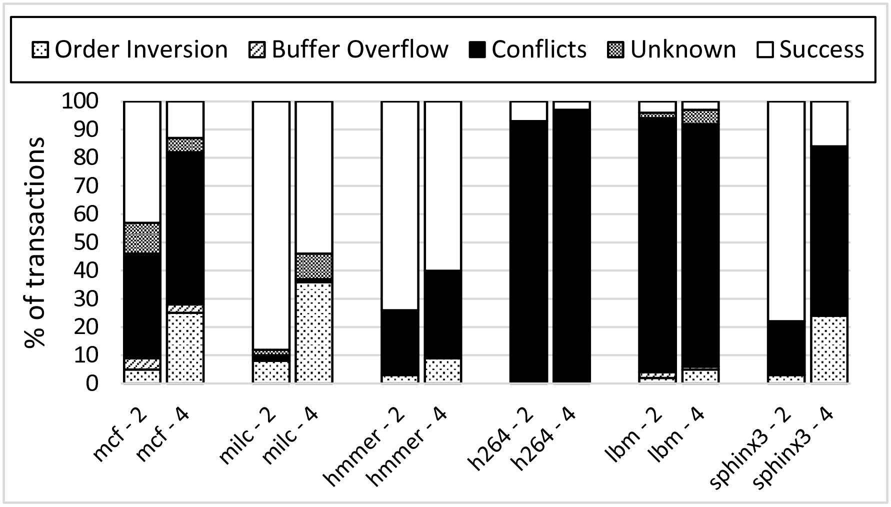


Figure 6.4: The percentage of transactions aborted in the speculatively parallelized versions of the evaluated SPEC2006 applications for 2 and 4 threads. The transaction abort rate is split to show the fraction of the total abort rate caused due to different reasons.

Application	TLS perf. w/o data sync.	TLS perf. w/ data sync.
HMMER - 2 threads	7.16	0.89
HMMER - 4 threads	8.76	0.91

Table 6.2: Impact of data synchronization on the TLS performance of HMMER.

much larger than the aborts due to other causes. The percentage of the transaction aborts due to order inversion increases for most of the applications when run with 4 threads. This is expected, as there are more speculative threads running in parallel in the 4 thread version which increases the probability of the transactions committing out of order. Even with the increase in the percentage of the transaction aborts due to order inversion, memory conflicts continue to be the dominant cause of aborts in the 4-thread versions of MCF, HMMER, H264, LBM and SPHINX3. Overall, MCF, MILC, HMMER, and SPHINX3 are amenable to TLS among the evaluated applications. However, they still suffer from frequent memory conflicts, and the lack of support for ordered transactions in Intel TSX; especially in the 4-thread versions.

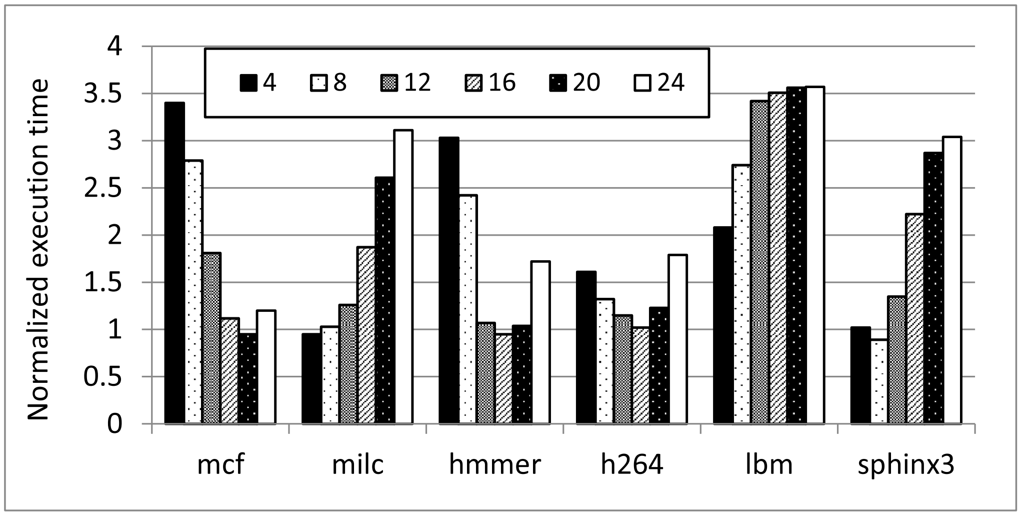


Figure 6.5: The execution times of the 2 thread TLS versions of the SPEC2006 applications normalized to their sequential execution times with various iteration block sizes.

