SYSTEM CHALLENGES AND OPPORTUNITIES FOR TRANSACTIONAL MEMORY

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

JaeWoong Chung

June 2008

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Christos Kozyrakis) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Kunle Olukotun)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Hector Garcia-Molina)

Approved for the University Committee on Graduate Studies.

Abstract

Recent trends in architecture have made chip multiprocessors (CMPs) increasingly common. CMPs provide programmers with an unprecedented opportunity for parallel execution. Nevertheless, the key factor limiting their potential is the complexity of parallel application development using primitives such as locks and condition variables. While transactional memory (TM) is a technique that helps with parallel program development by transferring concurrency management from the user to the system, there remain unsolved design challenges to building commercial TM systems and unexplored opportunities to use TM beyond concurrency control.

This thesis addresses the challenges to building an efficient and practical TM system and explores the opportunities for using it to support system software and to improve important system metrics other than performance. The contributions of the thesis are the followings. First, we analyze the common case transactional behavior of multithreaded programs and draw key insights towards building efficient TM systems tuned for the common case. Second, we present a TM virtualization mechanism that virtualizes all aspects of transactional execution (time, space, and nesting depth) to build correct and cost-effective TM systems for uncommon case. Third, we suggest a practical solution to use TM for correct execution of multithreaded programs within DBT frameworks. Fourth, we present a hardware-assisted memory snapshot system using TM to allow for algorithmic simplicity, easy code management, and performance at the same time. And last, we propose a scheme to accelerate software solutions for reliability, security, and debugging with hardware resources for TM.

Acknowledgements

It is the luck of my lifetime to meet Christos and to have him as my adviser. He is technically solid, kind to make a great effort to help me improve, fun to make people around him happy. Our collaboration at Stanford has been great and I hope we can work together for more exciting things after my graduation.

Kunle, our papa K, is such an optimist who knows how to laugh out whatever problems. His great insights on computer engineering helped me deal with many technical challenges. I really appreciate his support for me during all the four years at Stanford.

Hector's classes about database systems provide me wisdom that have been developed for transactional processing. It helped me understand better the technical challenges for TM. I love his smile that soothes people.

The TCC group has been at the center of my life at Stanford. All my works could not have been done without the support from the group member. Chi helped me with his diligence and talents for most of my works where he was my grammar master, graphic master, experiment master, and more. Austen joined the group almost at the same time as I did and spent a significant amount of time together to get our simulator working. Brian has made a great managerial effort for the group including the great robot system. Nathan is one of the closest person to a real genius and is always kind to help me solve technical problems. Jarad is someone I never stop bothering with English questions that he kindly answer with fun. Tayo's great smile and bone-like muscle convinces me that I never want to pick up a fight with him. Sewook is a friend who can ask a favor without hesitation and I hope to be such a close person to him as well. Woongki is an examplary student who gets the job done

and never miss out his words. Sungbak and I have known for a long time and it was my true pleasure to see him in our group yet again. I miss Hassan for his energy and Lance for his monk-like fun life style.

Hari and Mike are great buddies of mine. We exchanged ideas about technical problems and shared a fun time drinking a lot. They are the ones that I am willing to do crazy things with. Jacob and Suzy are smart and kind. I only regret that I did not spend more time with them.

My parents in-laws and bother in-law are great supports of mine. We leave on different sides of the world, but I always miss them.

I love my younger bother. I miss drinking with him and have a boy chit-chat about girls and all small things we care together. I hope he finds his love of lifetime soon. My parents dedicate their life for me and bother. I am all what they give me. Their love is what makes me strong even in the worst time.

My love, my heart, my wife. Ever since I married her, my life has changed to be better, happier, and brighter. I promise that I will do whatever it takes to protect a blessed life of ours.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Difficulty of Parallel Programming

In the past two decades, the performance of microprocessors has been improving exponentially thanks to clock frequency increases and the use of wide-issue, out-of-order processing techniques. However, further speedup of single-core CPUs is constrained by the limitation of instruction-level parallelism (ILP), power consumption, and design complexity [158, 153]. Recent trends in architecture have made chip multiprocessors (CMPs) increasingly common [72, 74, 86]. CMPs provide programmers with an unprecedented opportunity for parallel execution. Abundant hardware parallelism in CMPs allows parallel programs to improve performance through thread-level parallelism (TLP). Nevertheless, the key factor limiting their potential is the complexity of parallel programming using primitives such as locks and condition variables.

Locks synchronize accesses to shared data by providing mutual exclusion among concurrent threads. By acquiring a lock, a thread obtains the ownership of data associated with the lock and blocks the other threads that want to access the data. The difficulty of lock-based schemes is that they demand programmers deal with a tradeoff between functional correctness and scalable performance. Coarse-grain locking is easy to use but introduces unnecessary serialization that degrades system performance. On the other side, fine-grain locking scales better but can easily lead to deadlocks, livelocks, or races. Lock-based code does not automatically compose. Before making a

library call, a programmer must fully understand the locking behavior of the library code. Finally, locks are not robust: a thread that fails while holding a lock may make inconsistent memory updates and block other threads from making forward progress.

1.2 Transactional Memory

Transactional memory (TM) is a technique that helps with parallel software development by transferring concurrency management from the user to the system [65, 60, 90, 14]. A transaction encloses a group of instructions and executes them in an atomic and isolated way. With TM, a programmer simply declares that code segments operating on shared data should execute as atomic transactions. It is easy for programmers to reason about the execution of a transactional program since the transactions are executed logically sequentially according to a serializable schedule.

TM systems execute multiple transactions in parallel with optimistic concurrency control as long as they do not conflict. Two transactions conflict if they access the same address and one of them writes. If they conflict, one of them is aborted and restarts. A transaction starts with register checkpointing to save the old register values when the transaction is aborted. Transactional writes are isolated from shared memory by maintaining an undo-log or a write-buffer (data versioning). Memory accesses are tracked in order to detect read/write conflicts among transactions. If a transaction completes without conflicts, its updates are committed to shared memory atomically. If a conflict is detected between two transactions, one of them rolls back by restoring the register checkpoint and either by applying the undo-log or by discarding the write-buffer.

1.3 Challenges and Opportunities for Transactional Memory

While there have been proposals to implement high-performance TM systems using hardware support [6, 60, 76, 90, 119], there remain unaddressed important challenges

to building an efficient and practical TM system. TM systems use hardware resources to accelerate the basic functions such as data versioning and conflict detection. For efficiency's sake, it is important to balance the performance gain obtained from the hardware resources against the additional hardware cost. To accelerate TM systems in a cost-effective manner, we should tune the resources for the common case. Hence, analysis on the common behavior of TM programs is an essential part of designing efficient TM systems. On the other hand, the hardware resources for acceleration are limited. For transactional memory (TM) to achieve widespread acceptance, it is important to provide a practical solution to deal with uncommon cases of applications with transactions that exceed the capabilities of the hardware.

Once an efficient and practical TM system is available, we can use it to support system software and to improve system metrics other than performance (e.g. reliability and security). Specifically, we are interested in the following three problems that can be solved efficiently with TM systems. First, dynamic binary translation (DBT) has become a versatile tool that addresses a wide range of system challenges while maintaining backwards compatibility for legacy software. However, DBT frameworks may incorrectly handle multithreaded programs due to races involving updates to the application data and the corresponding metadata maintained by the DBT. Second, the lack of concurrency in system software modules such as garbage collector or memory analysis tools prevents us from fully exploiting the abundant resources in chip-multiprocessors (CMPs) in order to minimize their overhead on application. Ideally, system code could be easily changed to use spare cores in a CMP. However, parallelization is not trivial in practice because programmers must deal with the complications of concurrency management in complex system software and the interactions with the applications they serve. Third, reliability, security, and debugging are as important as performance. There have been efforts to develop techniques that independently provide system support for reliability, security, or debugging in software and hardware [138, 114, 117, 36, 144, 160, 121, 122]. Software solutions are flexible, but have performance issues. Hardware solutions require special hardware resources to accelerate a specific feature, which makes them hard to be adopted to commercial systems due to the lack of generality.

1.4 Contributions

In the thesis, we address the challenges to building an efficient and practical TM systems and explore the opportunities for using them to support system software and to improve important system metrics other than performance (e.g. reliability and security). The challenges and opportunities dealt with in the thesis are the following.

- **Analysis of the common case behavior of TM programs**

  We present an analysis of the common case behavior of transactions in existing parallel programs. We translate existing synchronization primitives to transactions and measure the key metrics of transactional execution. Based on these metrics, we make suggestions for values of key architectural parameters that tune hardware TM systems towards the common case behavior.

- **Design of the eXtented Transactional Memory (XTM) for TM virtualization**

  We present *eXtended Transactional Memory (XTM)*, a software-base TM virtualization system that virtualizes all aspects of transactional execution (time, space, and nesting depth). It is implemented in software using virtual memory support in the operating system. XTM operates at page granularity. It uses private copies of the pages overflowing the cache to buffer transactional state until the transaction commits. It also uses snapshots of the pages to detect interference between transactions. We also describe two enhancements, XTM-g and XTM-e, to XTM that use some hardware support to address key performance bottlenecks.

- **Multithreading support for dynamic binary translation (DBT) using TM**

  We present a practical solution that uses TM to support correct execution of multithreaded programs within DBT frameworks. The DBT uses memory transactions to encapsulate the data and metadata accesses in a trace, within one atomic block. This approach guarantees correct execution of concurrent threads of the translated program, as TM mechanisms detect and correct races.

- **Improving software concurrency with hardware-assisted memory snapshot**

  We present *MShot*, a hardware-assisted memory snapshot system to allow for algorithmic simplicity, easy code management, and high performance at the same time. We use the hardware resources in TM systems to accelerate memory snapshots of arbitrary lifetime that consist of multiple disjoint memory regions.

- **Accelerating software solutions for reliability, security, and debugging using TM**

  We propose a scheme to accelerate software solutions for reliability, security, and debugging with the hardware resources for transactional memory. We provide four acceleration primitives on top of TM hardware resources and use them to accelerate the solutions by targeting just the common case behavior.

1.5 Organization

The thesis is organized as follows. Chapter 2 explains TM programming model and implementation options. It also presents the outline of our approach for the challenges and opportunities for TM systems as well. Chapter 3 describes the analysis of the common case behavior of TM programs and shows the analysis results. Chapter 4 presents eXtended Transaction Memory (XTM) to virtualize TM systems. Chapter 5 explains how the correctness issue of dynamic binary translation frameworks with multithreaded programs is fixed using TM. Chapter 6 describes and evaluates fast memory snapshot built on top of TM hardware. Chapter 7 explains how TM hardware resources are used to accelerate software solutions for reliability, security, and debugging. Chapter 8 concludes the thesis.

Chapter 2

Transactional Memory

In this chapter, we review programming models and implementation options for TM systems. A base TM system design with hardware resources is described in detail. Then, we summarize our approach to address the challenges to building an efficient and practical TM system and to exploit the opportunities to use TM for other than parallel programming.

2.1 Transactional Programming Model

Transactional memory (TM) is a technique that helps with parallel software development by transferring concurrency management from the user to the system [65, 60, 90, 14]. With TM, a programmer simply declares that code segments operating on shared data should execute as atomic transactions. Multiple transactions may execute in parallel and the TM system is responsible for synchronization management.

2.1.1 Semantics

A transaction encloses a group of instructions and executes them in an atomic and isolated way. Atomicity means that either all or no instructions in the code block are executed. Isolation means that intermediate results of transactions are not exposed to any other code. A TM system schedules transactions logically in a serializable

manner so that they look like being executed one by one at a time.

Differently from database systems [49, 107], TM systems have to deal with not only transactions but also non-transactional memory accesses. TM systems with weak isolation guarantee transactional isolation only between code running within transactions [77]. Non-transactional memory writes can be visible to transactions allowing multiple reads from the same address in a transaction to return different values. TM systems with strong isolation provides isolation between transactions and non-transactional memory accesses as well. Read repeatability within a transaction is guaranteed as well as atomicity of transactional writes. Strong isolation makes it easy for programmers to reason the execution of transactional programs since non-transactional accesses are ordered with transactions in a sequential schedule as well [77].

2.1.2 Interface

There are different types of interfaces to provide TM features in programming languages. Implicit transactions allow programmers to declare only the transaction boundaries [60, 90]. Hardware or the compiler are responsible for tracking all memory locations or objects accessed with transactions. Explicit translations require programmers either to manually annotate memory accesses to be part of the transaction or to use transactional objects that interact with TM systems [24, 7].

A low-level transactional interface can be provided as part of instruction set architecture (ISA) in hardware [60, 90]. Such an interface is fast but less flexible. It is intended as a building block for higher level APIs rather than for direct use by programmers. A higher level interface in software can be provided either as part of language specification [24] or through a TM library [87]. In any case, a high-level specification should clearly explain how TM primitives interact with existing parallel programming primitives and memory management mechanisms [67].

Some interfaces provide only basic TM primitives such as atomic block while others may provide advanced TM primitives. Software handlers are triggered at the events related to transactions such as transaction commit, abort, and conflict.

Nested transaction support allows a transaction to start inside another transaction. Some primitives interact with the internals of TM systems for performance improvement [84].

2.2 Implementation Options

TM systems perform data versioning and conflict detection to support transactional execution. A transaction starts with register checkpointing. Transactional writes are isolated from the rest of the system by maintaining an undo-log or a write-buffer. Memory accesses are tracked in order to detect read/write conflicts among transactions. If a transaction completes without conflicts, its updates are committed to shared memory atomically. If a conflict is detected between two transactions, one of them rolls back by restoring the register checkpoint and either by restoring the undo-log or by discarding the write-buffer.

2.2.1 Software vs. Hardware

There have been proposals to implement TM systems in software or hardware. Software TM systems (STM) implement all TM bookkeeping in software by instrumenting read and write accesses within a transaction [45, 62, 125]. Software read and write barriers are instrumented per memory access to track the memory addresses accessed by transactions. Depending on the case, the overhead of the barriers can range from 40% to 7x for each thread in the parallel program. [1, 125, 23]. In addition, most high-performance software TM systems do not provide strong isolation without additional compiler support [77].

Hardware TM systems (HTM) implement both data versioning and conflict detection by modifying cache and coherence protocol. It is crucial for performance to have such hardware resources to accelerate transactional execution. HTM systems do not require software barriers for read and write accesses within a transaction, and thus have minimal overhead for each thread [60, 90]. Hardware performs all bookkeeping

transparently. Transactional reads and writes are recorded using cache-line metadata bits or signature filters in order to help with conflict detection across concurrent transactions. Some designs support nested transactions and fast context switching of transactional code using multiple sets of metadata bits or transaction IDs [84, 91]. New versions of data produced by pending transactions are isolated either by buffering them in the cache or logging old values. All hardware TM systems provide strong isolation by default.

More recently, there have been proposals for hybrid TM systems [23, 126, 136]. While they still require read and write barriers, hybrid systems use hardware signatures or additional metadata in hardware caches in order to drastically reduce the overhead of conflict detection in software transactions. They are faster than software TM systems but slower than hardware TM systems. Some of them provide strong isolation as well [23].

In this thesis, we focus mostly on hardware TM systems since they provide superior performance and strong isolation.

2.2.2 Eager vs. Lazy Data Versioning

TM systems separate a new version produced by an uncommitted transaction from the committed version. If a transaction commits, the new version becomes the last committed version in an atomic way. If it is aborted, the new version is discarded safely. There are two types of data versioning: eager and lazy.

TM systems with eager versioning write transactional data in place and log the last committed version. The logged version is written back to the original place at abort. For the new version to be in place, an exclusive right should be acquired to prevent the other transactions and non-transactional memory accesses from reading the version without checking for conflicts. The contention manager should deal with the potential livelock when two transactions gain attempt to exclusive rights on two data items in opposite orders. In addition, eager versioning may make TM systems less resilient to faults by putting uncommitted data in place.

TM systems with lazy versioning buffer transactional data and keep the last committed version in place. The buffered data are flushed to memory at commit. Transactions are aborted simply by invalidating the buffered data. This scheme is better in terms of error recovery, especially for software TMs that may not have complete control of all code that the program may use (e.g., library code).

2.2.3 Pessimistic and Optimistic Conflict Detection

TM systems detect conflicts in order to produce a serializable schedule of transactions. A read-set and a write-set per transaction are maintained to track transactional memory accesses. The read-set contains memory addresses read by the transaction and the write-set contains the addresses written. Conflicts among transactions are detected by comparing the sets. They are two types of conflict detection: pessimistic and optimistic.

With pessimistic conflict detection, transactional memory accesses are first checked against the read-/write-sets of the other transactions. The accesses are allowed to complete only when they do not lead to a conflict. Two accesses conflict if they access the same address and one of them is a write operation. The pessimistic scheme detects conflicts early so that the contention manager takes actions as early as possible in order to minimize the amount of work lost by the aborting transactions. When being implemented in hardware, pessimistic conflict detection can be integrated easily with the cache coherence protocol [90].

With optimistic conflict detection, transactional execution splits into three phases. In the read phase, all reads and writes are allowed without checking for conflicts. Writes are buffered until transactions finish. In the validation phase, transactions are validated by comparing read-/write-sets and potential conflicts are detected. In the commit phase, all buffered writes are committed in place. Optimistic conflict detection avoids the overhead of detecting conflicts at each memory access. It also allows for a larger set of serializable schedules compared to the pessimistic approach. On the other hand, the validation phase is required to be done in an atomic way, which can pose a performance issue.

2.3 Base Hardware TM system

Hardware TM (HTM) systems provide hardware support for basic TM tasks in order to reduce their overhead and make them transparent to transactional applications. While there are several HTM variations, HTMs are quite similar in their structure. Atomicity of register state is supported using a hardware mechanism such as a shadow register file that can take, restore, or release a hardware checkpoint within a few clock cycles. Atomicity of memory state is supported by using the cache as an undo log or a write buffer for the store accesses within transactions. If a transaction commits successfully, the undo log is discarded or the write buffer is applied to the shared memory. If the transaction is rolled back, the undo log is applied or the write buffer's contents are discarded. Isolation is provided by detecting conflicts between transactions on coherence events. A conflict occurs when two transactions access the same data and at least one of them performs a write. HTMs track the addresses read (read-set) or written (write-set) by a transaction either by extending each cache line with metadata bits or using separate signatures [13]. Conflicts are detected by checking the addresses in coherence messages from other threads against the read-set or write-set.

Early HTMs expose these mechanisms in a monolithic manner, with programs only marking transaction boundaries. Recent proposals expose the basic mechanisms to software [84]. Software can control when register checkpoints are taken, which stores are versioned, and which addresses are inserted into the read-set or write-set. Software is also invoked on conflict detection to determine if the transaction should roll back or continue. Some HTMs provide enhanced support for nested transactions in the form of additional shadow register files and separate cache metadata bits or filters [84, 91]. The additional mechanisms allow independent tracking of atomicity and isolation for nested transactions.

Figure 2.1 shows the hardware resources for the base hardware TM system that we refer to in the thesis. It has extra register files for register checkpointing in the beginning of transactions. A pair of a read bit and a write bit per cache line are added for tracking transactional memory accesses. Multiple pairs of the bits are added to

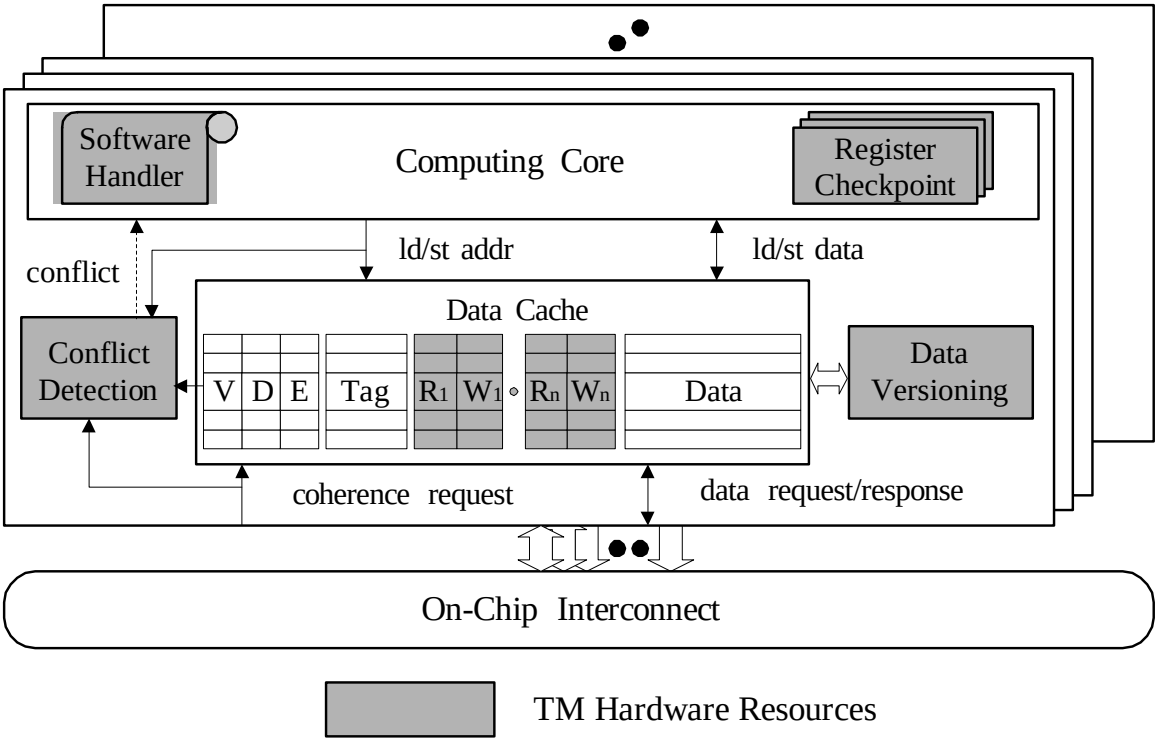


Figure 2.1: The hardware resource for a base hardware TM system.

support nested transactions. It uses cache as buffer for transactional data and perform lazy data versioning. Conflict detection mechanism is integrated with cache coherence protocol. Memory requests from the other cores are snooped and checked against the metadata bits for pessimistic conflict detection. Software handlers are registered and invoked for transaction commit, abort, and conflict.

2.4 Challenges and Opportunies for TM systems

Supporting basic TM features using hardware mechanisms is just the first step toward commercial TM systems. There are design challenges to building an efficient and practical TM system. On the other hand, there are great opportunities to use it beyond concurrency control. The challenges and opportunities of interest for the thesis are the following.

2.4.1 How can we build an efficient hardware TM system?

It is a well-known wisdom in computer architecture to make common case fast in order to improve system efficiency. Hence, it is a prerequisite for an efficient TM design to understand how transactions behave in common case.

There are the key metrics that describe the common case behavior of transactions in multithreaded programs. We would like to know how large transactions are in order to size the buffer requirement for transactional data in the cache hierarchy. The number of instructions within a transaction is important to decide how much cost is required for the basic TM primitives such as transaction begin and end. The depth of nested transactions tells us how many pairs of read bits and write bits are needed to support enough levels of transaction nesting. In Chapter 3, we present a methodology to measure the key metrics and use them to make suggestions for efficient hardware TM system design.

2.4.2 How can we build a practical TM system?

Regardless of the common case characteristics, the hardware resources will always be limited and insufficient to cover all possible cases of programs. For transactional memory (TM) to achieve widespread acceptance, it is important to deal practically with uncommon cases as well. A practical TM system should guarantee correct execution even when transactions exceed scheduling quanta, overflow the capacity of hardware caches or physical memory, or include more independent nesting levels than what is supported in hardware.