We focus on HMMER since it is a special case where synchronization was effective in eliminating a frequent cross-iteration dependence (Section 5.3.2). Table 6.2 shows the execution times of the speculatively parallelized main loop of HMMER, with and without data synchronization, normalized to its sequential execution time. The data is shown for both the 2 and the 4 thread versions. The frequent inter-iteration data dependence in the main loop of HMMER (line 9 of Listing 5.3) leads to almost all the transactions aborting due to memory conflicts. Consequently, the TLS performance of the application is much worse than its sequential performance. However, once the frequent inter-iteration dependence is serialized using synchronization, we see that the TLS performance improves greatly. We only focus on the TLS version of HMMER with data synchronization in this section.

6.3.3 Amortizing Transactional Execution Overhead

Figure 6.5 shows the execution times of the 2-thread TLS versions of the evaluated applications normalized to their sequential execution times for the various iteration block sizes. The data is shown for the iteration block sizes of 4, 8, 12, 16, 20, and 24 iterations.

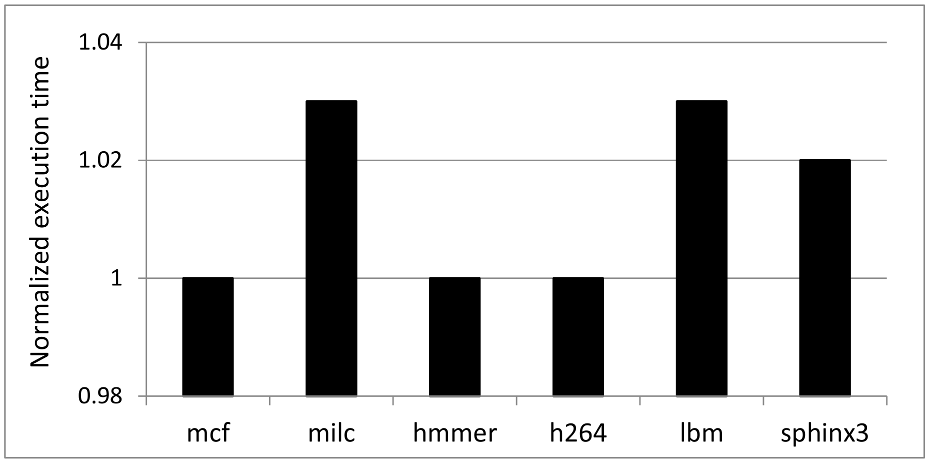


Figure 6.6: The execution times of the TLS versions of the SPEC2006 applications run with 1 thread normalized to their sequential execution times.

The results show that the choice of the iteration block size has a big impact on TLS performance. We see that each application has an optimal iteration block size which varies depending on its characteristics. A very small iteration block size cannot amortize the overhead of starting and terminating a transaction. However, a very large iteration block size increases the probability of a memory conflict among the parallel threads. Moreover, a very large transaction size can also lead to the per thread read/write buffers overflowing as the number of the unique memory locations accessed within the transaction increases. The iteration block size also has an impact on the amount of the memory conflicts due to false sharing, as described in Section 5. Manually computing the optimal iteration block size for an application is challenging. A detailed study on automatically choosing the optimal iteration block size for an application is beyond the scope of this paper. Prior work has demonstrated that the thread size for speculative parallelization can be chosen using compiler-based [39, 34, 33, 44, 64, 59] and hardware techniques [63, 45].