There are architectural challenges for TM virtualization. It is wasteful to add significant hardware resources for uncommon cases since they will be used rarely. How can we virtualize TM systems at a low hardware cost? We may have a special version of code with software barriers to run long-lived transactions with software TM systems that do not have virtualization issues. However, this makes the TM virtualization mechanism not transparent to applications by demanding two versions of code. How can we provide TM virtualization in a way transparent to applications? In Chapter 4, we present *eXtended Transactional Memory (XTM)*, a software-base

TM virtualization system that virtualizes all aspects of transactional execution (time, space, and nesting depth) transparently to applications.

2.4.3 Can we use TM to support system software?

Dynamic binary translation (DBT) has become a versatile tool that addresses a wide range of system challenges while maintaining backwards compatibility for legacy software. DBT has been successfully deployed in commercial and research environments to support full-system virtualization [147], cross-ISA binary compatibility [28, 124], security analysis [34, 73, 102, 118, 130], debuggers [100], and performance optimization frameworks [10, 22, 142]. However, DBT frameworks may incorrectly handle multithreaded programs due to races involving updates to the application data and the corresponding metadata maintained by the DBT.

It is easy to understand that transactions can help fix the atomicity issues between user data accesses and DBT metadata accesses. However, they are the key questions to answer for instrumenting transactions. At what granularity, should transactions be instrumented? Is it per instruction, per basic block, or per trace? What if there are user lock, user transactions, or I/O operations? Are software TM systems sufficient from the performance perspective? In Chapter 5, we present a practical solution that deals with various challenges to using TM for correct execution of multithreaded programs within DBT frameworks.

Lack of concurrency in software modules such as garbage collector prevents us from fully exploiting abundant parallelism in chip-multiprocessors (CMPs). Especially, linked with user code, they degrade the overall system performance despite the existence of underutilized cores. Ideally, system code could be easily changed to use multiple cores. However, parallelization is not trivial in practice because programmers must deal with the complications of concurrency management in complex system software.

It is ideal if we can find programming primitives that can simplify system software development and handle the interaction with user code. However, are there such primitives? Moreover, if we find one, can we implement it in a cost-effective manner?

In Chapter 6 we present *MShot*, a hardware-assisted memory snapshot system using TM hardware resources to allow for algorithmic simplicity, easy code management, and performance at the same time.

2.4.4 Can we use TM to improve important system metrics other than performance?

Reliability, security, and debugging are as important as performance. While chip-multiprocessors (CMPs) take advantage of Moore's law to continue the growth of raw computing power available, this progression does little to improve the other metrics. In the past decades, there have been efforts to develop techniques that independently provide system support for reliability, security, or debugging in software and hardware [138, 114, 117, 36, 144, 160, 121, 122]. Software solutions are flexible to provide various features at no hardware cost. However, they have performance issues since they have to rely on the commodity hardware primitives. Hardware solutions add special hardware resources to accelerate a specific feature. However, the more highly tuned the resources are for the feature, the harder they are to be adopted to commercial systems due to the lack of generality.

Are there common hardware primitives that various software solutions can benefit from for performance improvement? Can we build such primitives in a cost effective manner to make it easy for them to be adopted to commercial systems? How do they interact with the software solutions? In Chapter 7, we propose a scheme that provides common hardware acceleration primitives built on top of TM hardware resources to accelerate software solutions for reliability, security, and debugging.

Chapter 3

Common Behavior of Multithreaded Programs

3.1 Introduction

It is important to analyze the common case behavior of TM programs to design an efficient TM system based on hardware or software. In this chapter, we measure the key metrics of transactions in multithreaded programs and extract architectural parameters to tune TM systems for common case.

The problem with such an analysis is that there are few transactional programs available for analysis as transaction-based programming is still in early development phases. Transactional memory researchers need to tune their implementations for the common-case behavior of parallel programs written with transactions. On the other hand, the application developers need efficient and complete transactional memory systems in order to port a significant volume of applications. This is symptomatic of a classic chicken and egg problem.

We address this problem by studying a wide range of existing parallel applications and analyzing the common-case behavior likely to be seen in their future transactional versions. This is reasonable since programmers have already identified parallelism and synchronization points in these programs; hence, we can evaluate the program behavior as if parallelism and synchronization are coded with transactions.

We study 33 parallel programs from a wide range of application domains written with Java threads, C and threads, or OpenMP. We carefully examine the primitives used for concurrency control in these programming models and re-interpret their meaning in a transactional context in order to define transaction boundaries. This approach allows us to evaluate where transactions could be used either for non-blocking synchronization or speculative parallelization. Once transactions have been identified, we can measure common-case characteristics such as the length of transactions, size of read- and write-set, frequency of nested transactions, and the frequency of I/O accesses within transactions. These characteristics provide key insights on the support necessary for buffering, committing, and nesting in hardware and software TM systems.

This chapter is organized as follows. Section 3.2 outlines the key metrics of TM programs that characterize common-case application behavior. In Section 3.3, we present our experimental environment for analyzing multithreaded applications from a transactional point of view. Section 3.4 presents the analysis results when transactions are used for non-blocking synchronization. Section 3.5 presents the analysis results when transactions are used for speculative parallelization. Section 3.6 discusses related works and Section 3.7 concludes the chapter.

3.2 Key Metrics

To build an efficient transactional memory system, a designer must tune several components to common case application behavior. In this section, we review some of the critical application characteristics and how they can influence design decisions.

What are the common transaction lengths? Transactions can restart due to dependency conflicts or during context switches. Re-executing transactions after restart incurs significant waste. To quantify the cost of retrying transactions and to balance overhead associated with transaction creation, termination, and checkpointing, the average transaction length is important. Additionally, small transactions would require transaction creation/destruction and register checkpointing be fast hardware mechanisms. This study measures the distribution of transaction lengths

in instructions.

What are the buffering requirements? To support transactional execution, the hardware provides buffering resources to track read- and write-set. While virtualizing transactional memory (that is, allowing transactional state to overflow in main or even virtual memory) is necessary for completeness [120], it is important to know how frequently this mechanism will be used. If it is rarely used, it should be as simple as possible. A major design point of any transactional system will then be the amount of physical transactional buffering it provides. Furthermore, transactional data is tracked at a given granularity: too fine and the design will incur unnecessary storage overheads; too coarse and excessive false conflicts will result.

What is the write-set to computation ratio? Regardless of the versioning mechanism (eager or lazy), the write-set to computation ratio provides a way to determine commit bandwidth requirements. A lazy mechanism logically flushes all writes to main memory on commit. Eager versioning has additional per-write overhead in order to maintain the undo-log. We measure this ratio to help architects decide if additional mechanisms are needed to amortize or hide the cost of these operations, such as "double buffering" [60].

What support is needed for nested transactions? Early transactional systems do not address nested transactions, but they are necessary to support composable software [84]. This study explores the nesting behavior of current multithreaded applications in order to determine the appropriate level of nested transaction support. We measure the nesting depth and other metrics further explained in Section 3.4.4.

How does I/O fit into a transactional environment? One of the most thorny issues in transactional execution today is I/O, more specifically non-idempotent I/O. Such I/O operations cannot be rolled back, but other solutions exist, such as buffering. While this study does not attempt to present a solution, it does attempt to determine whether I/O is a significant problem in most applications. Problematic I/O would occur inside a transaction that cannot be split into smaller transactions. If a transaction can be safely divided, then I/O occurring inside that transaction can be handled by forcing a commit before the I/O call, and executing the I/O within its own atomic transaction.

Can we use transactional memory for speculative parallelism? Because transactions guarantee atomicity, programmers can easily expose parallelism by placing separate items of parallel work into different transactions (e.g., iterations in a parallel loop). If the work is truly parallel, then transactions will commit with few conflicts and speedup will be obtained. If not, transactions will conflict frequently and execution will be very similar to sequential ordering. Using speculative parallelism, programmers will be less conservative in sizing transactions. Therefore, we expect the resources required for transactional execution (those listed above) to increase if speculative parallelism is used. We briefly evaluate this increase in resources.

3.3 Methodology

To study transactional application behavior, we selected 33 multithreaded programs written in widely-used parallel programming models. After collecting a trace of each program, we translated the existing parallel programming primitives into transactional demarcations, and from there, we extracted the common case transactional behavior. This section describes the applications, traces, demarcation, and analysis.

3.3.1 Applications

Table 3.1 shows the multithreaded applications covering four programming models: Java, OpenMP [39], Pthreads, and the Argonne National Laboratory's (ANL) parallel processing macros [81]. Java is becoming more and more prevalent and includes multithreading as part of the language specification. OpenMP is a widely adopted model for semi-automatic parallelization with easy-to-use compiler directives. Pthreads is a widely available multithreading interface for POSIX systems. ANL, used extensively in SPLASH-2, is designed to provide a simple, concise, and portable interface covering a variety of parallel and distributed applications. The key insight of our approach is that the language primitives of these four models mark where the programmer wants synchronization and parallelism; from this information, we can manually extract representative transactions (entirely automatic methods may produce incorrect

code [15]). For example, a critical section protected by a lock and unlock pair may be interpreted as a transaction since this region is required to be executed atomically in isolation from other threads.

The 33 multithreaded applications allow us to study a wide range of application characteristics. Included are not only scientific kernels and applications, but also commercial programs including web servers, web proxies, relational databases, and e-commerce systems. Additionally, the abundant parallelism in graphics and multimedia applications make them ideal candidates for transactional systems. Applications from robotics and artificial intelligence (AI) further extend the coverage of our study.

We obtained the programs from a wide variety of sources. All ANL [81] applications were obtained from the SPLASH-2 benchmark suite [155]. Most of the Java applications are from the Java Grande benchmark suite [69]. `hsqldb` and `PMD` are from the DaCapo benchmark suite [38] and SPECjbb2000 is from the Standard Performance Evaluation Corporation (SPEC) [140]. All OpenMP applications are from either SPEComp [141] or the NASA Advanced Supercomputing (NAS) benchmark suite [95]. Pthread applications are each from a different source: Apache web server is from the Apache website [9], Kingate web proxy and the Tic-Tac-Toe (`uttt`) game were obtained from SourceForge.net [139], BP-vision was available from the University of Chicago website [20], `localize` was available as part of the CARMEN project [145], and the original sequential version of MPEG-2 was provided by the MPEG Software Simulation Group [93], and then parallelized as in Iwata [68].

3.3.2 Transactional Boundaries

The applications use well-defined primitives for multithreading; however, we mapped these primitives into 13 abstract annotations to mark transaction boundaries as shown in Table 3.2. Each abstract primitive represents actual programming interface elements in the four programming models. For example, the abstract primitive Lock represents the `LOCK()` macro in ANL; the opening bracket of a `Synchronized` block in Java; the `pthread_mutex_lock()`, `pthread_mutex_rdlock()`, and `pthread_mutex_wrlock()` in Pthreads; and the opening brackets of `CRITICAL` and `ATOMIC` pragmas and

Prog. Model	App name	Problem size	Source	Domain	Description	Key Algorithm / Data Structures
Java	MolDyn	2,048 Particles	JavaGrande	Scientific	Molecular Dynamics	N-body under Lennard-Jones Potential
	MonteCarlo	10,000 Runs	JavaGrande	Scientific	Finance	Monte Carlo Simulation
	RayTracer	150x150 Pixels	JavaGrande	Graphics	3D Raytracer	3D Ray Tracing
	Crypt	200,000 Bytes	JavaGrande	Kernel	Encryption and Decryption	IDEA (International Data Encryption Algorithm)
	LUFact	500x500 Matrix	JavaGrande	Kernel	Solving NxN Linear System	LU Factorization
	Series	200 Coefficients	JavaGrande	Kernel	First N Fourier Coefficient	Iteration for Fourier Coefficient of f(x) = (x+1)^x
	SOR	1,000x1,000 Grid	JavaGrande	Kernel	Successive Over-Relaxation	Red-Black Ordering on NxN Grid
	SparseMatmult	250,000x250,000 Matrix	JavaGrande	Kernel	Matrix Multiplication	Sparse Matrices
	SPECjbb2000	8 Warehouses	SPECjbb2000	Commercial	E-Commerce	Binary Trees
	PMD	18 Java Files	DaCapo	Commercial	Java Code Checking	Static Code Analysis
	HSQLDB	10 Tellers, 1,000 Accounts	DaCapo	Commercial	Banking with hsql database	Database with JDBC API
Pthreads	Apache	PerChild MPM, 20 Worker Threads	Apache	Commercial	HTTP web server, worker MPM	Thread pool, task queuing
	Kingate	10,000 HTTP Requests	SourceForge	Commercial	Web proxy	Thread pool, task queuing
	Bp-vision	384x288 Image	Belief Propagation	Machine Learning	Loopy Belief Propagation	Efficient Belief Propagation for Early Vision
	Localize	477x177 Map	CARMEN	Robotics	Finding a Robot Position In a Map	Master-Slave Task Assignment
	Ultra Tic Tac Toe	5x5 Board, 3 Step LookAhead	SourceForge	AI	Tic Tac Toe Game	AI Engine with Decision Tree
	MPEG2	640x480 Clip	MPEG S.S.G.	MultiMedia	MPEG2 Decoder	MPEG2
OpenMP	APPLU	30x30x30 Matrix	SPEComp	Scientific	Parabolic / Elliptical PDEs	Dense Matrices
	Equake	380K Nodes	SPEComp	Scientific	Seismic Wave Propagation Simulation	Sparse Matrices
	Art	640x480 Image (c756hel.in)	SPEComp	Scientific	Neural Network Simulation	Adaptive Resonance Theory
	IS	1M Keys	NAS	Scientific	Large-scale Integer Sort	Buckets
	Swim	1,900x900 Matrix	SPEComp	Scientific	Shallow Water Modeling	Dense Matrices
ANL Macros	Barnes	16K Particles	SPLASH-2	Scientific	Evolution of Galaxies	Barnes-Hut; octree
	Mp3d	3,000 Molecules, 50 Steps	SPLASH-2	Scientific	Rarefied Hypersonic Flow	Monte Carlo
	Ocean	258x258 Ocean	SPLASH-2	Scientific	Eddy Currents in an Ocean Basin	Red-black Gauss-Seidel
	Radix	1M Ints., Radix 1024	SPLASH-2	Kernel	Radix Sort	Radix Sort
	FMM	2,049 Particles	SPLASH-2	Kernel	N-body Simulation	Adaptive Fast Multipole Method
	Cholesky	TK23.0	SPLASH-2	Kernel	Sparse Matrix Factorization	Blocked Sparse Cholesky Factorization
	Radiosity	Room	SPLASH-2	Graphics	Equilibrium of Light Distribution	Rapid Hierarchical Radiosity Algorithm
	FFT	256K points	SPLASH-2	Kernel	1-D version of the radix-N2 FFT	Fast Fourier Transfrom
	Volrend	Head-Scaledown 4	SPLASH-2	Graphics	3-D Volumn Rendering	Ray Casting
	Water-N2	512 molecules	SPLASH-2	Scientific	Evolution of a System of Water Molecules	Direct, Cutoff Radius, Predictor Corrector
	Water-Spatial	512 molecules	SPLASH-2	Scientific	Evolution of a System of Water Molecules	Direct, Cutoff Radius

* The results from the TPC benchmarks will be available shortly.

Table 3.1: 33 programs used for the common case behavior analysis.

Category	Abstract Primitive	Atomicity	Demarcation	Description
Task Parallelization	Thread Creation	Non-Critical	BEGIN	Launching new Threads
	Thread Join	Non-Critical	END	Waiting for the End of Threads
	Thread Entry	Non-Critical	BEGIN	The Entry Point of Threads
	Thread Exit	Non-Critical	END	The Exit Point of Threads
	Parallel Loop BEGIN	Non-Critical	BEGIN	Beginning of Auto-Parallelized Loops
	Parallel Loop END	Non-Critical	END	End of Auto-Parallelized Loops
Exclusive Access	Lock	Critical	BEGIN	Beginning of Critical Section
	Unlock	Critical	END	End of Critical Section
Task Synchronization	Barrier	Non-Critical	NEW	Barrier
	Wait	Critical	NEW	Wait for a Signal, then move on
	Notify	Non-Critical	END	Signalling to Waiting Threads
Explicit Communication	I/O	Non-Critical	NEW	I/O Calls
	Flush	Non-Critical	NEW	Flushing cached data to all Threads

Table 3.2: The transactional language primitives. "Critical" means it cannot be split into smaller transactions, and "Non-Critical" means it can be split. BEGIN and END create and destroy transactions and NEW ends the previous transaction and begins a new one immediately.

`omp_locks()` in OpenMP.

We mapped the parallel programming primitives to three transactional primitives. BEGIN starts a new transaction, END ends the current transaction, and NEW causes the current transaction to end and a new one to begin immediately.

To further characterize the transactions created in our study, we categorized them according to their atomicity type. A transaction can be one of two atomicity types: A critical transaction that encapsulates a critical section or a non-critical transaction that consists of any other parallel regions. A critical transaction cannot be split into smaller transactions, since the application's algorithm enforces atomic execution of the code segment mapped to the transaction.

The abstract primitives are divided into four categories, according to their behavior. The first group concerns itself with the proper task parallelization of the application. For example, Thread creation naturally maps to the transaction BEGIN marker; while Thread termination maps to transaction END.

The second group of the primitives is for mutual exclusion, or critical sections. A critical section is a group of instructions executed in an atomic manner, so critical sections map directly to transactions. All four programming models use lock-based

operations to protect critical sections, and we map the acquisition and the release of locks to the beginning and end of transactions, respectively.

The third group of primitives is for synchronization between threads (generally waiting for a condition to be satisfied before moving on to the next phase of execution). Since these primitives set a boundary between two consecutive phases of execution, two transactions are mapped to the code: one before and after the primitive. An interesting observation about the Java conditional wait implemented with `java.lang.Object.wait()` follows: Java requires a thread to obtain a monitor for an object before it calls `wait()`. This requirement can be satisfied by making the call inside a `synchronized` block. The key point is that when `wait()` is called in the block, the thread releases the monitor, which means the `synchronized` block is split into two exclusive regions. To reflect this properly, we map `wait()` to NEW: ending the previous transaction and beginning a new transaction. Without this, commonly-used barrier patterns in code such as SPECjbb2000 will not function properly.

The last group of the primitives provides communication between threads or generates I/O. We use these communication primitives as boundaries between two tasks and split non-critical transactions into two separate transactions, because non-critical transactions can be split arbitrarily without violating atomicity requirements. This is not the case with critical transactions, and we address that issue in Section 3.4.5.

The translation between abstract primitives and transactional demarcations allows us to easily recast the semantics of the original code to a transactional programming environment. Since the markers help express the parallelism of applications with transactions, they can also be considered a new transaction programming interface, in a preliminary form.

3.3.3 Trace-based Analysis

Before tracing the execution of each application, we annotated it with the transactional primitives, as described in the previous section. For the Java applications, we used the Jikes Research Virtual Machine (RVM) 2.3.4 [5] and modified the just-in-time (JIT) compiler to automatically insert the transaction markers. The ANL

applications were also annotated automatically using macros. For the OpenMP and Pthread applications, we simply went through the source code and inserted transaction markers manually. After annotation, we gathered execution traces on PowerPC machines using `amber`, which is part of Apple's Computer Hardware Understanding Developer (CHUD) tools [30].

We analyzed the traces to extract the following transactional characteristics: transaction size, read- and write-set, write-set to computation ratio, transaction nesting, and transactional I/O. To measure transaction sizes in words, lines, and pages, the analyzer simulates 4-Byte words, 32-Byte lines, and 4-KByte pages. The write-set to computation ratio is obtained by dividing the write-set size by the number of instructions. For nested transactions, the analyzer has a stack to push and pop transactional contexts according to BEGIN and END events.

3.3.4 Discussion

Our methodology makes the measurement of transactional properties largely impleme ntation-independent. For example, instead of measuring the number of buffer overflows due to the limitation of a specific buffer, we provide the distribution of transaction sizes for the design and evaluation of transactional memory systems. From the distribution, we can say how many buffer overflows may happen with a given buffer size and what percentage of transactions will be covered with such a buffer. This allows system designers to evaluate performance/cost tradeoffs.

This study doesn't include measurements regarding transaction rollback caused by conflicts. Because they are dynamic, rollbacks can only be detected after selection a schedule among transactions, a conflict detection scheme, and a contention management policy. We could attempt to arrange the extracted transactions in time, but this would amount to an implementation-specific measurement. However, we can infer the impact of conflicts on system performance by measuring the size of critical transactions: under the assumption that critical transactions modify shared data, long critical transactions modify more shared data (thus increasing the probability of rollback) and have higher rollback penalties (because they take longer to re-execute),

creating performance bottlenecks.

When these applications are re-written with Transactional Memory from scratch, the observed characteristics may change. However, the existing versions provide good indicators of where parallelism exists and where synchronization is needed. Hence, it is likely that transactional behavior will be similar.

3.4 Analysis for Non-blocking Synchronization

This section presents results from analyzing the traces looking for common-case, non-blocking transactional behavior, as described in the previous section. Section 3.5 describes our results with speculative parallelization. We do not include OpenMP applications nor `uttt` because they do not use locking primitives and consequently, have no critical transactions. We discuss a different methodology for dealing with non-critical transactions in the next section.

Throughout this section, we present tables with averages taken over whole application groups, e.g., an average over all Java applications. We also select outliers to present alongside the averages, so one table may contain an average for all Java applications as well details about one or two specific applications.

3.4.1 Transaction Lengths

Table 3.3 shows the distribution of critical transaction lengths. As shown in the table, most transactions tend to be small. For most programs, up to 95% of critical transactions have less than 5,000 instructions. However, the distribution of lengths exhibits a long tail, and a small number of transactions become quite large.

ANL applications tend to have small transactions, since they are mostly fine tuned for high scalability; they display small atomic critical regions. `radix` and `fft`, with their few small critical transactions, are typical examples of barrier-oriented applications, rarely using locks. `mpeg2` has large critical transactions because in the main parallel loop, it locks a slice of a video stream and does not release it until completing operations on that slice.

Application	Size in Instructions			
	Mean	50% Tile	95% Tile	Max
ANL avg	256	114	772	16,782
fft	157	157	157	157
radix	9	9	9	9
Java avg	5,949	149	4,256	13,519,488
sparsematmult	2,723	41	34,987	53,736
series	7,756	97	43,250	524,636
Pthread avg	879	805	1,056	22,591
mpeg2	93,694	101,327	167,267	347,339
apache	209	147	233	2,766

Table 3.3: The table shows the average transaction lengths of ANL, Java, and Pthread applications. The applications with unusual characteristics are not included in the average and presented separately.

Most Java applications share a similar distribution of transaction sizes. Transactions tend to be larger than in other programming models due partly to medium sized Jikes RVM critical sections whose main tasks include scheduling, synchronization, class loading, and memory management. Furthermore, the Java applications themselves exhibit even longer transactions.

Apache is an application that exemplifies the behavior of well-tuned commercial applications; most of its critical regions are small to maximize scalability.

Conclusion: The results in this section indicate a wide distribution of transaction sizes across programming model boundaries. In highly tuned parallel applications, transactions can be quite small; thus the overheads associated with starting, aborting, and ending transactions should be small and there should be support for hardware register checkpointing. This means that software transactional memory systems may not be attractive due to their high overheads. Other applications exhibit long transactions—rolling back these transactions due to conflicts or context switches may be expensive and should be avoided.