Figure 6.6 evaluates the overhead of transactional execution in the evaluated applications. Figure 6.6 shows the execution times of the evaluated applications speculatively parallelized using 1 thread, normalized to their sequential execution times. The speculatively

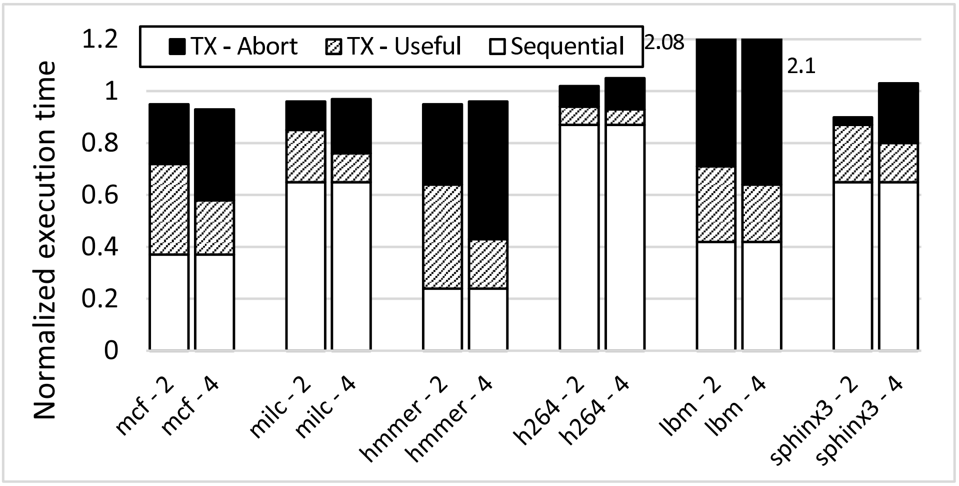


Figure 6.7: Parallel execution times of the evaluated SPEC2006 applications, normalized to their corresponding sequential execution times for 2 and 4 thread versions.

parallelized versions use HTM to execute the code. However, since they are sequential, there are no transaction aborts due to conflicts or buffer overflows. Any overhead is due to starting and terminating a transaction. Since we use the optimal iteration block size for each application, there are no transaction aborts due to read/write buffer overflows. From the figure we see that the overhead of starting/ending a transaction is minimal in all the applications. It is negligible in the case of MCF, HMMER, and H264. It is 2% in the case of SPHINX3, and 3% for MILC and LBM. These results show that the overhead of starting and ending transactions can be amortized in these applications by selecting the appropriate iteration block size.

6.3.4 Overall Performance

Figure 6.7 compares the total sequential and parallel execution times of the SPEC2006 applications. The figure shows the total execution time of the TLS version of each application normalized to its sequential execution time, for 2 and 4 threads. The normalized execution time split to show the fraction of the time spent outside the parallelized main loop (sequential), the fraction of the time spent doing useful work within the speculatively parallelized loop (TX-Useful), and the fraction of the time wasted to transaction aborts (TX-Aborts). The results show the same trend as in Figure 6.3. Speculative parallelization is faster for 2-thread applications such as MCF (0.95), MILC (0.95), HMMER (0.95), and SPHINX3 (0.89), while it is slower than the sequential version for H264 (1.02) and LBM (2.08). For the 4 thread applications, TLS improves the performance in the case of MCF (0.92), MILC (0.97), and HMMER (0.96), while it degrades the performance for H264 (1.05), LBM (2.1) and SPHINX3 (1.02). Although TLS degrades the performance of the main loop in both LBM and H264, the overall performance of H264 is much better than LBM since the execution coverage of the main loop is much lower in H264 (Table 6.1). As expected, we see that the execution time spent outside the main loop does not change between the 2-thread and the 4-thread versions. Overall, the time taken to complete the useful work in the parallelized loop decreases with the increasing number of threads. However, in all the cases we see that the time wasted due to the transaction aborts increases as the number of threads increases. This trend is also reflected in the transaction abort rates shown in Figure 6.4, and explains why we see minimal improvement in the performance of the parallelized applications when going from 2 to 4 threads.

6.3.5 TLS Performance with Dynamic Tuning

Figure 6.8 shows the total execution times of the TLS versions of the SPEC2006 applications parallelized using the dynamic tuning mechanism (Section 5.3.3), normalized to their sequential execution times. The figure shows that the simple dynamic tuning scheme correctly predicts the applications where speculative parallelization degrades the performance and disables TLS in such cases. In these applications, the main loop is executed sequentially using a single thread. For applications where TLS improves the performance, the dynamic tuning policy does not disable TLS. We find that the aggressive abort rate threshold of 90% is required since TLS fairs better than sequential execution in applications

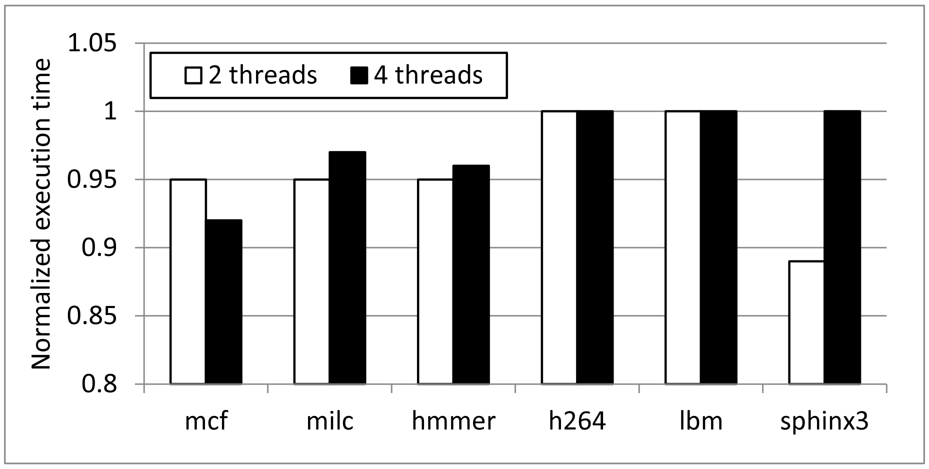


Figure 6.8: Parallel execution times of the dynamically tuned SPEC2006 applications normalized to their corresponding sequential execution times for 2 and 4 thread versions.

such as MCF, MILC, and SPHINX3 (2-thread version) despite a considerably high transaction abort rate. Moreover, since the TLS performance of an application is likely to be better than that of the sequential version unless it experiences very frequent transaction aborts, using an aggressive threshold helps disable TLS only in cases where TLS degrades the application performance significantly. We note that the threshold of the dynamic tuning policy can be easily varied to a more conservative value if desired.

6.4 Summary

In this work we demonstrated the performance potential of implementing TLS using the HTM support offered by the Haswell processor. HTM support provides mechanisms for efficient detection of inter-thread data dependence violations and mechanisms for rollback when speculation has failed. These mechanisms are also the basis for implementing efficient TLS. However, HTM does not provide the hardware support to ensure that all the threads can commit in a pre-determined order and does not support synchronized inter-thread data communication. Thus, we proposed software mechanisms to emulate these behaviors.

Under TLS, we parallelized a set of sequential applications from the SPEC2006 benchmark suite, which are not amenable to parallelization using traditional parallelizing compilers, and showed that TLS yields a performance improvement: we achieved more than 10% performance improvement in the parallelized code regions, and 5% or more overall performance improvement with 2 threads in the following applications: MCF, MILC, HMMER, and SPHINX3. Our evaluation also shows that using software synchronization to reduce frequent inter-iteration dependences can greatly help reduce the number of memory conflicts in certain applications. The relatively small overall program improvement is partially because of the lack of a TLS compiler that can automatically select all the code regions amenable to speculative parallelization. Therefore, we are only able to parallelize a relatively small fraction of the total execution. The proposed dynamic tuning policy accurately identifies and disables TLS in the applications where TLS degrades performance. Under the dynamically tuned TLS policy, we achieve an average improvement of 15% in the parallelized code regions, and an overall performance improvement of 4% with 2 threads. However, TLS implemented using the limited hardware support available on the current HTM does not scale well as the number of threads increases. Our analysis reveals that the most common cause of speculation failures is memory conflicts, followed by aborts due to transactions committing out of order. Our work demonstrates the feasibility of parallelizing sequential applications, that are otherwise not parallelizable, through Thread-Level Speculation on real multicore processors.