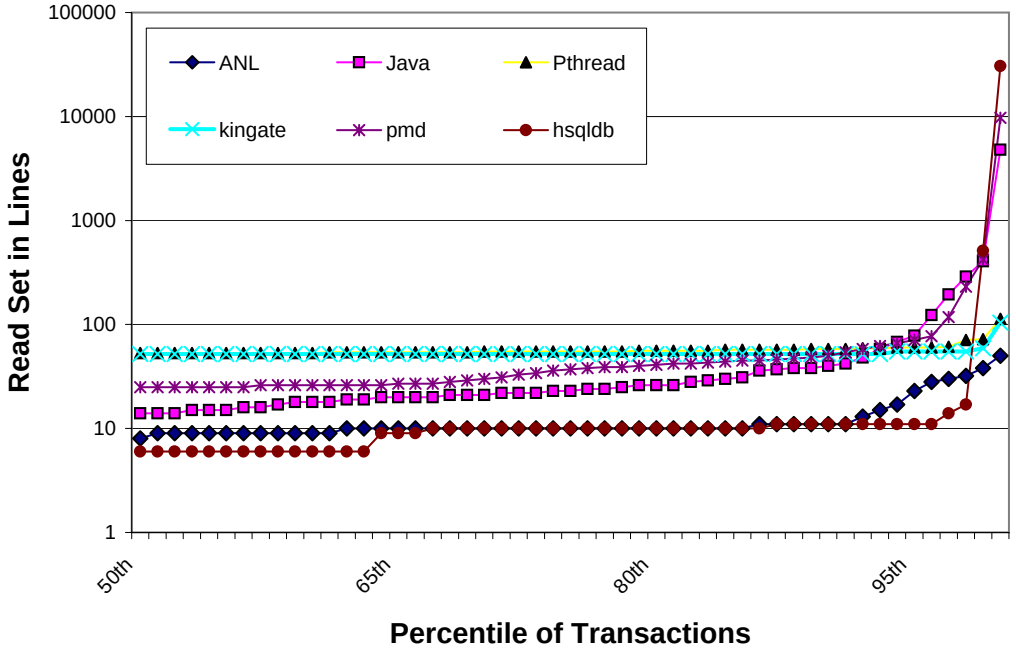


Figure 3.1: Cumulative distribution function of read-set, in 32-Byte lines.

3.4.2 Read- and Write-set Size

Understanding the read- and write-set distributions can help in elucidating the common-case transactional buffering size requirements. Figure 3.1 shows that in most applications, a 16-KByte buffer is sufficient to hold the read set from 98% of critical transactions. If we exclude Java applications, a 2-KByte buffer is sufficient.

The buffering requirement for writes is smaller than that for reads, shown in Figure 3.2, since writes are generally less frequent. A 6-KByte write buffer can deal with up to 98% of critical transactions and if we exclude Java applications and `mpeg2`, a 2-KByte buffer can handle 100% of critical transactions. This observation leads to the conclusion that most transactions are small and buffering can be provided in first-level caches.

An astute reader will point out that even though most transactions are small, the remaining transactions could be the bulk of computation. We measured the number of cycles that a buffer of a specific size can contain all the transactional state without

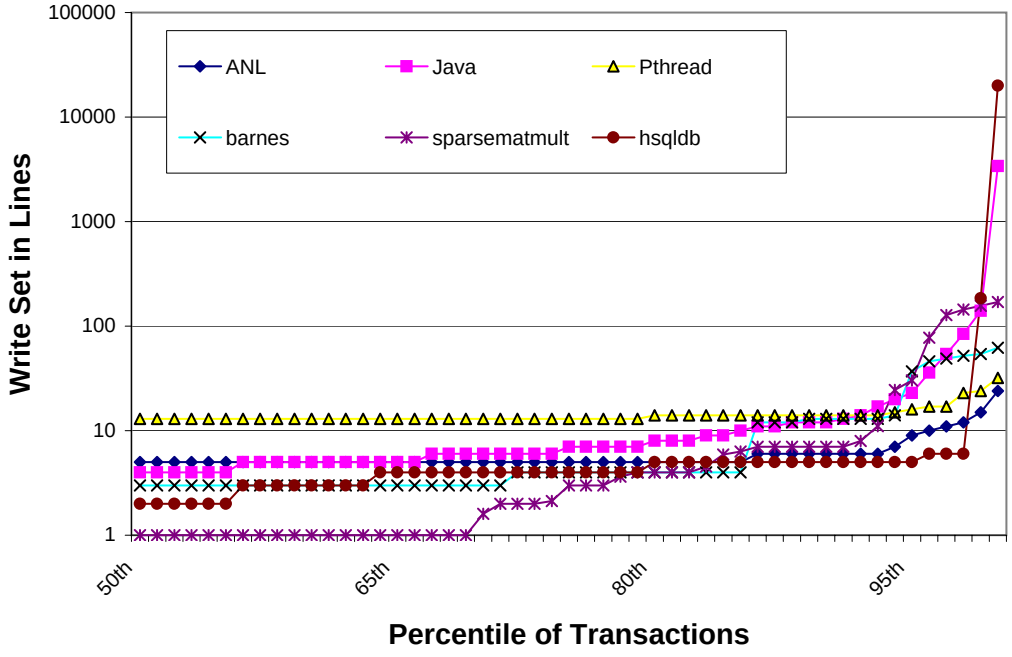


Figure 3.2: Cumulative distribution function of write-set, in 32-Byte lines.

overflowing. This allows us to quantify the time spent in each size transaction. In Figure 3.3, a 24-KByte read buffer covers more than 60% of the execution time in ANL and Pthread applications. For most other applications, a 48-KByte buffer is required to cover 80% or more of the execution time. Figure 3.4 paints a similar picture as Figure 3.3 for the write-set. `hsqldb` has a few transactions with large read- and write-sets because they open JDBC [71] connections and execute SQL queries. These results tell us that most transactions are small, fitting in L1-sized buffers, but hardware designers should support longer-running transactions with L2-sized read- and write-sets. However, virtualization mechanisms can afford to be cost-effective and software-based because they will be rarely used.

Table 3.4 shows the ratio of read- and write-set sizes in words per line and lines per page, which is an interesting characteristic because it hints at the right granularity to track reads and writes. Words are 4 bytes and lines are 32 bytes. For most applications, only 2 out of the 8 words in a cache line are read. The ratio for writes is higher, around 3, meaning nearly half of a cache line is touched. This leads us to

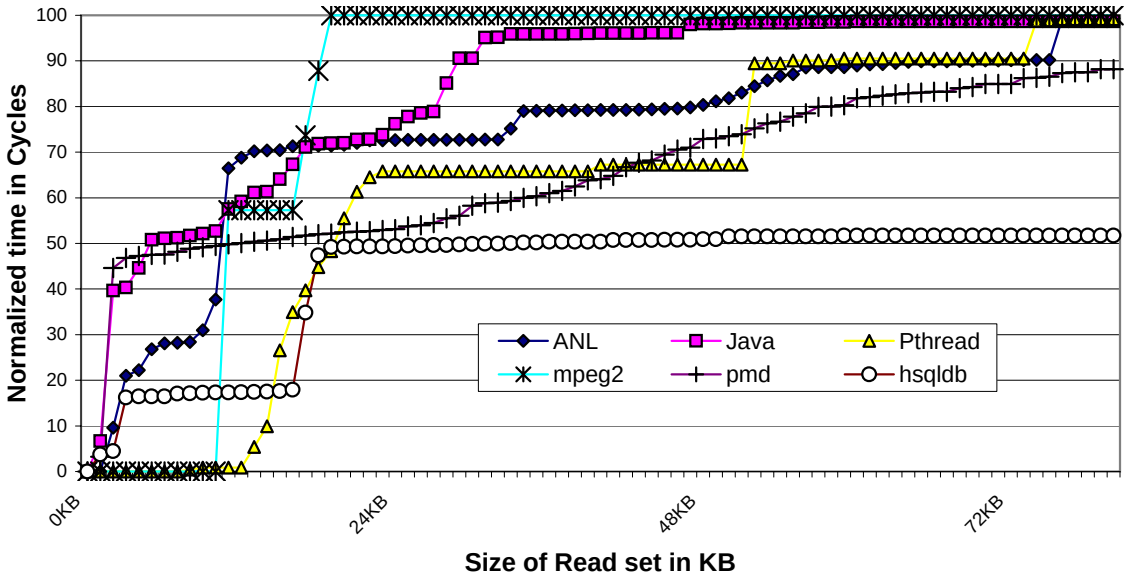


Figure 3.3: Normalized cycles of critical transactions spent with various read-set sizes.

two conclusions: first, a transactional system may suffer from false conflicts between transactions, if the reads and writes are tracked at cache line granularity. Second, since tracking granularity at line level means flushing entire lines for lazy versioning schemes or logging entire lines for eager versioning schemes, tracking state at word level uses over 50% less bandwidth, since only half of each line will be flushed on average.

The ratio of read- and write-set sizes in lines per page is also important for VM-based transactional memory systems that store a significant portion of transactional state in virtual memory. Since those systems have a higher per-page-access overhead than hardware transactional memory systems, an application with high spatial locality in pages will benefit from such systems. In Table 3.4, `mpeg2` shows a high spatial locality, which makes it suitable for VM-based systems. On the other hand, most applications have very sparse access patterns, making VM systems unattractive.

Conclusion: Most transactions are small, so virtualization will be rarely used. But designers should support L2-sized read- and write-sets. Furthermore, techniques that allow transactional state to overflow into virtual memory will not be performance critical and can be implemented in software.

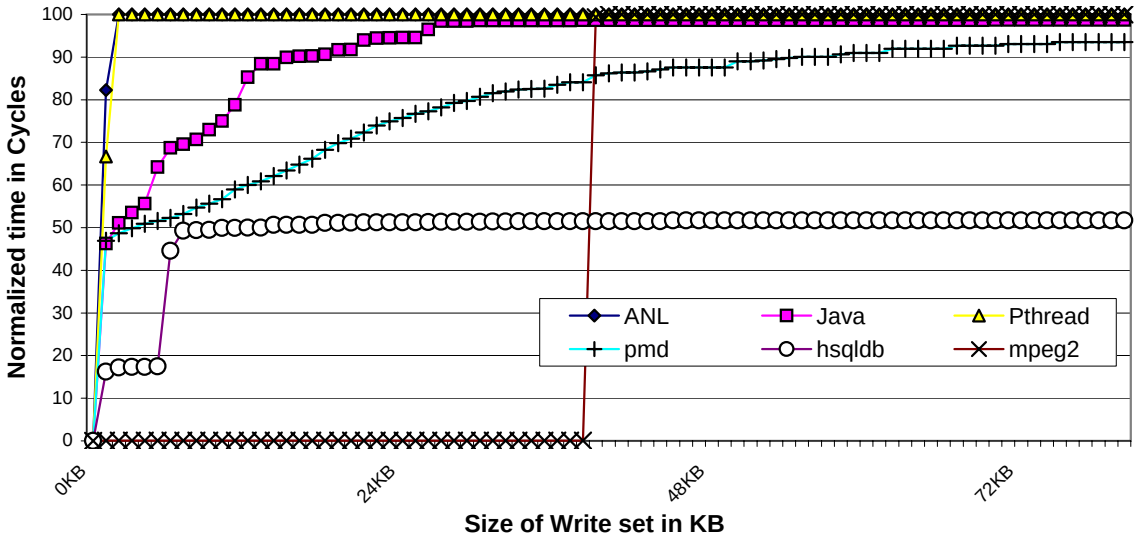


Figure 3.4: Normalized cycles of critical transactions spent with various write-set sizes.

Application	Read Set		Write Set	
	Words / Line	Lines / Page	Words / Line	Lines / Page
ANL avg	1.65	3.69	1.69	4.29
Java avg	2.02	3.83	3.52	8.50
Pthread avg	2.14	6.80	2.30	7.86
sparsematmult	1.32	1.65	2.14	3.00
mpeg2	7.77	71.17	2.30	7.86

Table 3.4: The table shows the ratio of read- and write-set sizes in words per line and lines per page.

3.4.3 Write-set to Computation ratio

Transactional memory systems with lazy versioning generate an inherently bursty traffic pattern and eager versioning systems have overheads per unique address written. Therefore, it is crucial to bound the network bandwidth during commit phases and cache write bandwidth during execution for eager systems. Figure 3.5 shows the ratio of the write-set, in words, to the number of instructions in critical and non-critical transactions. The ratio is under 25% in most critical transactions and around 10% in more than 60% of critical transactions. In Figure 3.6, 95% of non-critical

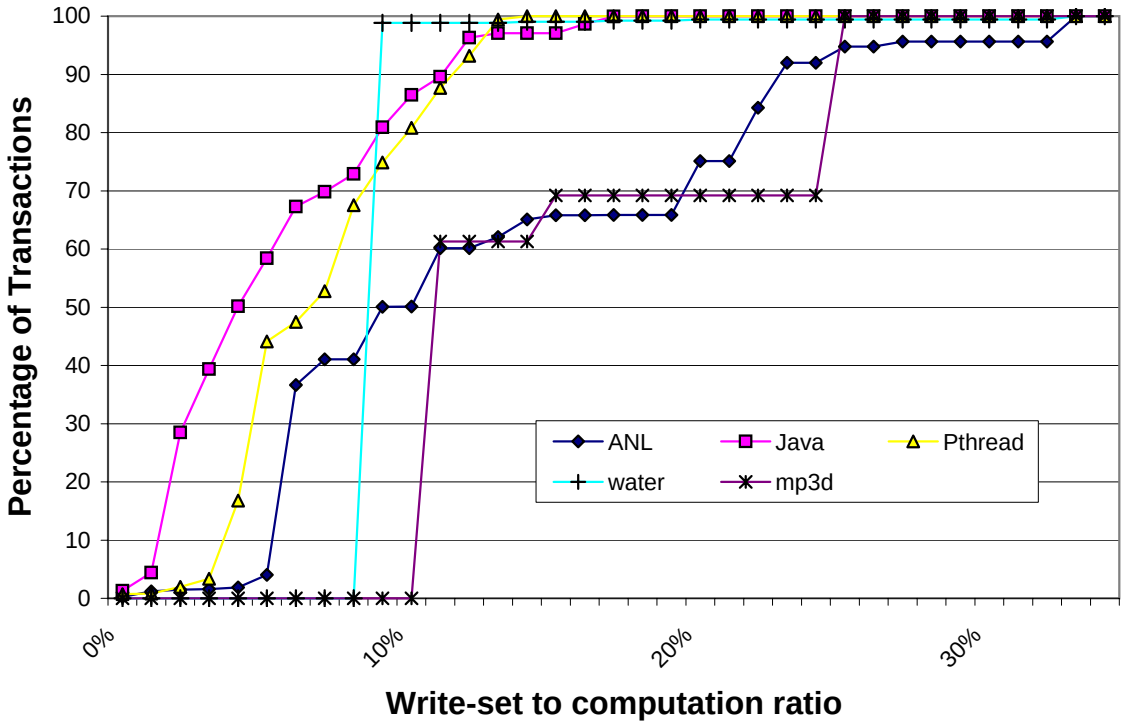


Figure 3.5: Write-set to Computation ratio of critical transactions.

transactions have a lower ratio, about 15%, which identifies an interesting trait of critical sections in multithreaded programs: they tend to perform more writes than non-critical sections. This is natural because shared data are read and written in critical sections, while computation is done in non-critical sections. The high ratio of `radix` is from a high communication-to-computation ratio caused by the key exchange at each radix sorting iteration, as found in [155].

Conclusion: There are many applications with high ratios, hinting that software transactional memory systems are unattractive because it is difficult to amortize the cost of making clean copies before performing writes. Additionally, systems with lazy versioning may need some mechanism to hide the latency of commits, such as double buffering or a two phase commit scheme [60]. Eager versioning systems will also need techniques to avoid excessive cache write bandwidth problems.

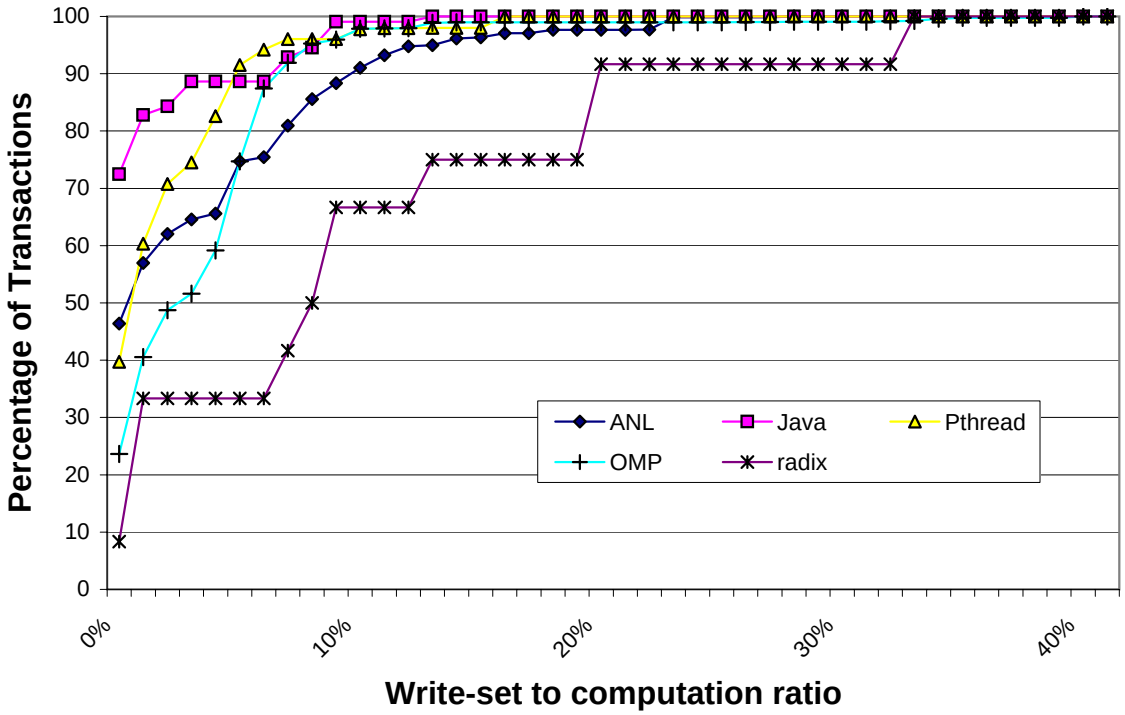


Figure 3.6: Write set to Computation ratio of non-critical transactions.

3.4.4 Transaction Nesting

Large-scale programs are built on external libraries and the resulting multi-layer software structure naturally calls for nested synchronization. For applications written using a transactional programming model, nesting may no longer be a requirement. Table 3.5 shows four characteristics of nested transactions: average size, breadth, depth, and distance. Figure 3.7 shows the definition of each property. Breadth is the number of immediate child transactions. Depth is the level of nesting a transaction has above itself. Distance is the number of instructions between the beginning of a transaction and that of its children. Only applications with more than 1% nested transactions appear in the table.

Only one C application displayed any nesting behavior: `radix` had one nested transaction. All the applications in Table 3.5 are Java applications. Furthermore, the nested transactions described in the table actually come from Jikes RVM, and not

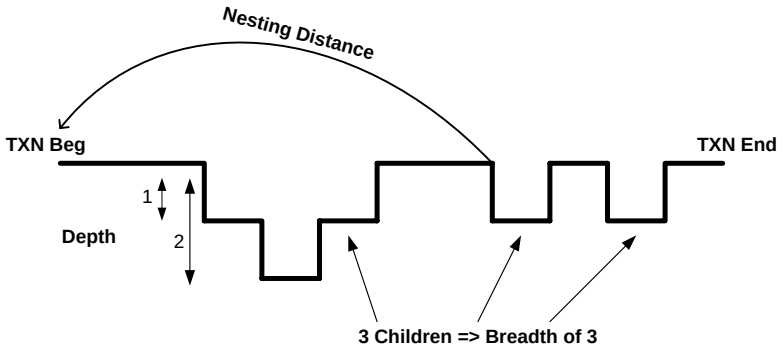


Figure 3.7: The figure explains the definition of nesting depth, nesting breadth, and nesting distance.

the applications themselves. There are several sources of these nested transactions; one is class loading. The Java Virtual Machine defines a tree-like Java class loader structure. While the Jikes RVM finds, loads, and resolves classes, it searches up the tree with nested calls to synchronized methods. If the loaded class is an array, the call stack enters another round of these nested calls. This is the major source of nested transactions with a deep nesting depth. Another source is the optimizing compiler. At first, all source code is compiled by the base compiler without optimization. While running the application, often-run sections of code are recompiled, and there are many locking operations through this process. Thread management and scheduling also generates nested transactions. For example, when a thread is created and enters the ready queue for execution, most of these operations are done with locks.

The table shows that Java applications have an average of 2.2 immediate child transactions. The importance of breadth depends on the way nesting is implemented. If true transaction nesting is supported, the transactional metadata from all nested transactions should be held in buffers up to the commit of the outermost transaction. In this case, the number of buffers required is equal to the number of all nested transactions. If all nested transactions are flattened to the outermost transaction, the breadth and the depth do not matter because no additional buffers are required. The penalty for flattening can be measured by the nesting distance, since a greater distance means conflicts are more costly. In the table, Java applications have a long average distance of 140,000 instructions, which reinforces the necessity of nesting

Application	% of Trans. With Nesting	Nesting Depth (in % of Trans.)			Nesting Breadth (in %)			90% Tile Nested Size	Mean Nesting Distance
		1	2	> 2	1	2	> 2		
moldyn	22	16	42	41	13	7	3	1065	291889
montecarlo	14	99	0	0	0	0	14	144	2784
raytracer	14	36	41	23	7	4	3	1168	99262
crypt	18	45	37	19	10	2	4	1023	56211
lufact	18	39	38	23	12	3	6	1071	87913
series	14	40	51	8	7	2	4	1047	68782
sor	16	48	4	48	9	3	5	1013	75400
sparsematmult	13	87	11	2	6	1	6	155	10440
specjbb	9	63	35	2	1	4	4	1148	58855
pmd	17	19	30	52	8	4	5	2158	659871
hsqldb	1	3	97	0	0	0	1	134	6538826
bp-vision	4	100	0	0	5	0	0	436	165
localize	2	100	0	0	2	0	0	439	641

Table 3.5: The table shows the characteristics of transaction nesting. Only the applications with more than 1% nested transactions are presented.

Application	% of Trans. With I/O	% of Trans. With Rd I/O	% of Trans. With Wr I/O	% of Trans. With Rd then Wr. I/O	% of Trans. With Wr. Then Rd. I/O
water-spatial	5.26	0	5.26	0	0
moldyn	0.35	0.35	0	0	0
raytracer	0.4	0.4	0	0	0
crypt	0.4	0.4	0	0	0
series	0.31	0.31	0	0	0
sor	0.35	0.35	0	0	0
pmd	0.49	0.49	0	0	0
kingate	1.31	0	1.31	0	0
mpeg2	20	20	0	0	0

Table 3.6: The table shows the breakdown of transactions with I/O.

support.

Conclusion: Our analysis presents a mixed picture of the need for hardware based nested transaction support. If our goal is to support all existing applications, then explicit nesting support is useful.

3.4.5 Transactions and I/O

In this study, we count only I/O operations in critical transactions. Since non-critical transactions can be split, we assume that an I/O operation in a non-critical transaction is easily dealt with by ending the transaction and executing the I/O operation in a new transaction. Table 3.6 shows the percentage of transactions with I/O and a

Application	Size in Instructions			
	Mean	50% Tile	95% Tile	Max
equake	244	9	1,134	40,750,634
art	70,062,948	71,978,851	74,117,449	74,824,088
is	129	3	3	19,844,217
swim	62,130	68,467	91,296	91,296

Table 3.7: The table shows the transaction size of speculatively-parallelized programs.

breakdown of those I/O operations. Most applications have few critical transactions with I/O, which is natural because a long I/O operation is unattractive, and usually unneeded, within a critical section. `mpeg2` and `water-spatial` are the only applications with a high percentage of critical transactions with I/O operations. The high ratio of `mpeg2` is due to its algorithm of holding a lock while reading a slice in a video stream. All output operations of `water-spatial` are for printing intermediate results to console. Additionally, we did not observe any reads followed by writes or writes followed by reads within one transaction, thus deadlocks resulting from I/O buffering are unlikely to occur.