Chapter 7

Conclusion and Future Work

Progress in computer architecture in tandem with advances in semiconductor technology lead to the persistent improvements in the performance of uniprocessor systems until the early 2000s. At the turn of the century, the processor industry made a decisive shift towards multicore processors in order to extract more performance while consuming less power. Multicore and many-core processors have become pervasive ever since. As a response to this trend, software developers are aiming to extract more parallelism from a wide range of existing and emerging applications. Numerous programming and compiler tools have been proposed in order to aid the development of multithreaded software capable of harnessing the processing power of modern multicore processors. There have also been numerous works aimed specifically at improving the performance of multithreaded applications on multicore processors. For example, the authors in [46] characterize the cache behavior of multithread applications with the aim of improving their cache performance on multicore processors. Recently, processor vendors have introduced hardware support for transactional execution in order to further aid software developers extract sufficient parallelism from applications. Although primarily intended for improving the performance of multithreaded software, the introduction of hardware support for speculative parallel execution opens up avenues of research in other areas. This dissertation explores the use of transactional execution in order to address performance and correctness challenges in a broad range of software.

In the first half of the dissertation, we describe a correctness problem that can arise due to a discrepancy in the memory consistency models of the host and guest systems in a cross-ISA system virtualization environment. We demonstrate the need for memory consistency model emulation support in parallel system emulators. We propose two mechanisms for emulating memory consistency models: memory fence insertion and execution transactionalization. We discuss the tradeoffs involved, and compare the performance impact of the two mechanisms. We show that, on microprocessors with adequate hardware support for transactionalizing instruction sequences, transactional execution is a viable alternative to memory fence insertion for certain workloads. Therefore, we propose and evaluate a hybrid approach that dynamically determines whether to emulate the memory consistency model by inserting fence instructions or through transactional execution. The proposed hybrid technique outperforms both the fence insertion mechanism and the transactional execution approach.

In the second half of the dissertation, we describe how transactional execution support in multicore processors, in the form of Hardware Transactional Memory (HTM), can be leveraged to improve the performance of sequential applications which cannot be parallelized using traditional parallelization techniques. We explore how a previously proposed speculative parallelization technique for sequential applications, Thread-Level Speculation (TLS), can be realized using HTM support available on current processors. We begin by highlighting the similarities and the differences between the hardware features required for TLS, and those guaranteed by HTM. We then demonstrate software techniques to implement TLS using HTM support. Further, we illustrate software optimizations to improve the performance of our proposed TLS implementation. Our evaluation of TLS on a set of sequential applications which cannot be automatically parallelized shows that TLS improves the overall program performance by up to 11% compared to the sequential version.

7.1 Future Directions

The work presented in this dissertation demonstrates the use of transactional execution to address a broad range of software issues. The work in this dissertation opens multiple avenues of research and development in both the industry and academia. Based on our experiences, we present a concrete list of features, if implemented in future HTM offerings, we believe would help the adoption of HTM and TLS in a wide range of software. We also discuss future research problems in the areas of transactional memory and speculative parallelization.

7.1.1 Recommendations for Future HTM Implementations

Although the HTM support in current processors greatly aids in the development of multithreaded software, the following features can promote the use of HTM in a wider range of software.

Hardware Performance Counters for Speculation Management : Current processors implement a limited set of hardware performance counters for monitoring and tuning speculative execution. Existing hardware performance counters supply information about a limited set of speculation failure causes, and provide information about the transaction abort rate, but not the actual CPU cycles spent in successful and failed transactions. A richer set of hardware performance counters can help developers gain more insight into the performance of their software and identify the bottlenecks more easily. A richer set of hardware performance counters can also help in the dynamic management of speculation aggressiveness in both HTM and TLS execution modes.

Hardware Support for Improving TLS Efficiency : Incremental hardware features on top of the already available support for transactional memory can improve the efficiency of TLS. Hardware support that allows speculative threads to commit in a pre-determined order can eliminate speculation failures due to order inversion in TLS. Support that allows synchronized data communication can avoid speculation failures due to frequently occurring data dependences. Finally, hardware support that enables word level conflict detection can eliminate speculation failures due to false sharing. The support of these features will eliminate the need to use software techniques, such as those presented in this thesis, to realize TLS.

7.1.2 Directions in Transactional Memory and Speculative Parallelization Research

The introduction of hardware support for transactional execution in current processors presents new avenues of research.