Conclusion: I/O is not likely to be a problem in transactional systems since most transactions in which it occurs can be split to accommodate it.

3.5 Analysis for Speculative Parallelization

In Section 3.4, we analyzed mature parallel programs—ones whose critical sections have been fine-tuned to lower contention between threads. Since transactions shift the burden of correctness from the programmer to the hardware, we expect transactional developers to rely more on speculative parallelism instead of carefully identifying parallel regions. Speculative parallelism is identifying roughly parallel regions, enclosing them in transactions, and allowing the system to dynamically resolve conflicts, hopefully resulting in speedup.

To observe the changes in transactional behavior due to this coding paradigm,

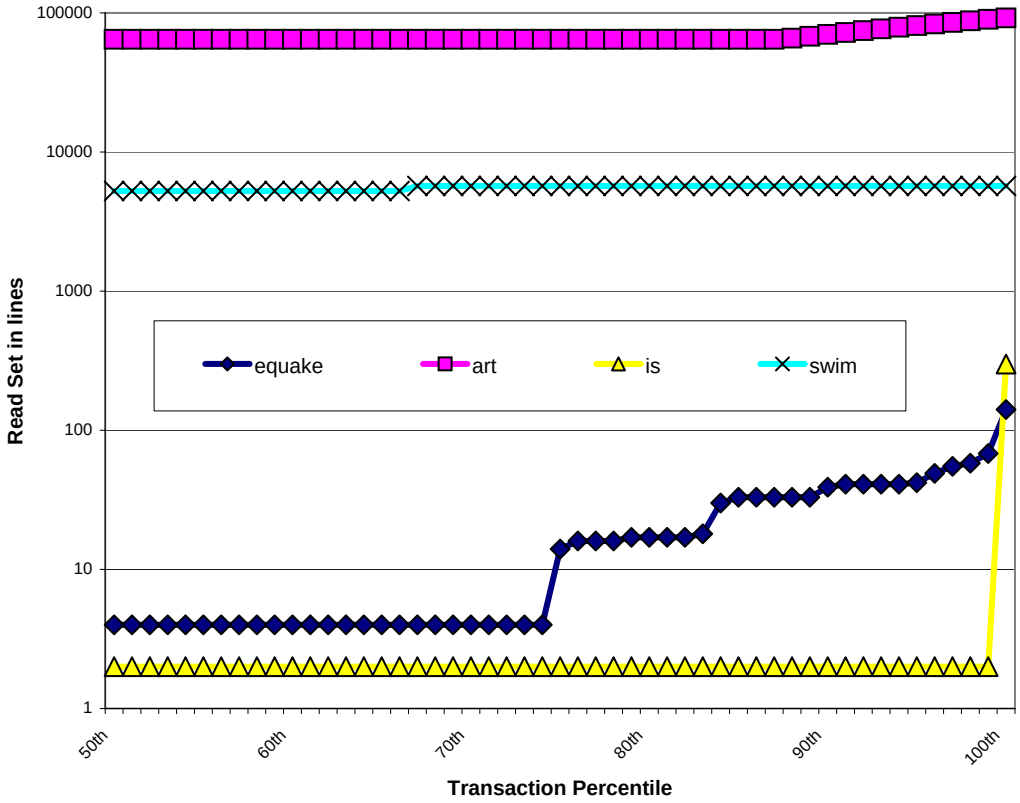


Figure 3.8: Distribution of read-set sizes. Cache lines are 32 bytes.

we re-investigate the OpenMP applications with a different interpretation of transactional primitives. OpenMP `#pragma`s deliver two kinds of information: first, which segments are parallel, and second, information about shared variables (which ones are shared, private, used in reductions). Since classifying variables and protecting them with critical sections is exactly what transactions allow us to avoid, we simply use `omp parallel` loops to identify critical transactions and compare the result from the new interpretation with that from Section 3.4.

Table 3.7 shows sizes of critical transactions for the four OpenMP programs. The studied applications tend to fall into two groups according to their distinctive characteristics shown in Figure 3.8 and Figure 3.9. The flat lines of `art` and `swim` in the figures indicate that the parallel loops are large and of the same size; the source code confirms this. It is obvious that buffering requirements for read- and write-set

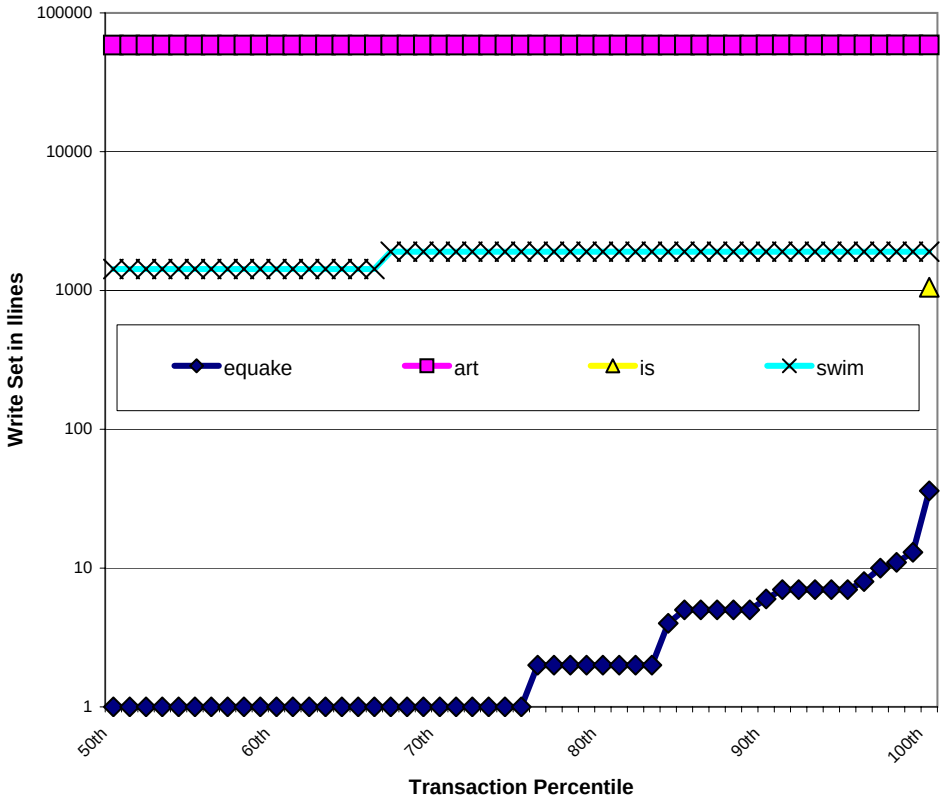


Figure 3.9: Distribution of write-set sizes. Cache lines are 32 bytes.

will be larger for this group than we observed in the previous section. The other group, `equake` and `is` has ascending lines that look similar to those in the previous section, indicating that their loops are mostly small and of various sizes.

With speculative parallelism, we observed larger transactions, and this might lead us to change the way we think about parallelism. In these traditional applications, generally outer loops are parallelized. Thus, when converted to transactions, the transactions are large. Perhaps parallelizing the inner loops is a better practice for transactional systems, because it limits the buffering requirements.

Figure 3.10 shows a much lower write-set to computation ratio than that of the previous section. This is because the critical transactions in the previous section are small, coming from programmer-identified and -tuned critical sections, so their ratios are higher. On the other hand, in this section, we map large parallel regions

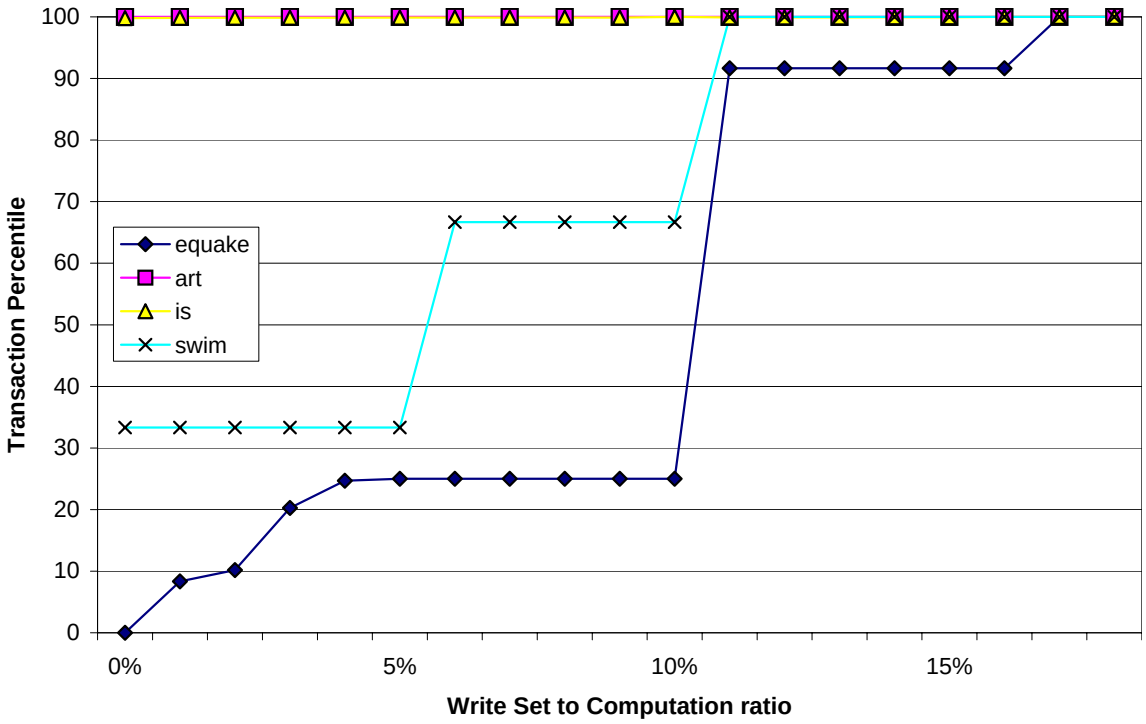


Figure 3.10: The write-set-to-computation ratio of speculatively-parallelized programs.

to critical transactions, so they have lower ratios. Finally, none of the four programs had noticeable nesting, as no nested parallelism is exploited in these applications.

Conclusion: When you use transactions for speculative parallelization, even though an applications' data-set may no longer fit in an L1-sized transactional buffer, they still would fit in an L2-sized transactional buffer. Because of this increase in transaction size, perhaps new ways of defining transactions would better suit transactional systems. Moreover, since the write-set to computation ratio decreases noticeably, extra bandwidth resources will not be needed—the bandwidth management techniques needed for general transactional execution (discussed in Section 3.4.3) will suffice for speculative parallelism.

3.6 Related Work

There are been efforts to develop transactional programs from the scratch. STAMP is a comprehensive benchmark suite for evaluating TM systems [88]. It includes seven applications and twenty variants of input parameters and data sets in order to cover a wide range of transactional execution cases. Compared to our results with existing multithreaded programs, they report longer transaction lengths and larger read-/write-set sizes on the average. This is mainly due to the fact that the benchmark is designed to stress TM systems with active usage of transactions for system evaluation. However, the numbers from both results are not order-different. This confirms our assumption that the inherent parallelism does not change much regardless of the programming primitives used to express the parallelism.

In [11], the usage of I/O and system calls within critical sections are analyzed in two large applications. The analysis shows the implication of their usage for TM that the large majority of syscalls performed within critical sections can be handled with a range of existing techniques in a way transparent to the application developer. This coincides with our suggestion for handling I/O in transaction boundaries.

3.7 Conclusion

In this chapter, we analyze the transactional behavior of lock-based multithreaded programs by extracting the transactions from parallel regions and critical sections already marked in the original programs. We find most transactions are small, but significant execution time is spent within larger transactions. Write-set to computation ratio shows that commit bandwidth will be an issue and clever techniques might be required. We also conclude that only limited nesting support would be required. Additionally, using speculative parallelism increases buffering requirements, but they remain reasonable and commit bandwidth is not an issue. Finally, we make two conclusions about software transactional memory systems: with the high write-set to computation ratios observed, such systems will have significant overhead and similarly, small transaction sizes will expose the high overhead of transaction creation

and destruction. Taking advantage of the observations made in this chapter (i.e., short-lived transactions, small read-/write-set sizes, and shallow nesting in the common case), we present a practical solutions for TM virtualization that handles the uncommon case in a cost-effective manner in the next chapter.

Chapter 4

Virtualizing Transactional Memory

4.1 Introduction

While hardware acceleration for TM systems is crucial for performance, the hardware resources are limited. For transactional memory (TM) to achieve widespread acceptance, it is important to deal practically with the uncommon case of applications with transactions that exceed the capabilities of hardware resources. Transactions must not be limited to the physical resources of any specific hardware implementation.

There are challenging uncommon cases for TM systems. Hardware TMs typically use the cache as buffer for transactional data and metadata [60, 90]. What if there is a long transaction that overflows the buffer capacity? Multiple pairs of read and write metadata bits are used to support nested transactions. What if transactions are nested deeper than what the hardware can support? TM systems should interact with the operating system. How can we allow a transaction to keep executing in the presence of various system events such as interrupt handling, context switch, paging, and thread migration?

A practical TM system should guarantee correct execution even when transactions exceed scheduling quanta, overflow the capacity of hardware caches or physical memory, or include more independent nesting levels than what is supported in hardware. Existing proposals for TM virtualization are either incomplete or rely on complex and inflexible hardware-based implementations, which is often an overkill for the common

case behavior where virtualization is invoked infrequently.

This chapter presents *eXtended Transactional Memory (XTM)*, a software-base TM virtualization system that virtualizes all aspects of transactional execution (time, space, and nesting depth). It is implemented in software using virtual memory support in the operating system. XTM operates at page granularity, using private copies of overflowed pages to buffer transactional state until the transaction commits and snapshots of pages to detect interference between transactions. We also describe two enhancements, XTM-g and XTM-e, to XTM that use some hardware support to address key performance bottlenecks.

This chapter is organized as follows. Section 4.2 discusses the limitation of hardware TM systems. Section 4.3 reviews the requirements and design space for TM virtualization. Section 4.4 describes the base XTM design, while Section 4.5 presents the two enhanced XTM systems. Sections 4.6 and 4.7 present qualitative and quantitative comparisons between XTM and hardware virtualization schemes. Finally, Section 4.8 presents related work and Section 4.9 concludes the chapter.

4.2 Limitation of Hardware TM Systems

There are several proposals for TM systems that use hardware resources for data versioning and coherence protocols for conflict detection [119, 60, 6, 90]. While the hardware is essential for accelerating transactional processing, for TM to become useful to programmers and achieve widespread acceptance, it is important that transactions are not limited to the physical resources of any specific hardware implementation.

There are three types of limitations in TM systems using the hardware resources.

- **TM Space**: Typically, the cache is used as buffer for transactional data and metadata. There are the cases where transactional data escape the cache. Long-lived transactions can overflow the capacitance of the cache. Virtual memory support requires paging that needs to move transactional data from cache to disk. In multi-core and multi-processor systems, the operating system may migrate threads from a core/processor to another.

- **TM Time**: There are the cases where transactions have to relinquish the resouces. Operating system may preempt threads for fair sharing of system resources. Long-lived transactions may last longer than the OS time quanta and get context-switched. The OS may also preempt a user threads in order to deal with external events such as interrupts.

- **TM Depth**: There are the cases where TM systems need additional resources to manage transactions inside another transactions. Programs are built on libraries and the composition of multiple software modules can generate nested transactions.

TM systems should guarantee correct execution even when transactions exceed scheduling quanta, overflow the capacity of hardware caches or physical memory, or include more independent nesting levels than what the hardware supports. In other words, TM systems should transparently virtualize *time*, *space*, and *nesting depth*. While recent application studies have shown that the majority of transactions will be short-lived and will execute quickly with reasonable hardware resources [6, 32], the infrequent long-lived transactions with large data sets must also be handled correctly and transparently.

Existing TM proposals are incomplete with respect to virtualization. None of the proposals supports nesting depth virtualization, and most do not allow context switches or virtual memory paging inside a transaction (TCC [60], LTM [6], LogTM [90]). UTM [6] and VTM [120] provide time and space virtualization but require complex hardware to manage overflow data structures in memory and to facilitate safe sharing among multiple processors. However, since long-lived transactions are not the common case [32], perhaps such a complex and inflexible approach is not optimal. There are proposals to switch to software TM systems at the demand on TM virtualization [41]. They require two versions of application code: one for hardware TM system and the other for software TM system. They also do not provide strong isolation.

4.3 Design Considerations for TM Virtualization

While the various TM architectures differ in the way they operate, their hardware structure is similar. They all track the transaction read-set and write-set in the private caches (L1 and potentially L2) of the processor executing the transaction [119, 60, 6, 90]. Membership in either set is indicated using additional state bits (metadata) associated with each cache line. The data for the write-set are also stored in the caches. Conflicts between concurrently executing transactions are detected during coherence actions for cache lines that belong to the write-set of one transaction and the read-set of another. More recent proposals support nested transactions that can rollback independently [84]. Tracking the read-set and write-set for nested transactions requires an additional tag per cache line to identify the nesting depth.

TM virtualization allows transactions to survive cache overflows, virtual memory paging, context switches, thread migration, and extended nesting depths. Virtualization is achieved by placing transactional state (read-sets and write-sets) in virtual memory, which provides processor-independent and practically infinite storage. Depending on the case, we may place some of the transactional state in virtual memory (e.g., on a cache overflow) or all of it (e.g., on a context switch).

A good virtualization scheme should satisfy the following requirements with respect to correctness and performance. First, it should be completely *transparent to the user*. Second, it should *preserve transactional atomicity and isolation* under all circumstances. Third, it should *not affect the performance of the common case* when virtualization is not needed. Fourth, it should maintain strong isolation between transactional and non-transactional codes. Finally, virtualized transactions should have *no significant effect on the performance of non-virtualized transactions* executing concurrently.

While the data for virtualized transactions always go through virtual memory, there are several design options to consider for the mechanisms that implement data versioning, conflict detection, and commit for virtualized transactions[1]. Table 4.1 summarizes the advantages of the major alternatives for each mechanism. The basic

[1]There are similar design options for hardware support for TM. However, the thesis focuses exclusively on the design tradeoffs in TM virtualization.

		Data Versioning	Conflict Detection	Commit
Implementation	**HW**	Low per access overhead	Overlap with other work	Low overhead
	SW	No ISA/HW changes needed	Flexibility in conflict resolution	Supports transactional I/O
Granularity	**Cache Line**	Low memory & BW overhead	Less false sharing	Low overhead
	Page	Reuse paging mechanisms	No ISA/HW changes needed	Amortize overheads better
Timing	**Eager**	Fast commits	Early detection	N/A
	Lazy	Fast aborts	Deadlock-free	N/A

Table 4.1: TM virtualization options for data versioning, conflict detection, and transaction commit. Each cell summarizes the advantage of the corresponding implementation, granularity, or timing option.

choices are between a) hardware vs. software implementation (performance vs. cost and flexibility), b) cache line vs. page granularity (storage efficiency and performance vs. complexity), and c) eager vs. lazy operations (performance vs. isolation). While it is difficult to quantitatively evaluate all reasonable combinations of the above options, this thesis aims at characterizing the design space for TM virtualization sufficiently so that major conclusions can be drawn.

If performance was the only optimization metric, it is obvious that a virtualization system should be hardware-based, should handle data at cache line granularity, and should perform all operations eagerly. However, a virtualization system is by nature a backup mechanism, only invoked when the hardware mechanisms are no longer sufficient. Recent studies show that the majority of transactions will not exceed the hardware capabilities [6, 32]. In Chapter 3, we showed that, when transactions are used for non-blocking synchronization, 98% of them require less than 22 Kbytes for read-set and write-set buffering. About 95% of transactions include less than 5,000 instructions and are unlikely to be interrupted by external events (context switches, interrupts, paging, etc.). When transactions are used for speculative parallelization, they showed that read- and write-sets get significantly larger, but that the capacity of an L2 cache (e.g., 128 Kbytes) is rarely exceeded. The rare occurrence of

transactions requiring virtualization implies that one's choices in architecting a virtualization system should better balance performance and cost. We propose such systems in Sections 4.4 and 4.5.

4.4 eXtendend Transactional Memory (XTM)

The XTM system provides space, time, and nesting depth virtualization while meeting all the requirements introduced in Section 4.3. XTM is software-based and operates at the OS level. The only hardware requirement for XTM is that an exception is generated when a transaction overflows hardware caches or exceeds the hardware-supported nesting depth. XTM handles transaction read-sets and write-sets at page granularity. It uses lazy versioning and optimistic conflict detection.

4.4.1 XTM Overview

With XTM, a transaction has two execution modes: all in hardware (no virtualization) or all in software (virtualized). When the hardware caches are filled, XTM catches the overflow exception and switches to virtualized mode, where it uses private pages from virtual memory as the exclusive buffer for read- and write-set. Switching first aborts the transaction in hardware mode, which clears all transactional data from hardware caches, and then restarts it in virtualized mode. While aborting introduces re-execution overhead, it eliminates the need for an expensive hardware mechanism to transfer the physically-addressed transactional data in caches to virtual memory. XTM also clears the data TLB for the processor executing the overflowed transaction. No other transactions are affected by the switch.

In virtualized mode, XTM catches the first access to each page through a page-fault and creates on-demand copies of the original page in virtual memory. By operating on copies, the virtualized transaction is isolated from any other transactional or non-transactional code. We create two copies of the original page: the *private copy* is created on first access (load or store) by copying the private page, and the *snapshot page* is created just before the first store. Essentially, the snapshot is a pristine copy

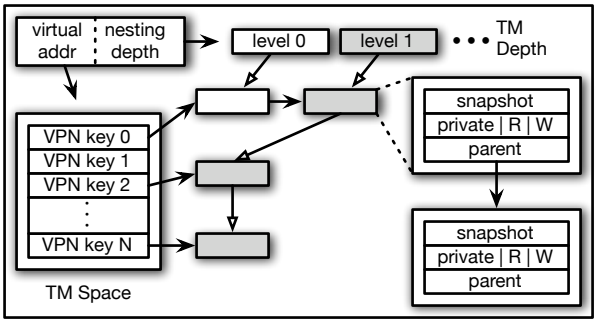


Figure 4.1: The Virtualization Information Table (VIT). The white box belongs to level 0, and the gray boxes belong to level 1.

of the original page in memory at the time the transaction started accessing it, and it is used for rolling back and conflict detection. If a page is never written, we avoid creating the snapshot as the private page is sufficient.

Once the necessary copies are created by XTM, the transaction can access data directly through loads/stores, without the need for XTM to intervene. The transaction makes all writes to its private page (lazy versioning). When creating the copies, XTM uses non-cached accesses to avoid thrashing caches. A virtualized transaction checks for conflicts when it is ready to commit (lazy conflict detection). Detection is performed by comparing each snapshot page to the original page in virtual memory similar to backward-oriented validation developed in database literature [59]. If the contents of the two pages differ, a conflict is signaled and the virtualized transaction is rolled back by discarding all private pages. If all snapshots are validated, we commit the transaction by copying its private pages to their original locations.