Software Support for Speculative Parallelization : Until hardware support for TLS is available in processors, the software techniques for implementing TLS presented in this dissertation can facilitate the parallelization of legacy and emerging sequential applications which cannot be parallelized through traditional parallelization techniques. The development of an optimized TLS library which uses current HTM support will enable easy adoption of TLS by software developers. The development of a compiler that automatically parallelizes the code regions in sequential applications using hardware support for speculative execution will further help in the widespread deployment of TLS.

Leveraging HTM to Address Challenges in Lock-Free, Concurrent Software : Parallel algorithms and data structures that do not rely on traditional mutex locks in order to enforce mutual exclusion among parallel threads have been researched extensively by prior works. These lock-free, concurrent programs aim to extract more performance by eliminating lock contention among parallel threads in an application. When implemented on current multicore processors with relaxed memory models, many of these data structures and algorithms require the use of memory fences in order to enforce certain ordering constraints between memory operations for correct execution. This thesis presented the use of transactional execution as an alternative means of enforcing memory ordering in processors implementing relaxed memory models. Future research into the use of transactional execution in concurrent data structures and algorithms can help to further improve the performance of these applications.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

[2] Haitham Akkary and Michael A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 226–236, 1998.

[3] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, October 2008.

[5] Colin Blundell, E Christopher Lewis, and Milo Martin. Deconstructing transactional semantics: The subtleties of atomicity. *Annual Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, pages 48–55, 2005.

[6] Bochs, 2014. http://bochs.sourceforge.net.

[7] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel.* Oreilly & Associates Inc, 2005.

[8] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: Checking consistency of concurrent data types on relaxed memory models. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 12–21, New York, NY, USA, 2007. ACM.

[9] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 225–236, New York, NY, USA, 2013. ACM.

[10] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: Bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 278–289, New York, NY, USA, 2007. ACM.

[11] Jaewoong Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 279–289, Feb 2008.

[12] Marcelo Cintra and Josep Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, 2002.

[13] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM.

[14] Dave Dice and Nir Shavit. Tlrw: Return of the read-write lock. In *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '10, pages 284–293, New York, NY, USA, 2010. ACM.

[15] Edsger Wybe Dijkstra. Cooperating sequential processes, technical report ewd-123. 1965.

[16] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. Pqemu: A parallel system emulator based on qemu. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 276–283, 2011.

[17] Zhao-Hui Du, Chu-Cheow Lim, Xiao-Feng Li, Chen Yang, Qingyu Zhao, and Tin-Fook Ngai. A cost-driven compilation framework for speculative parallelization of sequential programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 71–81, New York, NY, USA, 2004. ACM.

[18] Yuelu Duan, Xiaobing Feng, Lei Wang, Chao Zhang, and Pen-Chung Yew. Detecting and eliminating potential violations of sequential consistency for concurrent c/c++ programs. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 25–34, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Pradeep K. Dubey, Kevin O'Brien, Kathryn M. O'Brien, and Charles Barton. Single-program speculative multithreading (spsm) architecture: Compiler-assisted fine-grained multithreading. In *Proceedings of the IFIP WG10.3 Working Conference on Parallel Architectures and Compilation Techniques*, PACT '95, 1995.

[20] Xing Fang, Jaejin Lee, and Samuel P. Midkiff. Automatic fence insertion for shared memory multiprocessing. In *Proceedings of the 17th Annual International Conference on Supercomputing*, ICS '03, pages 285–294, New York, NY, USA, 2003. ACM.

[21] Manoj Franklin and Gurindar S. Sohi. The expandable split window paradigm for exploiting fine-grain parallelsim. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, 1992.

[22] Marco Galluzzi, Enrique Vallejo, Adrián Cristal, Fernando Vallejo, Ramón Beivide, Per Stenström, James E. Smith, and Mateo Valero. Implicit transactional memory in kilo-instruction multiprocessors. In *Proceedings of the 12th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'07, pages 339–353, Berlin, Heidelberg, 2007. Springer-Verlag.

[23] Manish Gupta and Rahul Nim. Techniques for speculative run-time parallelization of loops. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–12, 1998.

[24] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (tcc). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, 2004.

[25] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (tcc). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 1–13, New York, NY, USA, 2004. ACM.

[26] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 58–69, 1998.

[27] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, 2004.

[28] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[29] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The ibm blue gene/q compute chip. *Micro, IEEE*, 32(2):48–60, March 2012.

[30] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, 1993.