XTM uses two private data structures to track transactional state. First, a per-transaction page table provides access to the private copies of pages in the read-set or write-set of the transaction. Assuming a hierarchical organization, this page-table is not fully populated. Instead, it is allocated on demand and consists only of the translation entries necessary for TM virtualization. For every virtual page, the page table points to the physical location of the private copy. The second structure is the *Virtualization Information Table (VIT)*, which is shown in Figure 4.1. The VIT is organized as a hash table and contains one entry per page accessed at each nesting

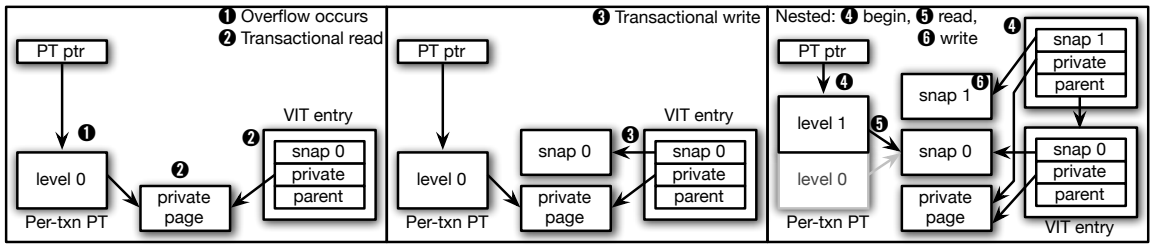


Figure 4.2: Example of space and nesting depth virtualization with XTM. ❶ When an overflow first occurs, a per-transaction page table (PT) and a VIT are allocated. ❷ On the first transactional read, a private page is allocated, and ❸ a snapshot is created on the first transactional write. ❹ When a nested transaction begins, a new PT and VIT entry are created. ❺ A nested read uses the same private page, and a ❻ nested write creates a new snapshot for rolling back the nested transaction.

level. An entry holds pointers to the private and snapshot pages, metadata indicating if it belongs in the read-set or write-set, and the pointers necessary for the hash table and to link related entries (see Section 4.4.2). The VIT is queried using a virtual address and a nesting depth. It can also be traversed to identify all private copies for a transaction at a specific depth.

4.4.2 XTM Space and Depth Virtualization

Figure 4.2 presents an example of space and depth virtualization using XTM. After an overflowing transaction is aborted, XTM allocates a per-transaction page-table and a VIT, both initially empty (❶). When the transaction restarts in virtualization mode and attempts to read its first page, XTM creates a private page copy and a VIT entry. The newly allocated private page is pointed to by both the VIT and the page table, and the R bit is also set in the VIT entry (❷). On the first transactional write to the page, a snapshot page is created, the VIT entry is updated, and the W bit is set (❸). If the first transactional access had been a write instead of a read, XTM would have executed steps (❷) and (❸) together.

When a nested transaction begins in virtualized mode, we need to independently track its read- and write-set. Hence, we allocate a new per-transaction page table independent from that of its parent transaction (❹). With the new table, XTM

catches the first read/write to a page by the nested transaction without walking the parent's table to change access permissions. The new page table is also only partially populated. On the other hand, we do not allocate a new VIT. Nested reads (❺) and nested writes (❻) are handled like those of the parent transaction. The first nested read creates a new VIT entry that points to the parent's private page. If this is the first time this page is accessed at any depth, we create a new private page. On the first nested write, a new snapshot of the private page is created, and the modification goes to the private page. If multiple transactions in the nest access the same page, we have multiple linked VIT entries and multiple snapshots, but the private page is shared among the parent and all its nested children.

When the nested transaction needs to commit, it validates its read-set. The read-set is all snapshot pages and all read-only private pages. Validation involves comparing all pages in the read-set to the current state of the pages in memory. If validation is successful (no differences), the transaction commits by merging its transactional metadata with that of its parent. Finally, the per-transaction page table for the nested transaction is discarded. If validation fails, the transaction is rolled back by discarding all VIT entries at that level and its page table. Modified private pages are rollback back with snapshots. When the outermost transaction (nesting depth 0) commits, we copy all private pages to the original locations and merge the private page table's metadata bits into the master page table.

To make the outermost commit atomic, a transaction must gain exclusive access of all its virtualized pages. In some TM architectures, this involves TLB shootdowns which can be expensive. In the TCC system [60], it is sufficient for the virtualized transaction to retain the commit token; however, this serializes commits. One can devise an adaptive protocol that selects between the two options, if available, based on the number of pages committed and the expected impact of serialization. The writes used to copy private pages into original locations must be snooped by hardware to check conflicts for pending hardware-mode transactions.

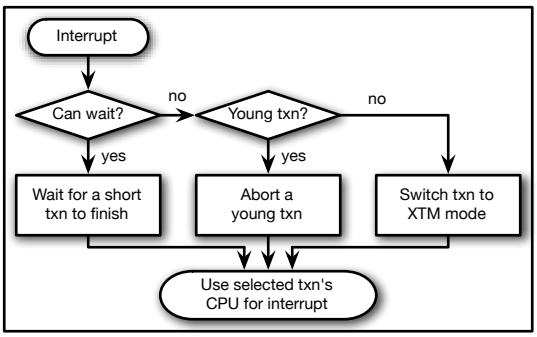


Figure 4.3: The process for handling interrupts in XTM.

4.4.3 XTM Time Virtualization

XTM also handles events that require process swapping (e.g., context switches or process migration). Once a transaction is in virtualization mode, all its state is in virtual memory and can be paged in and out on demand [2].

Other events, like I/O interrupts, require interrupt handling, before resuming user code. Existing TM virtualization schemes [6, 120] propose swapping out transactional state on such interrupts. Since most transactions are short [32], XTM uses an alternate approach, shown in Figure 4.3, that avoids swapping overhead in most cases. On an interrupt, we wait for one of the processors to finish its current transaction and then assign it to interrupt processing. Since most transactions are short, this will probably happen quickly. If the interrupt is real-time or becomes critical, we abort the youngest transaction and use its processor for interrupt handling. When we restart the transaction, we use the hardware mode. If a transaction is restarted many times due to interrupts, we restart it virtualized so further interrupts will not cause aborts. The latter case only happens if all transactions in the system are long.

4.4.4 Discussion

XTM can be implemented either in the OS as part of the virtual memory manager or between underlying TM systems and the OS, like virtual machines [151]. Its only

[2]Long virtualized transactions are more likely to abort due to interference. The same problem can occur with long transactions in hardware mode. To avoid livelock one must introduce some aging mechanism as in [60].

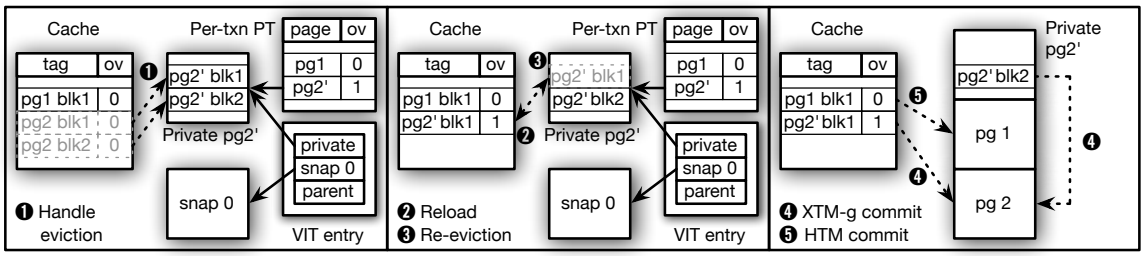


Figure 4.4: Example of space virtualization with XTM-g. This example starts with a transaction with three transactionally-modified blocks. ❶ When a line is evicted because of overflows, the transaction is not aborted. A private page is allocated, a snapshot is made (since the line is modified), the OV bit is set in the PT, and the line is copied. ❷ When the evicted line is reloaded in the cache, the line's OV bit is set. ❸ The line is re-evicted to the private page without an eviction exception. ❹ XTM-g commits first, then ❺ the hardware TM (HTM) commits.

hardware requirements are exceptions for virtualization events (e.g., cache overflow, exceeding nesting depth, etc.). XTM implements per-transaction page tables, which can be cheaply realized by copying and modifying only the portion of the master page table containing the addresses accessed by the overflowed transaction. For example, XTM in x86 starts with a 4KB empty page directory and augments it with second-level 4KB page tables as needed. Since XTM is software-only, all its policies and algorithms can be tuned or changed without affecting the processor ISA. In addition, we do not need to escape from the context of an overflowed transaction in order to perform XTM-related operations on an overflow exception. Non-transactional loads and stores [84] are sufficient for the operations.

4.5 Hardware Acceleration for XTM

We now introduce XTM-g and XTM-e to improve performance by using a few key hardware features. XTM-g and XTM-e are essentially hybrid hardware/software schemes for TM virtualization.

4.5.1 XTM-g

On a cache overflow in the base XTM design, we abort the transaction and switch to virtualized mode; this incurs large penalties. In XTM-g, a transaction overflows to virtual memory *gradually*, without an abort. Hardware handles the data previously accessed, and any new data are tracked in the virtual memory system. XTM-g is especially beneficial when the hardware caches are almost sufficient to to hold all transactional state.

When virtualized, a transaction buffers state in both hardware and virtual memory. To distinguish the two, we introduce the overflow or *OV* bit to page table entries, TLB entries, and cache lines. If OV is set, the corresponding data have been virtualized. Upon eviction, data that do not belong in the read-set or write-set of any transaction are evicted as usual. If a line accessed by a transaction is evicted, XTM-g copies the line to a private page and marks its OV bit in the page table. It is possible for virtualized lines to re-enter the hardware caches, in which case the OV bit is set in the cache. Lines with the OV bit set can simply be evicted from the cache, since they are already virtualized.

Figure 4.4 illustrates XTM-g. In this example, the hardware cache has three lines written by a transaction, where two of them belong to the same page. When one of the two lines is evicted because of an overflow, an exception is raised and the XTM-g software starts. It first uses reverse translation to find the virtual address for the overflowed line. Then it allocates a private page, creates a snapshot, updates the VIT, and writes the evicted data to the private page. Before returning, XTM-g queries the cache about other lines from the same virtual page and finds the other line. It is also evicted to the private page and their metadata are placed in the VIT. Finally, the OV bit of the page is set in the page table so cache lines that re-enter the cache will have their OV bit set. By the end of this process, the transaction has one page in virtual memory while the rest is still in the hardware cache (❶). If other overflows occur, more pages can be moved to virtual memory as needed. Once evicted, the cache lines are reloaded with the private page address and their OV bit set in the cache (❷). When they are re-evicted, cache eviction logic checks their OV bits. Since the bits are set, they are allowed to be evicted to the private page without an eviction

exception (❸). To properly commit such a transaction, the hardware TM system must support a two-phase commit protocol [84]. Once validation of hardware-tracked data is complete, control is transferred to XTM-g to validate the pages in virtual memory. If that validation passes, we first commit the data in virtual memory (❹) and then return to the hardware system to commit the cache data (❺). Essentially, XTM-g runs as a commit handler [84].

4.5.2 XTM-e

XTM and XTM-g operate at page granularity and are prone to aborts due to false sharing: a transaction may abort because it read a word in the same page as a word committed by another transaction. XTM-e allows cache line-level tracking of read- and write-sets, even for overflowed data. First, each VIT entry is made longer to have one set of R and W bits per cache line in the page. XTM-e evicts cache lines to virtual memory similar to XTM-g, but also sets the fine-grain R/W bits in the VIT using the metadata found in the cache. Second, XTM-e must handle the case where a cache line in a virtualized page is accessed later.

A naïve wait solution would be to switch to software on every access to a page with the OV bit set in the TLB. To avoid this unnecessary overhead, XTM-e uses an *eviction log buffer (ELB)*. The ELB is a small hardware cache, addressed by cache line address, that stores a tag and the R and W bits for a cache line. When a line from an OV page is accessed, we note the R or W metadata in the ELB without interrupting the user software. When the ELB overflows due to associativity or capacity issues, we merge the metadata in all valid ELB entries into the proper VIT entries. In other words, the ELB allows us to amortize the cost of an exception over multiple cache lines. If the ELB does not fill until the transaction completes, we transfer metadata from the ELB to the VIT before the validation step. During validation, XTM-e uses the fine-grain R/W bits available to determine which cache lines within a snapshot or private page should be compared with the original page in memory to check for conflicts. Overall, XTM-e improves on XTM-g by eliminating most of the false sharing, which is common if pages are sparsely accessed by transactions.

	XTM	XTM-g	XTM-e	VTM
Virtual-ization	space, time, nesting depth			space, time
HW Cost	OV exception	OV exception, OV bit	OV exception, OV bit, ELB	XADT walker, XADT cache, virtual tags in caches
SW Cost	VIT, page tables, extra copies per accessed page			XADT, XSW, overflow count, XF
Switch Overhead	transaction abort transaction abort	OS handler		HW handler
Other Overheads	page copying, page comparisons		page copying, cache line comparisons	accessing XADT/XF, XF/count consistency
Sensitivity	page occupancy, false-sharing		page occupancy	XF miss ratio
Flexibility	pure SW	mostly SW		mostly HW

Table 4.2: A qualitative comparison of XTM, XTM-g, XTM-e, and VTM.

4.6 Qualitative Comparison

This section presents a qualitative comparison between XTM systems and VTM system [120], a performance-oriented hardware virtualization solution. The basic mechanism of VTM is explained first. Then various aspects of XTM, XTM-g, and XTM-e are compared with those of VTM.

4.6.1 VTM

VTM uses mostly hardware mechanisms to provide virtualization at cache line granularity with eager conflict detection and lazy versioning [120]. It supports space and time virtualization, but does not virtualize nesting depth. For each process, VTM defines the XADT data structure in virtual memory. The XADT is organized as a hash table and contains an overflow count, the overflowed data (including metadata), and a bloom filter (called the XF) that describes which addresses have overflowed to the XADT. When a hardware cache overflows, VTM evicts a cache line into the XADT

and appropriately updates the overflow count and the filter. On a context switch, VTM evicts the entire read- and write-set for the transaction to the XADT. Conflict detection and refills for evicted data occur on demand when transactions experience cache misses. However, the XADT is only searched if the overflow count is non-zero and the XF filter returns a hit. Commits or aborts for data in the XADT happen lazily: VTM atomically sets the status of transactions to committed or aborted and does the transfer to memory or XADT clean up at a later point.

VTM provides fast execution when virtualization is not needed by caching the overflow count and XF in an additional hardware cache. It also provides for fast execution when virtualizing, as it uses hardware to evict cache lines to the XADT and search the XADT for conflicts or refills. Nevertheless, the performance advantages of VTM comes at a significant complexity cost. First, the hardware must be aware of the organization of the XADT, so the XADT must be defined in the instruction set, similar to how the page table organization is defined in ISAs if hardware TLB refills are desired. Second, in order to allow evictions to the XADT without trapping into the OS for reverse translation, VTM must append each cache line with its virtual page number. For 32-byte cache lines, this implies a 10% area increase for data caches. Third, the hardware must provide coherence for the cached copies of the overflow count and the XF in each processor. Also, these cached copies must be consistent with updates to the XADT. For example, a processor incurring a miss must receive the answer to its coherence miss requests before checking the overflow counter. Otherwise, one can construct cases where a conflict is missed due to a race between an XADT eviction and a counter update. Overall, updating the counter and the XF must be done carefully and, in several cases, accesses to these structures should act like memory barriers or acquire/release instructions for relaxed consistency models.

4.6.2 Comparision

Table 4.2 summarizes the key differences. Note that VTM does not provide virtualization of nesting depth.

Hardware Cost and Memory Usage: The only HW requirement for XTM is an exception when virtualization is needed. XTM-g requires OV bits in page-tables and caches, while XTM-e adds the ELB as well. On the other hand, VTM require significant hardware components and complexity: a cache for its overflow structure (XADT), hardware walkers for XADT, and hardware to enforce coherence and consistency for the overflow counter and the filter. Unfortunately, the complexity of VTM goes beyond microarchitecture. The XADT organization and any software visible issues about the way it is cached (e.g., consistency) must become part of the ISA.

On the other hand, XTM can lead to memory usage issues as it requires storage for the per-transaction page-table, the VIT and the private/snapshot copies. Even though the page-tables are not fully populated, the XTM space requirements will be higher than that for VTM, particularly if transactions overflow hardware caches by a few cache lines. VTM uses memory space only for XADT, which is a hash-table for evicted cache lines.

Implementation Flexibility: XTM is implemented purely in software. XTM-g and XTM-e have small and simple hardware requirements. Since most of these three systems is in software, there is significant flexibility in tuning their policies and integrating them with the operating system. On the other hand, VTM is mostly in hardware, which means that there is no flexibility in data-structure organization, underlying coherence protocols for XADT caching, etc. Nevertheless, the hardware implementation of VTM allows for better performance isolation between virtualized transactions and non-virtualized transactions. With XTM, making the software commit atomic can cause stalls to other transactions from the same process. Nevertheless, processors executing transactions from other processes are never affected, which is particularly important.

Performance: So far we have argued that XTM and its enhancements provide lower hardware cost and better flexibility than VTM. Hence the question becomes how they compare in performance. If the software-based XTMs can also provide competitive performance, then they have an edge over the hardware-based VTM.

The base XTM system can introduce significant overheads. When XTM virtualizes a transaction, it starts with an abort. The necessary re-execution can be expensive for long transactions. Of course, these overheads are important if virtualization events are often. If this is the case, XTM-g (eliminates aborts for switch) and XTM-e (reduces aborts due to false sharing) will be necessary for the XTM approach to be competitive. XTM also benefits from applications that access most of the data in each page they touch, as this makes page copying operations for versioning or commit less wasteful.

On the other hand, VTM's overhead comes mostly from accessing the XADT when the XF filter misses. Hence, the case when VTM can be slow (despite all the hardware support) is when many transactions overflow and searching, inserting, and deleting in a large XADT becomes too slow. Note that manipulating the VIT is faster as each entry is for a whole page, not a cache line. Furthermore, the VIT is private to each transaction, while the XADT is shared within a process; hence, some synchronization is needed for the XADT across processors. In all other cases, VTM provides fast virtualization as it avoids switching to OS handlers and operates on data at fine granularity. Again, this is particularly important if virtualization events are often.

For time virtualization, XTM has a better process that avoids swapping transactions in many cases. On the other hand, VTM always swaps out transactional state, even to run a short interrupt handler. Hence, VTM can be inefficient for handling frequent interrupts.

4.7 Quantitative Comparison

We compared XTM to VTM using an execution-driven simulator. To our knowledge, this is the first quantitative analysis of TM virtualization. Table 4.3 shows the simulation parameters for our experiments. The simulator models a CMP with 16 single-issue PowerPC processors. For an underlying hardware TM system, we used TCC because it allows transactions to be used for both non-blocking synchronization and thread-level speculation [60]. The latter case leads to larger transactions likely to

Feature	Description
CPU	16 PowerPC cores, 200 cycle exception handling overhead
Cache	Private, 4-way, 32KB, with 32B lines
Victim Cache	16 entries
Main memory	4KB page, 100 cycle transfer latency
Bus	16B wide, 3 cycle arbitration, 3 cycle transfer latency

Table 4.3: Parameters for the simulated CMP architecture.

stress hardware resources. Each processor can track transactional state in its 32KB, 4-way associative L1 data cache (32B lines). A 16-entry victim cache is used to eliminate most of the conflict misses [83]. The simulator captures the memory hierarchy timing, including all contention and queuing. We implemented VTM as a hardware extension to the simulator (XADT walker, coherence for XF and overflow count, etc.). We implemented XTM by running OS-level code on top of the simulator when a virtualization exception is triggered. For XTM-g and XTM-e, we implemented the OV bit and the ELB, and also provided software with instructions to take advantage of them. Our experiments focus on virtualization events when a single application runs on the CMP—we did not conduct multiprogramming experiments.

We used three parallel benchmarks from SPLASH2 (radix, volrend, and water-spatial) and one from SPLASH (mp3d), which used transactions to replace locks for non-blocking synchronization (so transactions from these programs will likely be small). We also used two SPEC benchmarks (equake and tomcatv), which use transactions for speculative parallelization at the outer-most loop level (likely to produce many long transactions). To explore other interesting patterns not generated by the six applications, we also designed a microbenchmark to produce randomized accesses with a desired average transaction length, size of read-/write-sets, and nesting depth.

4.7.1 Space Virtualization

Figure 4.5 presents the comparison between XTM and VTM for space virtualization (i.e., overflow of hardware caches). We have omitted mp3d, water-spatial, and equake

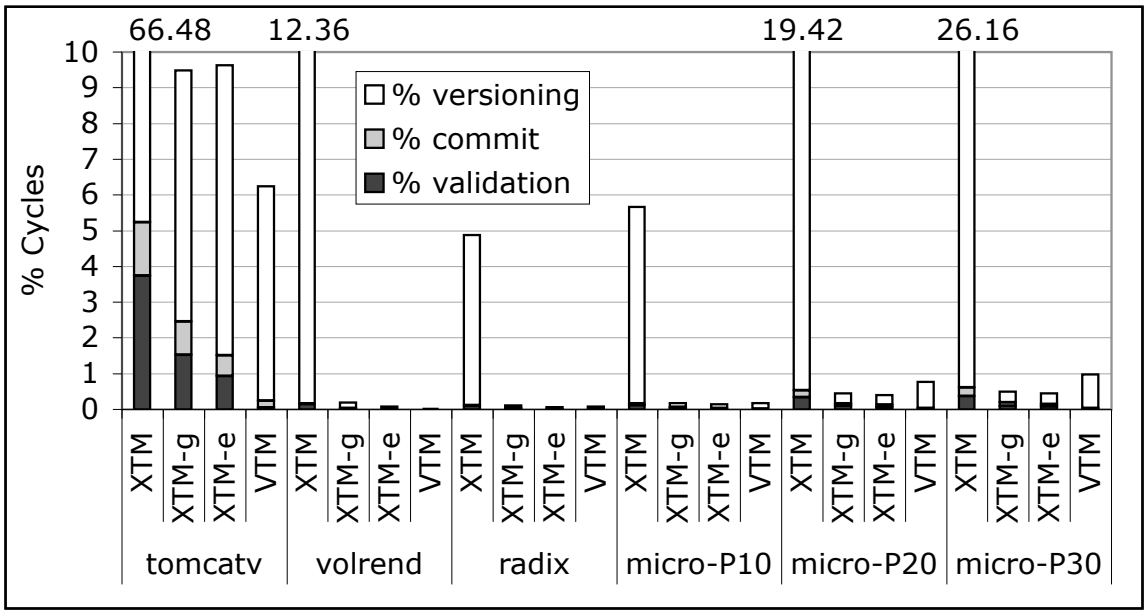


Figure 4.5: Comparison of overhead for space virtualization.

because they never overflow with the 32KB cache (0% overhead in those cases). The microbenchmark was configured with three average read-/write-set sizes, where -Pn means accessing a uniformly random number of pages between 1 and n, inclusive. Figure 4.5 shows the overhead introduced by each design as the percentage of total cycles for the program execution. The overhead is broken down into time spent for data versioning, committing, and validation (conflict detection). For radix and micro-P10, the base XTM works well and introduces overhead of less than 5%. For the rest of the programs, XTM introduces significant overhead due to the transaction aborts when switching to virtualized mode and due to the time necessary to make the private and snapshot copies. However, XTM-g and XTM-e reduce the overhead of XTM significantly (less than 0.5% for several cases) and make it comparable to VTM.

The overhead breakdown of radix and volrend is shown enlarged in Figure 4.6. For volrend, VTM performs better, while for radix, XTM-e is the fastest. The reason is the time spent searching for overflowed data. VTM's data versioning cycles come from time spent overflowing data to the XADT and then accessing it again later. On

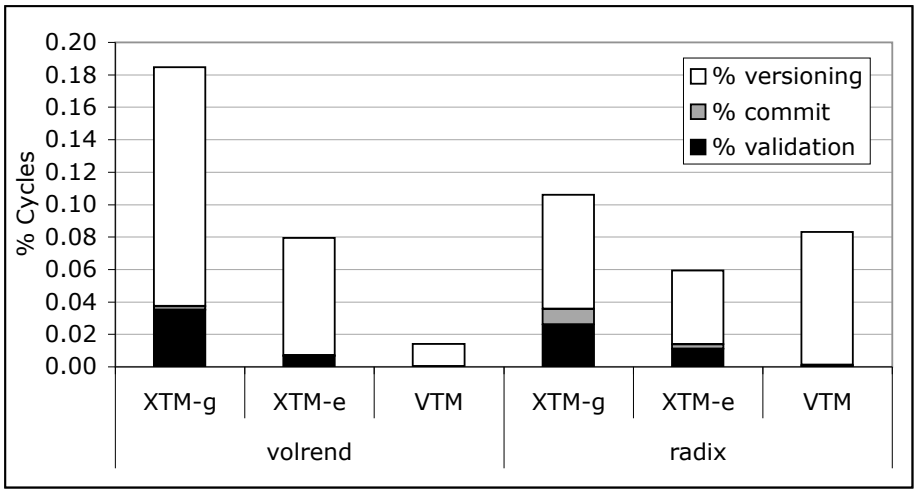


Figure 4.6: Comparison of overheads between XTM-g, XTM-e, and VTM.

the other hand, the XTMs' data versioning cycles come from changing the address mapping of overflowed pages to point directly to the corresponding private pages. For programs that repeatedly access overflowed data, search time is more significant than overflow time.

In tomcatv, all virtualization schemes have relatively large overheads. Other programs contain only a few transactions that overflow, but in tomcatv, a larger number of transactions invoke virtualization. Even the hardware-heavy VTM cannot reduce the overhead of virtualization any further than 6%. This is due to many transactions overflowing concurrently, which increases the size of the shared XADT and adds to the search time. Even though they are mostly software, XTM-e and XTM-g perform reasonably well at 9%.

So far we have assumed that hardware can buffer transactional state only in the 32KB, L1 data cache. However, one can also place transactional state in the L2 cache, reducing the frequency of overflow. Figure 4.7 shows the impact of HW buffer sizes on the comparison between XTM and VTM. If 64KB are available, tomcatv, volrend, and micro-P10 do not generate any overflows (0% overhead). For the other benchmarks, larger HW capacity means less-frequent overflow, hence the overhead of virtualization drops. At 128KB, no real application has any overflows and only micro-P30 requires 256KB before it shows the same behavior. Overall, the conclusion

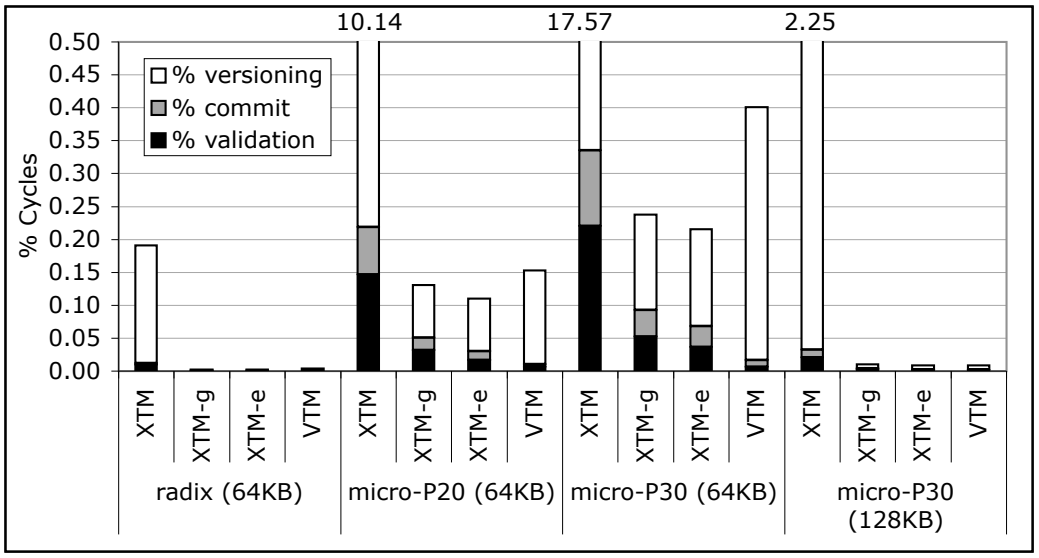


Figure 4.7: Influence of HW capacity of transactional state buffers on the overhead of TM virtualization.

is that if the L2 cache is used to buffer transactional state, even applications that use TM for speculative parallelization of outer loops will rarely overflow hardware resources. Hence, the small overhead increase of a software system like XTM is no longer significant.

4.7.2 Memory Usage

Table 4.4 measures the memory requirements of the virtualization schemes. For XTM, we measure the maximum number of VIT entries and extra pages needed for copies (in parenthesis) per transaction. For VTM, we count the maximum number of XADT entries per transaction, and we compare the number of VIT and XADT entries, which affect the searching latency for both data structures. Since the XTM has a VIT entry per page, the number of VIT entries is much smaller than the number of VTM's XADT entries. No benchmark uses more than 39 VIT entries, while some benchmarks use up to 17,000 XADT entries. The bulk of the memory overhead for XTM comes from the private and snapshot page copies. However, no benchmark uses more than 700 copies for base XTM (2.8MB) or 386 pages for XTM-e and XTM-g (1.5MB). For

Benchmark	XTM	XTM-g	XTM-e	VTM
Tomcatv	21 (439)	15 (257)	15 (254)	4025
Volrend	33 (316)	19 (136)	19 (139)	238
Radix	21 (124)	7 (23)	8 (27)	779
Micro-P10	19 (350)	9 (86)	9 (90)	1396
Micro-P20	29 (495)	18 (207)	18 (202)	8039
Micro-P30	39 (700)	28 (386)	28 (380)	17319

Table 4.4: Memory pressure. This table shows the maximum number of XADT entries for VTM and the maximum number of VIT entries XTM. The maximum number of extra pages used by XTM is enclosed in parenthesis.

VTM, the XADT must store 32B cache lines and metadata. Hence, for a maximum of 17,300 entries, the XADT will occupy about 650KB. In summary, despite using page granularity, XTM does not have unreasonable memory usage requirements for any modern system.

4.7.3 Time Virtualization

To compare XTM to VTM with time virtualization, we simulated the arrival of I/O interrupts every 100,000 cycles. On an interrupt, we need to find a processor to execute its handler. We set the handler size to zero cycles, so all the overhead is due to switching in and out of the handler. VTM suggests that when an interrupt arrives, a transaction is swapped out to virtual memory to provide a processor for the handler. For XTM, we evaluated two policies. One is the VTM policy (abort transaction, restart later in virtualized mode). The other is the three-stage interrupt handling process explained in Section 4.4.3 that avoids virtualization unless necessary to guarantee forward progress.

Figure 4.8 shows the overhead introduced by the virtualization scheme as interrupts occur within a program. The XTM, XTM-g, and XTM-e bars assume the VTM swap-based policy. The XTM+, XTM-g+, and XTM-e+ bars assume the proposed approach that first attempts to abort and retry (in hardware mode) a young transaction. The absolute percentage of overhead is not particularly interesting as

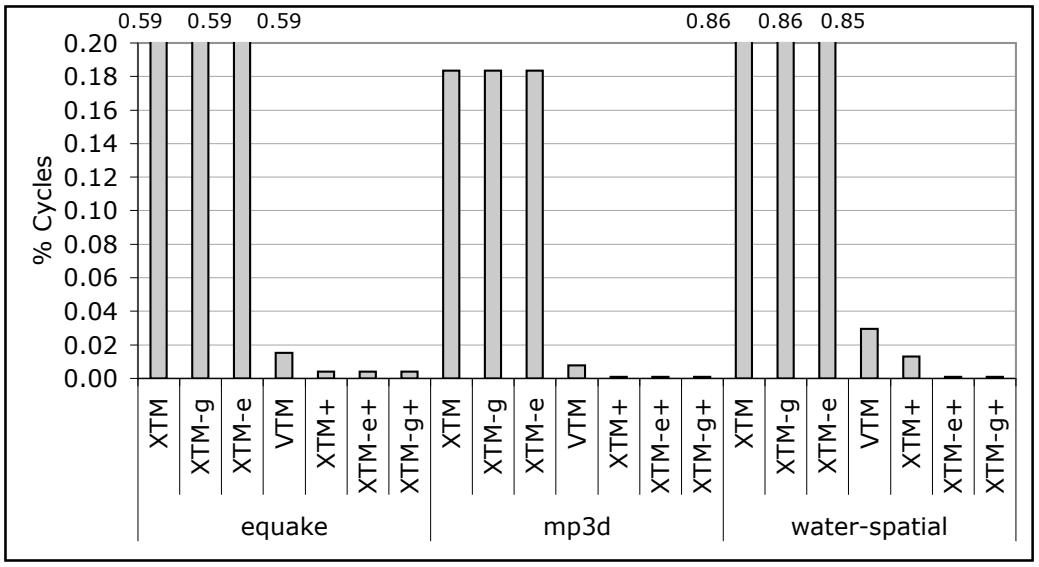


Figure 4.8: Time virtualization overhead as a percentage of the total execution time. '+' stands for our time virtualization mechanism described in Section 4.4.3.

we set the handler to be empty. What is interesting is the relative comparison between the different schemes. In all cases, using XTM with the policy that favors aborts over swapping leads to lower overhead even when compared to the hardware-based VTM. The proposed approach essentially eliminates swapping overhead. Using VTM's swapping approach with the XTM system leads to the highest overheads as swapping in XTM is expensive.

4.7.4 Depth Virtualization

None of our programs use a nesting depth that exceeds what the hardware can support (2 to 4 nesting levels) [84]. Hence, to measure nesting virtualization overhead we used a microbenchmark that generates nesting depths that exceed the hardware support. We varied the frequency of deeply nested transactions from 0.5% to 5% (Table 4.5). This range is reasonable given that Chung et.al. [32] found that most programs have less than 1% nested transactions overall and the average nesting depth is 2.2. For this case, we measure only the base XTM: VTM does not support nesting depth virtualization and XTM-g and XTM-e behaves identically to the base XTM for

Nesting Frequency	Versioning	Validation	Commit	Total
5%	42.86%	2.32%	0.42%	45.60%
2%	16.61%	0.84%	0.16%	17.61%
1%	5.54%	0.27%	0.06%	5.87%
0.5%	2.60%	0.12%	0.02%	2.74%

Table 4.5: Nesting depth virtualization overhead.

nesting. XTM added only 5.9% performance overhead to process a 1% frequency of deeply nested transactions and 2.7% overhead for a 0.5% frequency of deeply nested transactions. Hence, XTM can efficiently virtualize the uncommon case of deep nesting depth. If deeply nested transactions become very common, we should probably revisit the HW support as virtualizing frequent nested transactions through XTM can incur significant overheads. Note that in many cases, it is functionally correct to flatten nested transactions when the hardware support is exceeded. Nesting virtualization should be reserved for cases where independent abort may change the program functionality (the nested transaction need not be re-executed after aborting).

4.8 Related Work

The UTM system was the first to recognize the importance of virtualizing transactional memory [6]. It uses cache line granularity, eager versioning, and conflict detection. UTM supports space and time virtualization, but does not virtualize nesting depth. Unlike most other proposals that start with a limited hardware TM system and add virtualization support, UTM starts with a virtualized system and provides some acceleration through caching. UTM relies on the XSTATE data structure in virtual memory, which is a log with information on the read- and write-set of executing transactions. Portions of the XSTATE can be cached by each processor for faster access. The UTM implementation is rather idealized as it assumes all memory locations are appended with a pointer to XSTATE. On cache misses, a processor must always follow the pointer to detect conflicts. It also relies on the availability of global virtual addresses, which are not available in most popular architectures. Overall,

UTM is space inefficient and incurs significant overheads even in some common cases (e.g., cache miss, no conflict). The same paper introduces LTM, a more practical TM system that allows transactions to overflow caches, but does not allow them to survive virtual memory paging, context switches, or migration [6].

LogTM operates at cache line granularity and uses eager versioning and conflict detection [90]. It allows transactions to survive cache overflows, but not paging, context switches, or excessive nesting. LogTM uses a cacheable, in-memory log to record undo information that is used if a transaction aborts. Evicted log entries leave a bit set in the cache, which enables the processor to continue to check for conflicts on that line when requests arrive from other processors.

Proposed software TM implementations also provide transactional semantics without hardware constraints as they are always built on top of virtual memory [134, 62, 64, 82, 1]. This thesis focuses on virtualization for hardware TM systems because they provide transactional semantics with minimal overheads and make the implementation details transparent to software. XTM can also be seen as a hybrid TM system, as it support transactions in both hardware and software modes. Unlike [75] and [89] that use user-level software and compiler support, XTM uses kernel-mode software. XTM is completely transparent to all levels of user software (application and compiler).

HybridTM [41] switches to software TM systems at the demand on TM virtualization. They require two versions of application code: one for hardware TM system and the other for software TM system. Page-based TM (PTM) maintains in hardware a shadow page table when transactional data overflow hardware caches [29]. When a cache-line with transactional data or metadata is evicted, PTM allocates a new page (shadow page) to store the last committed version of the data and uses the old page to store the overflowed data. The shadow page table maintains the proper mapping information including a per page write summary vector that indicates which blocks in the page have overflowed.

XTM builds upon the research on page-based, cache-coherent DSM systems [143, 129]. Unlike page-based DSM, XTM is a backup mechanism utilized only in the uncommon case when hardware resources are exhausted. XTM also draws on research

that uses virtual memory to implement transactional semantics for the purpose of persistent storage [128, 79].

4.9 Conclustion

This chapter propose eXtended Transactional Memory (XTM) that virtualizes all three TM aspects: space, time, and nesting depth. XTM is a software-only approach that requires no hardware support since, in the common case, virtualization will be invoked infrequently. XTM operates at the operating system level and handles transactional state at the granularity of pages. We also present two enhancements to the base XTM, XTM-g and XTM-e, that use limited hardware support to address basic performance overheads. Finally, we provide the first quantitative evaluation of TM virtualization schemes, which included all three XTM schemes and a hardware-based alternative (VTM). We find that, despite their being software based, the XTM designs provide similar performance to VTM with the cache sizes easily affordable in modern CMPs (e.g., 32KB). Even for demanding applications, emulated by microbenchmarks in our experiments, XTM-g and XTM-e show competitive or better performance. Overall, XTM provides a fully-featured and flexible solution for the virtualization of TM hardware at a low hardware cost. Using XTM, software developers can use TM to simplify parallel code without implementation-specific constraints.

Chapter 5

Thread-Safe DBT Using TM

5.1 Introduction

Dynamic binary translation (DBT) has become a versatile tool that addresses a wide range of system challenges while maintaining backwards compatibility for legacy software. DBT has been successfully deployed in commercial and research environments to support full-system virtualization [147], cross-ISA binary compatibility [28, 124], security analysis [34, 73, 102, 118, 130], debuggers [100], and performance optimization frameworks [10, 22, 142]. DBT facilitates deployment of new tools by eliminating the need to recompile or modify existing software. It also enables the use of newer hardware which need not be fully compatible with the old software architecture.

However, DBT frameworks may incorrectly handle multithreaded programs due to races involving updates to the application data and the corresponding metadata maintained by the DBT. Existing DBT frameworks handle this issue by serializing threads, disallowing multithreaded programs, or requiring explicit use of locks [18, 100].

This chapter presents a practical solution to use TM for correct execution of multithreaded programs within DBT frameworks. We use TM to eliminate races involving metadata. The DBT uses memory transactions to encapsulate the data and metadata accesses in a trace, within one atomic block. This approach guarantees correct execution of concurrent threads of the translated program, as TM mechanisms

detect and correct races.

This chapter is organized as follows. Section 5.2 summarizes DBT technology and the challenges posed by multithreaded binaries. Sections 5.3 and 5.4 present the use of TM for DBT metadata atomicity. Section 5.5 describes our prototype system for DBT-based secure execution of multithreaded programs. Section 5.6 presents the performance evaluation, Section 5.7 discusses related work.

5.2 Dynamic Binary Translation (DBT)

5.2.1 DBT Overview

DBT relies on runtime code instrumentation to dynamically translate and execute application binaries. Before executing a basic block from the program, the DBT copies the block into a code cache and performs any required instrumentation. The DBT system controls all code execution and only translated code from the code cache is executed. Any control flow that cannot be resolved statically, such as indirect branches, invokes the DBT to ensure that the branch destination is in the code cache. Frequently executed basic blocks in the code cache may be merged into a longer trace. This reduces runtime overhead by allowing common hot paths to execute almost completely from the code cache without invoking the DBT system.

DBT may arbitrarily add, insert, or replace instructions when translating code. This powerful capability allows DBT to serve as a platform for dynamic analysis tools. DBT-based tools have been developed for tasks such as runtime optimization [22], bug detection [100], buffer overflow protection [73], and profiling [80]. A general-purpose DBT framework implements basic functionality such as program initialization, code cache management, and trace creation. It also provides developers with an interface (API) that they can use to implement tools on top of the DBT. Using the API, developers specify which, how, and when instruction sequences are instrumented. Finally, the DBT typically performs several optimizations on the instrumented code such as function inlining, common subexpression elimination, and scheduling.

DBT tools often maintain *metadata* that describe the state of memory locations

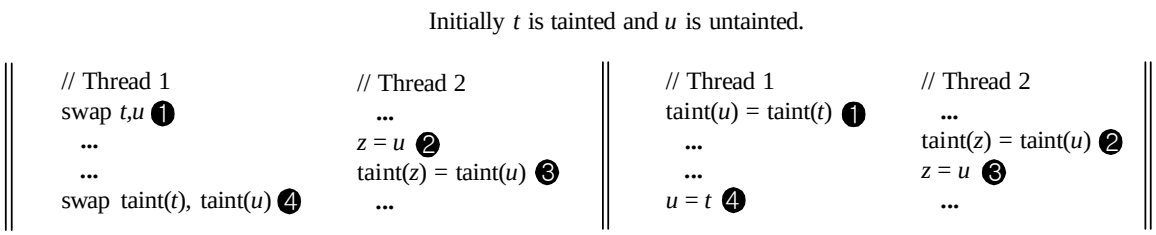
during program execution. For example, metadata can indicate if a memory location is allocated, if it contains secure data, or the number of times it has been accessed. The granularity and size of metadata can vary greatly from one tool to another. Some tools may keep instrumentation information at a coarse granularity such as function, page, or object, while other tools may require basic block, word, or even bit-level granularity. The popular Memcheck tool for the Valgrind framework uses a combination of heap object, byte and bit-level metadata to detect uninitialized variables, memory corruption, and memory leaks [99, 133]. Metadata are allocated in a *separate* memory region to avoid interference with the data layout assumed by the original program. Whenever the program code operates on data, the DBT tool inserts code to perform the proper operations on the corresponding metadata. The additional code can be anything from a single instruction to a full function call, depending on the tool.

5.2.2 Case Study: DBT-based DIFT Tool

In this thesis, we use dynamic information flow tracking (DIFT) as a specific example of a metadata-based DBT tool. However, the issues with multithreaded programs are the same for all other metadata-based DBT tools (profiling, bug detection, fault recovery, etc.). DIFT is particularly interesting because it provides security features that are important for multithreaded server applications such as web servers.

DIFT is a powerful dynamic analysis that can prevent a wide range of security issues [37, 102]. DIFT tracks the flow of untrusted information by associating a taint bit with each memory byte and each register. The OS taints information from untrusted sources such as the network. Any instruction that operates on tainted data *propagates* the taint bit to its destination operands. Malicious attacks are prevented by *checking* the taint bit on critical operations. For example, checking that code pointers and the code itself are not tainted allows for the prevention of buffer overflows and format string attacks.

There are several DBT-based DIFT systems [18, 33, 102, 118]. The metadata maintained by the DBT tool are the taint bits. Code is instrumented to propagate



(a) Race on metadata swap (data updated first) (b) Race on metadata store (metadata updated first)

Figure 5.1: Two examples of races on metadata (taint bit) accesses.

taint bits on arithmetic and memory instructions and check the taint bits on indirect jumps. The research focus has been primarily on reducing overheads using optimizations such as eliminating propagation and checks on known safe data and merging checks when possible [118]. For I/O-bound applications, the overhead of DBT-based DIFT over the original runtime (without DBT) is as low as 6% [118]. However, none of the DBT-based DIFT tools provide highly performant, safe support for multithreaded programs.

5.2.3 Metadata Races

DBTs cannot be readily applied to multithreaded binaries because of races on metadata access. Metadata are stored separately from the regular data. They are also operated upon by separate instructions. Since the DBT metadata can be of arbitrary size and granularity, it is impossible to provide hardware instructions to atomically update data and metadata in the general case. The original program is unaware of the DBT metadata and thus, cannot use synchronization to prevent metadata races. Hence, if the DBT does not provide additional mechanisms for synchronization, any concurrent access to a *(data, metadata)* pair may lead to a race.

Figure 5.1 provides two examples of races for a DBT-based DIFT tool. Consider a multithreaded program operating on variables `t` and `u`. The DBT introduces the corresponding operations on the metadata, `taint(t)` and `taint(u)`. Initially, `t` is tainted (untrusted) and `u` is untainted (trusted). In Figure 5.1.(a), thread 1 swaps `t` and `u` using a single atomic instruction (❶). The DBT inserts a subsequent instruction that swaps `taint(t)` and `taint(u)` as well (❹). However, the pair of swaps is not

an atomic operation. As thread 2 is concurrently reading `u` (❷), it gets the new value of `u` and the old value of `taint(u)` (❸). Even though thread 2 will use the untrusted information derived from `t`, the corresponding taint bit indicates that this is safe data. If `z` is later used as code or as a code pointer, an undetected security breach will occur (false negative) that may allow an attacker to take over the system.

Figure 5.1.(b) shows a second example in which the DBT updates the metadata before the actual data. Thread 1 uses a single instruction to copy `t` into `u` (❹). The previous instruction does the same to the metadata (❶). However, the pair of instructions is not atomic. As thread 2 is concurrently reading `u`, it gets the new value of the metadata (❷) and the old value of the data (❸). Even though thread 2 will use the original, safe information in `u`, the corresponding taint bit indicates that it is untrusted. If `z` is later used as a code pointer or code, an erroneous security breach will be reported (false positive) that will unnecessarily terminate a legitimate program.

In general, one can construct numerous scenarios with races in updates to *(data, metadata)* pairs. Depending on the exact use of the DBT metadata, the races can lead to incorrect results, program termination, undetected malicious actions, etc. Current DBT systems do not handle this problem in a manner that is both functionally-correct and fast. Some DBTs support multithreaded programs by serializing threads, which eliminates all performance benefits of multithreading [100]. Note that in this case, thread switching must be carefully placed so that it does not occur in the middle of a non-atomic *(data, metadata)* update. Other DBTs assume that tool developers will handle this problem using locks in their instrumentation code [22, 80]. If the developer uses coarse-grain locks to enforce update atomicity, all threads will be serialized. Fine-grain locks are error-prone to use. They also lead to significant overheads if a lock acquisition and release is needed in order to protect a couple of instructions (data and metadata update). More importantly, since locks introduce memory fences, their use reduces the efficiency of DBT optimizations such as common sub-expression elimination and scheduling.

5.2.4 Implications

Metadata races can be an important problem for *any* multithreaded program. Even if the application is race-free to begin with, the introduction of metadata breaks the assumed atomicity of basic ISA instructions such as aligned store or compare-and-swap. Numerous multithreaded applications rely on the atomicity of such instructions to build higher-level mechanisms such as locks, barriers, read-copy-update (RCU) [85], and lock-free data-structures (LFDS) [63]. It is unreasonable to expect that the DBT or tool developers will predict all possible uses of such instructions in legacy and future binaries, and properly handle the corresponding metadata updates. For instance, it is difficult to tell if the program correctness relies on the atomicity of an aligned store.