[31] Intel Corporation. Intel architecture instruction set extensions programming reference, 2013. http://intel.com.

[32] Christian Jacobi, Timothy Slegel, and Dan Greiner. Transactional memory architecture and implementation for ibm system z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 25–36, Washington, DC, USA, 2012. IEEE Computer Society.

[33] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, 2004.

[34] Troy A. Johnson, Rudolf Eigenmann, and T. N. Vijaykumar. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, 2007.

[35] Tom Knight. An architecture for mostly functional languages. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, 1986.

[36] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, FMCAD '10, pages 111–120, Austin, TX, 2010. FMCAD Inc.

[37] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, C-28(9):690–691, 1979.

[38] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. Efficient sequential consistency using conditional fences. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 295–306, New York, NY, USA, 2010. ACM.

[39] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. Posh: A tls compiler that exploits program structure. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, 2006.

[40] Yangchun Luo, Venkatesan Packirisamy, Wei-Chung Hsu, Antonia Zhai, Nikhil Mungre, and Ankit Tarkas. Dynamic performance tuning for speculative threads. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[41] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[42] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the arm and power relaxed memory models. 2012.

[43] Pedro Marcuello and Antonio González. Clustered speculative multithreaded processors. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 365–372, 1999.

[44] Pedro Marcuello and Antonio González. Thread-spawning schemes for speculative multithreading. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, 2002.

[45] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. In *Proceedings of the 12th International Conference on Supercomputing*, ICS '98, 1998.

[46] R. Natarajan and M. Chaudhuri. Characterizing multi-threaded applications for designing sharing-aware last-level cache replacement policies. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*, pages 1–10, Sept 2013.

[47] Ragavendra Natarajan and Antonia Zhai. Leveraging transactional emulation for memory consistency model emulation. *ACM Trans. Archit. Code Optim.*, 2015.

[48] Jeffrey T. Oplinger, David L. Heine, and Monica S. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, PACT '99, 1999.

[49] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *In TPHOLs09: Conference on Theorem Proving in Higher Order Logics, volume 5674 of LNCS*, pages 391–407. Springer, 2009.

[50] V. Packirisamy, A. Zhai, Wei-Chung Hsu, Pen-Chung Yew, and Tin-Fook Ngai. Exploring speculative parallelism in spec2006. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 77–88, April 2009.

[51] Venkatesan Packirisamy. *Efficient architecture support for thread-level speculation.* PhD thesis, UNIVERSITY OF MINNESOTA, 2009.

[52] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[53] L. Porter, Bumyong Choi, and D.M. Tullsen. Mapping out a path from hardware transactional memory to speculative multithreading. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 313–324, Sept 2009.

[54] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society.

[55] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, 2002.

[56] Carl G Ritson and Frederick RM Barnes. An evaluation of intel's restricted transactional memory for cpas. *Communicating Process Architectures 2013*, pages 271–291, 2013.

[57] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, April 1988.

[58] Jim Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[59] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture*, ISCA '95, 1995.

[60] Standard Performance Evaluation Corporation. The SPEC CPU2006 Benchmark Suite. http://www.specbench.org.

[61] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, 2000.

[62] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The stampede approach to thread-level speculation. *ACM Trans. Comput. Syst.*, 23(3):253–300, August 2005.

[63] J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, HPCA '98, 1998.

[64] T. N. Vijaykumar and Gurindar S. Sohi. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, 1998.

[65] Christoph von Praun, Harold W. Cain, Jong-Deok Choi, and Kyung Dong Ryu. Conditional memory ordering. In *Proceedings of the 33rd Annual International Symposium*

on Computer Architecture, ISCA '06, pages 41–52, Washington, DC, USA, 2006. IEEE Computer Society.

[66] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, and Binyu Zang. Coremu: A scalable and portable parallel full-system emulator. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 213–222, New York, NY, USA, 2011. ACM.

[67] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 24 –36, june 1995.

[68] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 171–183, New York, NY, USA, 2002. ACM.

[69] Antonia Zhai, Christopher B. Colohan, J. Gregory Steffan, and Todd C. Mowry. Compiler optimization of memory-resident value communication between speculative threads. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 39–, Washington, DC, USA, 2004. IEEE Computer Society.