We discuss the DBT-based DIFT tool as an illustrating example. Metadata races around locks and barriers do not pose a security threat, as lock variables are not updated with user input. On the other hand, RCU and LFDS manipulate pointers that control the correctness and security of the application and may interact with user input. Hence, metadata races can allow an attacker to bypass the DIFT security mechanisms.

Read-copy update (RCU) is used with data-structures that are read much more frequently than they are written. It is a common synchronization technique used in several large applications including the Linux kernel. RCU directly updates pointers in the protected data structure, relying on the atomicity of aligned, word-length stores. These pointers may be influenced by user input and may reference objects that include critical code pointers. For instance, Linux uses RCU to protect core components such as IPv4 route caches, directory caches, and file descriptors. NSA's Security Enhanced Linux uses RCU to protect access control information [131].

Lock-free data structures use atomic instructions such as compare-and-swap (CAS) to allow concurrent access without conventional locking [63]. While they are complex to implement, LFDS versions of queues, lists, and hash-tables are often part of libraries used in performance-critical software. Since they manipulate pointers within data structures, LFDS may access data from untrusted sources.

Additionally, attackers may use memory safety vulnerabilities to deliberately introduce race conditions into race-free applications. Attacks such as buffer overflows

and format string vulnerabilities give the attacker the ability to write anywhere in the program's address space. For example, if network input is used as an array index, the attacker can use one thread to overwrite thread-private or stack data in other threads. This essentially bypasses any techniques used to guarantee race-freedom in the original code. Moreover, an attacker can target metadata races in order to bypass security checks in DBT tools such as DIFT. By having one thread write malicious code or data to another thread's stack or thread-local data, an attacker can induce false negatives similar to the one in Figure 5.1.(a). Hence, the attacker may be able to overwrite critical pointers such as return addresses without setting the corresponding taint bit. This attack would not be possible on a single-threaded program as the DIFT propagation and checks would flag the overwritten data as untrusted.

5.3 DBT + TM = Thread-Safe DBT

To address metadata races in multithreaded programs, we propose the use of transactional memory (TM) within DBT systems. This section concentrates on the functional correctness of this approach, while Section 5.4 focuses on performance issues.

5.3.1 Using Transactions in the DBT

DBT systems can eliminate metadata races by wrapping any access to a *(data, metadata)* pair within a transaction. Figure 5.2 and 5.3 show the code for the two race condition examples in Section 5.2.3, instrumented with transactions. The additional code defines transaction boundaries and provides the read/write barriers required by STM and hybrid systems. In Figure 5.2, thread 1 encloses the data and metadata swaps within one transaction. Hence, even if thread 2 attempts to read `u` in between the two swaps, the TM system will detect a transaction conflict. By rolling back one of the two transactions, the TM system will ensure that thread 2 will read either the old values of both `u` and `taint(u)`, or the new values for both after the swap. Similarly, transactions eliminate the race in Figure 5.3. Since transactions are atomic, the order of data and metadata accesses within a transaction does not matter.

Initially `t` is tainted and `u` is untainted.

```
// Thread 1                      // Thread 2
Tx_Begin                         ...
 wrbarrier(t)                    Tx_Begin
 wrbarrier(u)                     rdbarrier(u)
 swap t,u                         wrbarrier(z)
 ...                              z = u
 ...                              rdbarrier(taint(u))
 wrbarrier(taint(t))              wrbarrier(taint(z))
 wrbarrier(taint(u))              taint(z) = taint(u)
 swap taint(t), taint(u)         Tx_End
Tx_End                           ...
```

Figure 5.2: Using transactions to eliminate the metadata swap race. The read and write barriers are necessary for STM and hybrid TM systems, but not for HTM systems.

For now, we assume that all data and metadata accesses are tracked for transactional conflicts in order to guarantee correctness. This is the default behavior of hardware TM systems. For software and hybrid TM systems, this requires at least one barrier for each address accessed within a transaction. We revisit this issue in Section 5.4 when we discuss optimizations. We focus here on the placement of transaction boundaries. To guarantee correctness, data and corresponding metadata accesses must be enclosed within a single transaction. Fine-grain transactions, however, lead to significant performance loss as they do not amortize the overhead of starting and ending transactions. Moreover, they interfere with many DBT optimizations by reducing their scope. To partially alleviate these problems, our base design introduces transactions at basic block boundaries, enclosing multiple accesses to data and metadata pairs as shown in Figure 5.4.(a). The Tx_Begin statement is added at the beginning of the block. In most cases, the Tx_End statement is added after the last non control-flow instruction in the block (branch or jump).

The Tx_End placement is complicated for any DBT-based tool that associates metadata operations with the control-flow instruction that ends a basic block. For instance, DIFT must check the address of any indirect branch prior to taking the branch. If the address is tainted, a security attack is reported. In this case, the

Initially `t` is tainted and `u` is untainted.

```
// Thread 1                  // Thread 2
Tx_Begin                     ...
 rdbarrier(taint(t))         Tx_Begin
 wrbarrier(taint(u))          rdbarrier(taint(u))
 taint(u) = taint(t)          wrbarrier(taint(z))
 ...                          taint(z) = taint(u)
 ...                          rdbarrier(u)
 rdbarrier(t)                 wrbarrier(z)
 wrbarrier(u)                 z = u
 u = t                       Tx_End
Tx_End                       ...
```

Figure 5.3: Using transactions to eliminate the metadata store race. The read and write barriers are necessary for STM and hybrid TM systems, but not for HTM systems.

DBT must introduce code that checks the metadata for the jump target in the same transaction as the control flow instruction to avoid metadata races. We handle this case by introducing the Tx_End statement at the beginning of the basic block that is the target of the indirect jump. For conditional branches, we introduce Tx_End at the beginning of both the fall-through block and the target block as shown in Figure 5.4.(b). If the target block includes a transaction to cover its own metadata accesses, Tx_End is immediately followed by a Tx_Begin statement.

5.3.2 Discussion

The DBT transactions may experience false conflicts due to the granularity of conflict detection in the underlying TM system, and the layout of data and metadata in memory. False conflicts can be a performance issue, but do not pose correctness challenges. Even if the contention management policy of the TM system does not address fairness, the DBT can intervene and use fine-grain transactions to avoid these issues.

The original binary may also include user-defined transactions. If a user-defined transaction fully encloses a DBT transaction (or vice versa), the TM system will

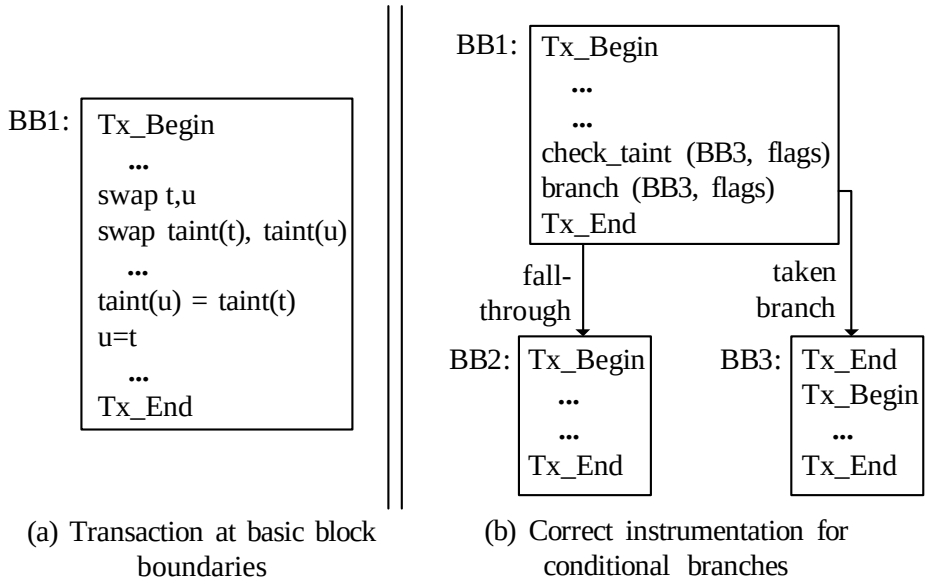


Figure 5.4: Transaction instrumentation by the DBT.

handle it correctly using mechanisms for nested transactions (subsuming or nesting with independent rollback). Partial overlapping of two transactions is problematic. To avoid this case, the DBT can split its transactions so that partial overlapping does not occur. As long as the accesses to each *(data, metadata)* pair are contained within one transaction, splitting a basic block into multiple transactions does not cause correctness issues.

A challenge for transactional execution is handling I/O operations. For DBT transactions, I/O is not an issue as the DBT can terminate its transactions at I/O operations. Such operations typically terminate basic blocks or traces, and act as barriers for DBT optimizations. Handling I/O operations within user-defined transactions is beyond the scope of this work.

A final issue that can occur if DBT transactions span multiple basic blocks (see Section 5.4), is that of conditional synchronization in user code. Figure 5.5 presents a case where transactions enclose code that implements flag-based synchronization. Without transactions, this code executes correctly. With transactions, however, execution is incorrect as the two transactions are not serializable. For correctness, statement ❶ must complete after statement ❷ and statement ❸ after statement ❹.

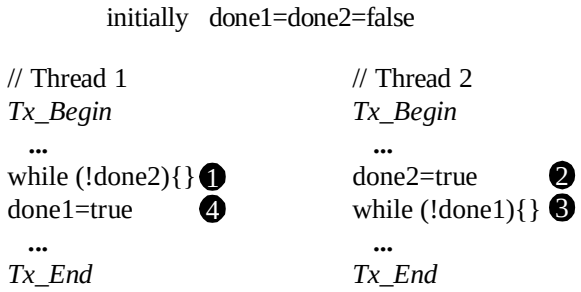


Figure 5.5: The case of a conditional wait construct within a DBT transaction.

Once transactions are introduced, statements ❶ and ❹ must complete atomically. Similarly, statements ❷ and ❸ must complete atomically. Regardless of the type of the TM system and the contention management policy used, the code in Figure 5.5 will lead to a livelock or deadlock. We handle this case dynamically. If the DBT runtime system notices that there is no forward progress (timeout on a certain trace or repeated rollbacks of transactions), it re-instruments and re-optimizes that code to use one transaction per basic block.

5.4 Optimizations for DBT Transactions

The use of transactions in DBT eliminates metadata races for multithreaded programs. However, transactions introduce three sources of overhead: the overhead of starting and ending transactions; that of the read and write barriers for transactional bookkeeping; and the cost of rollbacks and re-execution. This section focuses on the first two sources of overhead, which are particularly important for STM and hybrid systems. In Section 5.6, we show that rollbacks and are not common for DBT transactions.

5.4.1 Overhead of Starting/Ending Transactions

Longer transactions improve performance in two ways. First, they amortize the cost of starting and ending a transaction. Starting a transaction includes checkpointing register state. Ending a transaction requires clean up of any structures used for

versioning and conflict detection. Second, long transactions increase the scope of DBT optimizations, such as common subexpression elimination, instruction scheduling, and removal of redundant barriers [1].

There are two ways to increase the length of DBT transactions:

- **Transactions at trace granularity:** Modern DBT frameworks merge multiple basic blocks into hot traces in order to increase the scope of optimizations and reduce the number of callbacks to the DBT engine [18, 80, 100]. We exploit this feature by introducing transactions at the boundaries of DBT traces. As the DBT creates or extends traces, it also inserts the appropriate Tx_Begin and Tx_End statements.

- **Dynamic transaction merging:** We can also attempt to dynamically merge transactions as DBT traces execute. In this case, the DBT introduces instrumentation to count the amount of work per transaction (e.g., the number of instructions). When the transaction reaches the Tx_End statement, the STM checks if the work thus far is sufficient to amortize the cost of Tx_Begin and Tx_End. If not, the current transaction is merged with the succeeding one. Dynamic transaction merging is especially helpful if DBT transactions are relatively common, as is the case with a DBT-based DIFT tool.

While both methods produce longer transactions, they differ in their behavior. Trace-level transactions incur additional overhead only when traces are created. However, the transaction length is limited by that of the DBT trace. Dynamic transaction merging incurs some additional overhead as traces execute, but can create transactions that span multiple traces. Both approaches must take into account that increasing the transaction length beyond a certain point leads to diminishing returns. Moreover, very long transactions are likely to create more conflicts. The ideal length of transactions depends on the underlying TM system as well. Hardware and hybrid TM systems typically use hardware mechanisms for register checkpointing and allow shorter transactions to amortize the fixed overheads.

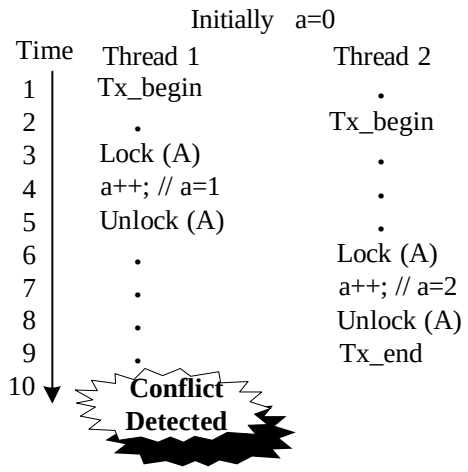


Figure 5.6: An example showing the need for TM barriers on data accesses even in race-free programs.

5.4.2 Overhead of Read/Write Barriers

For STM and hybrid TM systems, the DBT must introduce read and write barriers in order to implement conflict detection and data versioning in transactions. Despite optimizations such as assembly-level tuning of barrier code, inlining, and elimination of repeated barriers, the performance overhead of software bookkeeping is significant [1]. Hence, we discuss techniques that reduce the number read and write barriers for DBT transactions.

Basic Considerations for Barrier Use

In Section 5.3, we stated that all data and metadata accesses in a DBT transaction should be protected with a TM barrier. This is definitely the case for DBT-based DIFT tools. During a buffer overflow attack, a thread may access any location in the application's address space (see Section 5.2.4). Unless we use barriers for all addresses, we may miss metadata races that lead to false negatives or false positives in the DIFT analysis. For other DBT-based tools that require metadata, it may be possible to eliminate several barriers on data or metadata accesses based on knowledge about the sharing behavior of threads. One has to be careful about barrier optimizations, as aggressive optimization can lead to incorrect execution. One must also keep in

mind that TM barriers implement two functions: they facilitate conflict detection and enable rolling back the updates of aborted transactions.

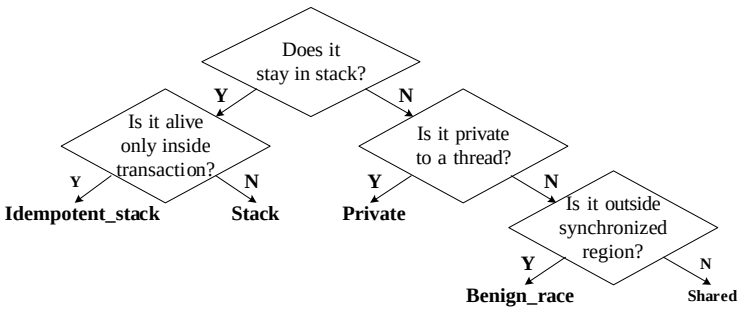
For instance, assuming that the original program is race-free, one may be tempted to insert barriers on metadata accesses but eliminate all barriers on the accesses to the original data. Figure 5.6 shows a counter example. The two threads use a lock to increment variable `a` in a race-free manner. The DBT introduces transactions that enclose the lock-protected accesses. The transactions include TM barriers for metadata but not for `a`. At runtime, thread 1 sets `a` to 1 at timestep 4 and thread 2 updates `a` to 2 at timestep 7. At timestep 10, the TM system detects a conflict between the transaction in thread 1 and another thread due to a different metadata access. The system decides to rollback the transaction of thread 1. Since there was no barrier on the access to `a`, its value cannot be rolled back as there is no undo-log or write-buffer entry.

Now, assume that we execute the same code with a TM barrier on `a` that does data versioning (e.g., creates an undo-log entry) but does not perform conflict detection on `a`. At timestep 4, thread 1 will log 0 as the old value of `a`. When the system rolls back thread 1 at timestep 10, it will restore `a` to 0. This is incorrect as it also eliminates the update done by thread 2. The correct handling of this case is to insert TM barriers for `a` that perform both conflict detection and data versioning, even though the original code had locks to guarantee race-free accesses to `a`.

Optimizations Using Access Categorization

For DBT-based tools unlike DIFT, it is possible to eliminate or simplify certain TM barriers by carefully considering data access types. Figure 5.7 presents the five access types we consider and the least expensive type of TM barrier necessary to guarantee correct atomic execution of the code produced by the DBT:

- **Stack and Idempotent_stack accesses**: For accesses to thread-local variables on the stack [54], we need barriers for data versioning but not for conflict detection. Moreover, if the access does not escape the scope of the current transaction (Idempotent_stack) there is no need for TM barriers at all.



(a) Access Categorization Process

Access Type	Conflict Detection Barrier	Data Versioning Barrier
Idempotent_ stack	No	No
Stack	No	Yes
Private	No	Yes
Benign_race	No	Yes
Shared	Yes	Yes

(b) STM barrier needed for each Access Type

Figure 5.7: The five data access types and their required TM barriers.

- **Private accesses**: Similar to stack variables, certain heap-allocated variables are thread-local. Their accesses do not require barriers for conflict detection. If the transaction updates the variable, a TM barrier is needed for data versioning.

- **Benign_race accesses**: There are access to shared variables that are not protected through synchronization in the original program. Assuming a well-synchronized application, any races between accesses to these variables are benign. Hence, there is no need for TM barriers for conflict detection, but there may be barriers for data versioning on write accesses.

- **Shared accesses**: Any remaining access must be assumed to operate on truly shared data. Hence, the DBT framework must insert full read and write barriers depending on the type of access. The only optimization possible is to eliminate any repeated barriers to the same variable within a long transaction [1].

The DBT framework classifies accesses during trace generation using well-known techniques such as stack escape and dynamic escape analysis [54, 135]. It also adds instrumentation that collects the information necessary for runtime classification [135]. The success of the classification partially depends on the nature of the DBT. For example, the information available in object-based binaries such as Java bytecode increases the analysis accuracy and provides additional optimization opportunities.

To identify Benign_race accesses, the DBT must know the synchronization primitives used by the application. It can be the case that the lack of synchronization is

Instruction Type	Example	Taint Propagation
ALU operation	r3 = r1 + r2	Taint[r3] = Taint[r1] OR Taint[r2]
Load	r3 = M[r1+r2]	Taint[r3] = Taint[M[r1+r2]]
Store	M[r1+r2] = r3	Taint[M[r1+r2]] = Taint[r3]
Register clear	r1 = r1 xor r1	Taint[r1] = 0
Bounds check	cmp r1, 256	Taint[r1] = 0

Table 5.1: Taint bit propagation rules for DIFT.

actually a bug. The translated code may change if and when the bug manifests in the execution. This problem is not specific to DBTs. Any small system perturbation such as the use of a faster processor, the use of more threads, or a change in memory allocation may be sufficient to mask or expose a race.

5.5 Prototype System

To evaluate the use of transactions in DBT, we used the Pin dynamic binary translator for x86 binaries [80]. We implemented DIFT as a Pintool and ran it on top of a software TM system.

5.5.1 DIFT Implementation

The analysis in our DIFT tool is similar to the one in [102, 118]. We maintain a taint bit for every byte in registers and main memory to mark untrusted data. Any instruction that updates the data must also update the corresponding taint bit. Table 5.1 summarizes the propagation rules. For instructions with multiple source operands, we use logical OR to derive the taint bit for the destination operand. Hence, if any of the sources are tainted, the destination will be tainted as well.

To execute real-world binaries without false positives, register clearing and bounds check instructions must be recognized and handled specially. For the x86 architecture, instructions such as `sub %eax, %eax` and `xor %eax, %eax` are often used to clear registers as they write a constant zero value to their destination, regardless of the source values. The DIFT tool recognizes such instructions and clears the taint bit

for the destination register. Programs sometimes validate the safety of untrusted input by performing a bounds check. After validation, an input can be used as a jump address without resulting in a security breach. We recognize bounds checks by untainting a register if it is compared with a constant value [102, 118].

To prevent memory corruption attacks, taint bits are checked on two occasions. First, when new code is inserted into the code cache, we check the taint bits for the corresponding instructions. This check prevents code injection attacks. Second, we check the taint bit for the operands of indirect control-flow operations (e.g., register-indirect jump or procedure call/return). This ensures that an attacker cannot manipulate the application's control-flow.

5.5.2 Software TM System

We implemented an STM system similar to the one proposed in [1]. For read accesses, the STM uses optimistic concurrency control with version numbers. For writes, it uses two-phase locking and eager versioning with a per-thread undo-log. The conflict detection occurs at word granularity using a system-wide lock table with 2^{10} entries. Each lock word contains a transaction ID when locked and a version number when unlocked. The least significant bit identifies if the word is locked or not. Given a word address, the corresponding lock word is identified using a hash function. This approach may lead to some false conflicts but can support transactional execution of C and C++ programs.

Figure 5.8 shows the pseudocode for the STM. Tx_Begin() clears the per-transaction data structures for data versioning and conflict detection and takes a register checkpoint using the Pin API. Three barrier functions are provided to annotate transactional accesses. RD_barrier() adds the address to the read-set for conflict detection. WR_barrier() adds the address to the write-set for conflict detection and creates an undo-log entry with the current value of the address. WRlocal_barrier() does not create an undo-log entry but adds the address into the write-set. This barrier is used for the Stack, Private, and Benign_race access types.

WR_barrier() first checks the lock word for the address using compare-and-swap.

```
Tx_Begin(){
 PIN::Checkpoint(buf);
 RdSet.clear();
 WrSet.clear();
 UndoQ.clear();}

Tx_Commit(){
 foreach addr in RdSet{
  lock=lockTable.get(addr);
  if(lock!=myID &&
     lock!=RdSet.get(addr))
  {
   Tx_Abort();} }
 foreach addr in WrSet{
  nextVer=WrSet.get(addr)+2;
  lockTable.set(addr,
                nextVer);}}

Tx_Abort(){
 foreach addr in UndoQ{
  *addr=UndoQ.get(addr);}
 foreach addr in WrSet{
  nextVer=WrSet.get(addr)+2;
  lockTable.set(addr,
                nextVer);}
 PIN::Restore(but);}
```

```
WR_barrier(addr){
 lock=lockTable.get(addr);
 if(lock==myID){
  UndoQ.insert(addr,*addr);
  return;
 } elif(lock%2==0 &&
    CAS(myID,lock,addr))
 {
  WrSet.insert(addr,lock);
  UndoQ.insert(addr,*addr);
  return;}
 Tx_Abort();}

RD_barrier(addr){
 lock=lockTable.get(addr);
 if(lock==myID){
  return;
 } elif(lock%2==0){
  RdSet.insert(addr,lock);
  return;}
 Tx_Abort();}
```

Figure 5.8: The pseudocode for a subset of the STM system.

If it is locked by another transaction, a conflict is signaled. If not, the transaction remembers the current version number and sets its ID in the lock word. It then creates the undo-log entry. Note that there can be multiple entries for the same address because we do not check for duplicates. Since the transactions generated the DBT are typically small (50 to 200 instructions), duplicates are not common. Hence, we prefer to waste some storage on the occasional duplicate rather than take the time to search through the undo-log on each WR_barrier() call. RD_barrier starts by checking the corresponding lock word. If it is locked, a conflict is signaled. If not, the version number is recorded in the read-set.

	Register Checkpointing	Conflict Detection	Data Versioning
STM	SW (multi-cycle)	SW read-set /write-set	SW undo-log
STM+	HW (single-cycle)	SW read-set /write-set	SW undo-log
HybridTM	HW (single-cycle)	HW signatures	SW undo-log
HTM	HW (single-cycle)	HW read-set /write-set	HW undo-log

Table 5.2: The characteristics of the four TM systems evaluated in this thesis.

Tx_End() first validates the transaction. The lock words for all addresses in the read-set are checked. If any entry has a different version number than the one in the read-set, or is locked by another transaction, a conflict is signaled. Once the read-set is validated, the transaction commits by releasing the locks for all addresses in the write-set. The version numbers in the corresponding lock words are incremented by 2 to indicate that the data has been updated. On a conflict, the transaction rolls back by applying the undo-log and then releasing the locks for addresses in its write-set. It then restores the register checkpoint. We re-execute aborted transactions after a randomized backoff period.

5.5.3 Emulation of Hardware Support for TM

We also evaluated the overhead of DBT transactions by emulating three systems with hardware support for transactional execution. Table 5.2 summarizes their characteristics. We used emulation instead of simulation because Pin is available only in binary form. Hence, it is very difficult to run Pin on a hardware simulator with ISA extensions that control the TM hardware support.

The first hardware system, *STM+*, is the same as our initial STM but uses a fast, hardware-based mechanism for register checkpointing. Such mechanisms are generally useful for speculative execution and are likely to be common in out-of-order

processors [3]. We emulate the runtime overhead of hardware-based checkpointing by substituting the call to the Pin checkpointing function with a single load instruction. The second hardware system, *HybridTM*, follows the proposals for hardware acceleration of software transaction conflict detection [23, 126, 136]. Specifically, we target the SigTM system that uses hardware signatures for fast conflict detection [23]. We emulate HybridTM by substituting the read and write barrier code in Figure 5.8 with the proper number of arithmetic and load/store instructions needed to control the SigTM signatures. HybridTM uses a hardware checkpoint as well but maintains the undo-log in software just like STM. At commit time, it does not need to traverse the read-set or write-set for conflict detection. The final hardware system, *HTM*, represents a full hardware TM system [60, 90]. We emulate it by eliminating the read and write barriers as HTM systems perform transactional bookkeeping transparently in hardware. Register checkpointing takes a single cycle and a successful transaction commit takes 2 cycles (one for validation and one to clear the cache metadata).

When executing a program using STM+, HybridTM, or HTM, we cannot roll back a transaction, as we simply emulate the runtime overhead of transactional bookkeeping without implementing the corresponding functionality. This did not cause any correctness problems during our experiments as we did not launch a security attack against our DIFT tool at the same time. In terms of accuracy, our results for the hardware TM systems do not include the overhead of aborted transactions. As shown in Section 5.6, the abort ratio for DBT transactions is extremely low (less than 0.03% on average). We also do not account for the overhead of false conflicts in hardware signatures or overflows of TM metadata from hardware caches. Since DBT transactions are fairly small, such events are also rare. Hence, we believe that the performance results obtained through HW emulation are indicative of the results that would be obtained if detailed simulation were an option.

5.6 Evaluation

Table 5.3 describes our evaluation environment, which is based on an 8-way SMP x86 server. Our DIFT tool is implemented using the Pin DBT framework [80]. Pin

CPU	4 dual-core Intel Xeon CPUs, 2.66 GHz
L2 Cache	1 MByte per dual-core CPU
Memory	20 GBytes shared memory
Operating System	Linux 2.6.9 (SMP)
DBT framework	Pin for IA32 (x86) Linux

Table 5.3: The evaluation environment.

runs on top of Linux 2.6.9 and GCC 3.4.6. We used nine multithreaded applications: *barnes*, *fmm*, *radix*, *radiosity*, *water*, and *water-spatial* from SPLASH-2 [155]; *equake*, *swim*, and *tomcatv* from SPEComp [141]. These applications are compute-bound and achieve large speedups on SMP systems. They are well-suited for our performance experiments, as the overhead of introducing transactions cannot be hidden behind I/O operations. All applications make use of the Pthreads API.

Table 5.4 presents the basic characteristics of the transactions introduced by our DBT tool when using one transaction per DBT trace. Transactions are relatively short, with the average length varying between 50 and 250 instructions, including those needed for DIFT. The SPEComp benchmarks have longer transactions because Pin extracts longer traces from loop-based computations. When using an STM system without the optimizations in Section 5.4.2, our tool introduces transactions with 10 read barriers and 5 write barriers on average. Transactions rarely abort (less than 0.03% on average). This is expected as these are highly parallel applications with little sharing between parallel threads.

5.6.1 Baseline Overhead of Software Transactions

Figure 5.9 presents the baseline overhead for our DBT tool when using software (STM) transactions without any hardware support for TM. These results are indicative of the performance achievable on existing multiprocessor systems. We measure the overhead by comparing the execution time of the DIFT tool with STM transactions (thread-safe) to the execution time of the DIFT tool without transactions (not thread-safe). Lower overheads are better. The DBT uses one transaction per trace in this case.

Application	# Instr. / # Instr. per Tx	# LD / # LD per Tx	# ST / # ST per Tx	# RD Barriers per Tx	# WR Barriers per Tx	Abort Abort Ratio (%)
Barnes	81.21	9.07	5.49	5.61	3.60	0.01
Equake	118.68	15.42	4.90	9.93	3.35	0.02
Fmm	111.42	14.60	8.28	8.77	5.68	0.03
Radiosity	61.85	7.27	5.44	4.02	3.18	0.00
Radix	118.70	18.89	12.30	11.00	l7.29	0.02
Swim	249.76	34.20	6.79	20.81	4.92	0.03
Tomcatv	118.77	13.19	6.13	8.78	4.13	0.00
Water	55.25	7.31	3.09	4.33	2.07	0.00
Water-spatial	60.93	8.03	3.79	4.81	2.53	0.00

Table 5.4: The characteristics of software (STM) transactions introduced by our DBT tool.

The average runtime overhead for software transactions is 41%. The cost of transactions is relatively low, considering that they make the DBT thread-safe and allow speedups of 4x to 8x on the measured system systems. The STM overhead varies between 26% (radix) and 56% (fmm). This variance is significantly lower than that observed in previous STM systems. This is because the tool introduces transactions at the level of DBT traces (a few basic blocks) where the influence of the algorithmic difference between applications is diluted to some extent. The exact value of the overhead does not always follow the transaction length and number of read/write barriers presented in Table 5.4 for two reasons. First, STM transactions have a higher impact on cache-bound applications, particularly when the STM barriers have low locality. Second, the averages in Table 5.4 hide significant differences in the exact distribution of the statistics.

5.6.2 Effect of Transaction Length

The results in Figure 5.9 assume one transaction per DBT trace, the largest transactions our DBT environment can support without significant modifications. Figure 5.10 illustrates the importance of transaction length. It shows the overhead of

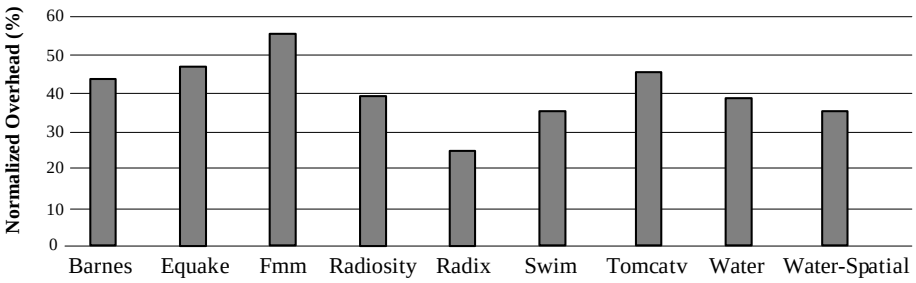


Figure 5.9: Normalized execution time overhead due to the introduction of software (STM) transactions.

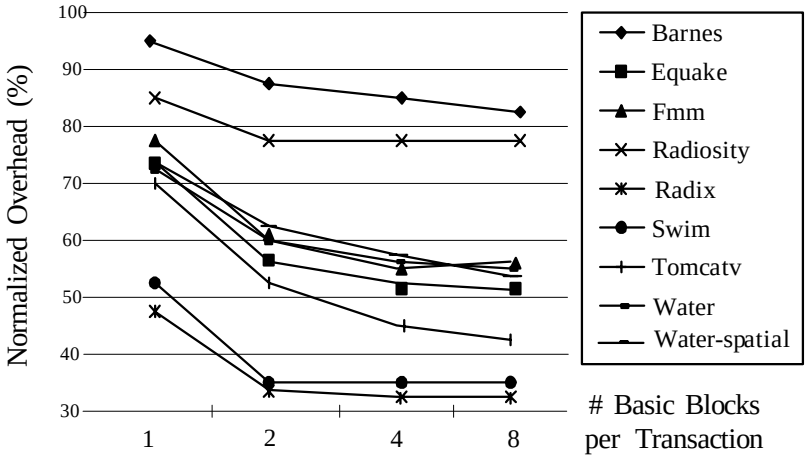


Figure 5.10: The overhead of STM transactions as a function of the maximum number of basic blocks per transaction.

STM transactions in the DBT-based DIFT tool as a function of the maximum number of basic blocks per transaction (1 to 8). If the DBT trace includes fewer basic blocks than the maximum allowed per transaction, the transaction covers the whole trace. Otherwise, the trace includes multiple transactions.

As expected, the overhead of using one basic block per transaction is excessive, more than 70%, for the majority of applications. The computation in one basic block is not long enough to amortize the overhead of STM transactions. As the maximum number of basic blocks per transaction grows, the overhead decreases, as the cost of starting and ending transactions is better amortized. Moreover, larger transactions increase the scope of DBT optimizations and benefit more from locality. At up to 8

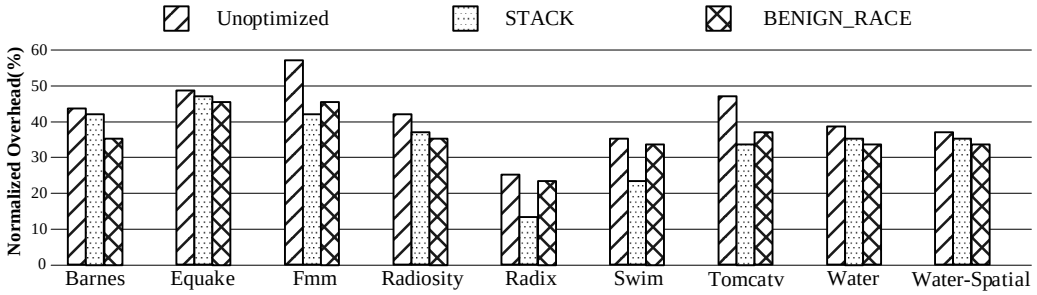


Figure 5.11: The overhead of STM transactions when optimizing TM barriers for Stack and Benign_race accesses.

basic blocks per transaction, several applications reach the low overheads reported in Figure 5.9 because most of their traces include less than 8 basic blocks, or 8 blocks include sufficient computation to amortize transactional overheads. This is not the case for Barnes, Radiosity, Water, and Water-spatial which have longer traces and benefit from longer transactions.

As the results for STM+ suggest in Section 5.6.4, most applications would also benefit from transactions that span across trace boundaries. To support multi-trace transactions, we would need dynamic support for transaction merging as traces execute (see Section 5.4.1).

5.6.3 Effect of Access Categorization

The baseline STM transactions in Figure 5.9 use read and write barriers to protect all accesses to data and metadata. In Section 5.4.2, we discussed how, in certain cases, we can use additional analysis of access types in order to reduce the overhead of STM instrumentation. To measure the effectiveness of these optimizations, we implemented two simple analysis modules in our DBT system that identify Stack and Benign_race accesses respectively. We classify all accesses that are relative to the stack pointer, as Stack. To identify Benign_race accesses, we use the DBT to add a per-thread counter that is incremented on pthread_mutex_lock() and decremented on pthread_mutex_unlock(), the only synchronization primitives used in our applications. A memory access is classified as a Benign_race if it occurs when the counter is zero.

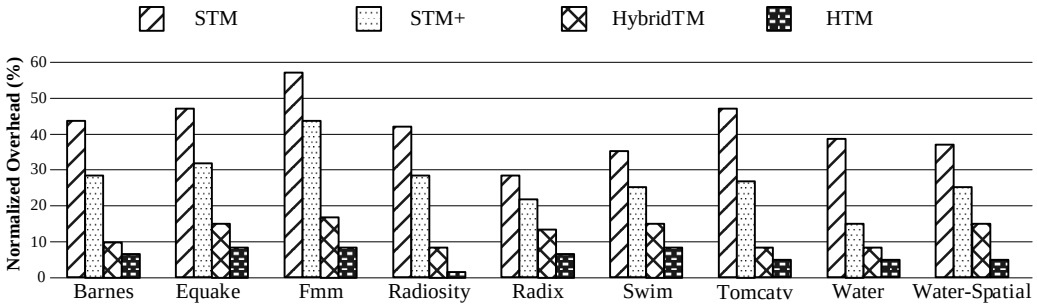


Figure 5.12: Normalized execution time overhead with various schemes for hardware support for transactions.

This analysis requires a priori knowledge of the synchronization functions used by the application. Note that in both analyses, we optimize the STM overhead for data accesses. All metadata accesses are fully instrumented to ensure correctness.

Figure 5.11 shows the impact of the two optimizations on the runtime overhead of STM transactions. The left-most bar (unoptimized) represents the results with full STM instrumentation from Section 5.6.1. Figure 5.11 shows that optimizing STM barriers for Stack accesses reduces the overhead of transactions by as much as 15% (radix) and 7% on the average. Optimizing the STM barriers for Benign_race accesses reduces the overhead by 5% on the average. Overall, the results indicate that software optimizations based on access-type classification can play a role in reducing the overhead of transaction use in DBT-based tools.

5.6.4 Effect of Hardware Support for Transactions

Finally, Figure 5.12 shows the reduction in the overhead as the amount of hardware support for hardware execution increases. As explained in Section 5.5.3, we emulate three hardware schemes: STM+, which provides hardware support for register checkpointing in STM transactions; HybridTM, which uses hardware signatures to accelerate conflict detection for STM transactions; and HTM, a fully-featured hardware TM scheme that supports transactional execution without the need for read or write barriers. For reference, we also include the original results with software-only

transactions.

STM+ reduces the average overhead from 41% to 28%. Hardware checkpointing is particularly useful for small traces for which a software checkpoint of registers dominate execution time. HybridTM reduces the average overhead to 12% as it reduces the overhead of conflict detection in the read and write barriers (read-set and write-set tracking). The full HTM support reduces the overhead of using transactions in the DBT-based DIFT tool down to 6% by eliminating read and write barriers within the traces. Overall, Figure 5.5.3 shows that hardware support is important in reducing the overhead of transactions in DBT tools. Nevertheless, it is not clear if a fully-featured HTM is the most cost-effective approach. The biggest performance benefits come from hardware register checkpointing and hardware support for conflict detection in software transactions.

5.7 Related Work

There have been significant efforts to develop general-purpose DBT systems, such as Dynamo [10], DynamoRIO [22], Valgrind [100], Pin [80], StarDBT [18], and HDtrans [142]. Apart from DIFT-based security tools, these frameworks have been used for performance optimizations [10, 142], profiling [80], memory corruption protection [73], and software bug detection [99, 133]. To the best of our knowledge, no existing DBT framework supports a functionally-correct, low-overhead method for handling metadata consistency issues in multithreaded programs. They require explicit locking by the tool developer, thread serialization, or disallow multithreaded programs altogether. This work provides the first in-depth analysis of the issue and proposes a practical solution.

Significant effort has also been made to develop DIFT as a general purpose technique for protecting against security vulnerabilities. DIFT has been implemented using static compilers [157], dynamic interpreters [103, 109], DBTs [18, 118, 33], and hardware [37, 40]. The advantage of the DBT-based approach is that it renders DIFT applicable to legacy binaries without requiring hardware modifications. The LIFT system proposed a series of optimizations that drastically reduce the runtime

overhead of DBT-based DIFT [118]. Our work complements LIFT by proposing a practical method to extend DBT-based DIFT to multithreaded programs.

The popularity of multi-core chips has motivated several TM research efforts. HTM systems use hardware caches and the coherence protocol to support conflict detection and data versioning during transactional execution [60, 90]. HTMs have low bookkeeping overheads and require minimal changes to user software. STM systems implement all bookkeeping in software by instrumenting read and write accesses within transactions [45, 62, 125]. STMs run on existing hardware and provide full flexibility in terms of features and semantics. To address the overhead of STM instrumentation, researchers have proposed compiler optimizations [1] and hybrid TM systems that provide some hardware support for conflict detection in STM code [23, 126, 136]. More recently, there have also been efforts to use TM mechanisms beyond concurrency control. In [96], TM supports complex compiler optimizations by simplifying compensation code.

5.8 Conclusion

This chapter presents a practical solution for correct execution of multithreaded programs within dynamic binary translation frameworks. To eliminate races on metadata accesses, we proposed the use of transactional memory techniques. The DBT uses transactions to encapsulate all data and metadata accesses within a trace into one atomic block. This approach guarantees correct execution as TM mechanisms detect and correct races on data and metadata updates. It also maintains the high performance of multithreaded execution as DBT transactions can execute concurrently.

To evaluate this approach, we implement a DBT-based tool for secure execution of x86 binaries using dynamic information flow tracking. This is the first such tool that correctly handles multithreaded binaries without serialization. We show that the use of software transactions in the DBT leads to runtime overhead of 40%. We also demonstrate that software optimizations in the DBT and hardware support for transactions can reduce the runtime overhead to 6%. Overall, we show that TM allows metadata-based DBT tools to practically support multithreaded applications.

Chapter 6

Hardware-assisted Memory Snapshot

6.1 Introduction

Lack of concurrency in system software modules such as garbage collectors and memory profilers prevents us from fully exploiting abundant parallelism in chip-multiprocessors (CMPs). Such modules run in concurrently with application code, causing performance degradation due to multiplexing. Ideally, system code could be easily changed to use spare cores in a CMP in order to remove its performance overhead. However, parallelization is not trivial in practice because programmers must deal with the complications of concurrency management in complex system software and its interaction application code that runs concurrently.

This chapter presents *MShot*, a hardware-assisted memory snapshot system to allow for algorithmic simplicity, easy code management, and performance at the same time. Our key observation is that the hardware requirements for MShot are already supported in hardware TMs as well [65, 60, 90, 25]. Hence, we use the hardware resources in TM to accelerate a memory snapshot of arbitrary lifetime that consist of multiple disjoint memory regions. With a single call to the MShot interface, the system takes an atomic snapshot of a set of regions and isolates it from further memory updates by the application. The snapshot image can be shared by multiple

threads that operate on the snapshot data using normal load/store instructions. The potential uses include, but not limited to, fast concurrent backup, checkpointing, debugging, concurrent garbage collectors, dynamic profilers, fast copy-on-write, and in-memory databases.

We applied memory snapshot to three interesting system software modules: garbage collection, dynamic profiling, and copy-on-write. We build prototypes of these system modules and show that MShot allows us to use additional cores in a CMP to hide their overhead without complex code that manages their interactions with the main application.

This chapter is organized as follows. Section 6.2 provides the use and potential applications of a fast snapshot mechanism. Section 6.3 describes the interface and base design for MShot. Section 6.4 explains how to implement MShot on top of HTM. Section 6.5 explains the use of snapshots in system software modules. Section 6.6 presents the quantitative evaluation and Section 6.7 discusses related work and Section 6.8 concludes the chapter.

6.2 Memory Snapshot

Memory snapshots are an old idea widely used in many useful applications. In this section, we provide their use and potential applications.

6.2.1 Basic Idea

In multi-core and multi-processor systems, concurrent threads are allowed to read and write word-granular data atomically. However, if multiple threads read and write multiple words, systems do not guarantee a consistent view of memory. For applications where consistency is required, a "snapshot" of m memory elements can be created to provide such a view across p processors [8, 2]. Processors are then allowed to execute two types of operations: *update* to write a memory element in the snapshot and *scan* to read memory elements. A snapshot should meet two conditions: (1) any operations on a snapshot are linearizable in a sequential order, and (2) a processor

completes its operations without waiting for other processors to participate.

Memory snapshot is useful to deal with semantically unnecessary contentions to shared data where some threads need to have a recent and consistent view on them while other threads keep updating them. For example, a system may use a memory profiler to analyze the object reference graph of an active application. Since the graph changes continuously as the application makes progress, it is necessary to pause it so that the profiler has an opportunity to process a consistent memory image. Even if the system runs on a CMP, the application and the profiler run sequentially. One could recode both the profiler and the application to insert the synchronization necessary for the two to run in parallel using locks or memory transactions; however, this is particularly difficult and introduces additional runtime overhead for synchronization. What the profiler really needs to see, is not the up-to-date data in memory, but just a consistent view on the object reference graph at recent time. Using a snapshot, the memory profiler can save a recent image of the object reference graph fast and atomically and safely analyze the graph in parallel with the application threads working on the up-to-date image of the graph. There is no need to recode the application or the profiler. Excluding the calls necessary to initiate and terminate a snapshot, the profiler code is the same as when we assumed that it stopped the application before running.

A number of efficient software implementations have been proposed for snapshots [53, 70, 8, 2]. Some proposals use normal memory read and write operations and require O(mp) time for *update* and *scan*[8, 2], where m is the number of data items in the snapshot and p is the number of processors. Update refers to a write operation and scan to a read operation. Others use synchronization primitives available in present CPUs such as compare-and-swap and load-linked/store-conditional to reduce the runtime overhead of *update* to O(1) and *scan* to O(m) [53, 70].

While these proposals provide a consistent view of large data sets, the high runtime overhead prevents snapshots from being used in performance-critical applications. Our proposal seeks to provide fast snapshots in order to speed up and simplify tasks such as parallel and concurrent garbage collection, dynamic profiling, and copy-on-write. Specifically, we aim to provide O(*1*) *update* (as fast as single memory write

operation) and O(p) *scan* in O(m) space. Moreover, we want to avoid the explicit use of synchronization in application code.

6.2.2 Applications

The potential applications of snapshot include, but not limited to, fast concurrent backup, checkpointing, debugging parallel programs, concurrent garbage collection, dynamic profilers, fast copy-on-write, and in-memory databases [8, 70]. We briefly summarize the use of snapshots in the following cases in the following use cases.

Garbage Collection (GC) plays a key role in automated memory management. There has been significant research on efficient GC algorithms [156, 66, 47, 46, 16, 43]. Stop-the-world GC is easy to develop and maintain but pauses the application during collection. Parallel GC reduces the pause time by using additional cores during collection [156]. Nevertheless, pauses are still required, which can be a problem for outline servers with guaranteed response time and QoS. Parallel and concurrent GC removes the pause time by running mutator and collector threads concurrently [66, 47, 46]. However, it is not deployed widely as it is difficult to develop and maintain the complex code that safely separates mutators from collectors.

Snapshot makes it easy to develop parallel and concurrent GC. Collectors (GC threads) take a memory snapshot at the beginning of collection and work on the recent object reference graph in the snapshot image. They share the snapshot image and keep running on spare cores to collect garbage concurrently and continuously, while application threads continue to operate on main memory.

Dynamic profilers run during program execution to tune application performance. Call path profilers [51, 57] provide useful information on hot execution paths but add run-time overhead by either inserting profiling code in function prologues and epilogues [57] or by stalling threads to walk the stack [51]. Memory profilers are widely used to analyze memory usage and find leaks, but cause slow down the application while traversing memory [97, 98].

Snapshot aids call path profilers by providing a private copy of the stack for analysis. Parallel and concurrent memory profilers are supported by allowing multiple

profiler threads to traverse a single snapshot image.

Copy-on-write (COW) has been widely used for efficient data management. For example, it enables *fast fork()* by providing shared virtual to physical page mappings [137] and disk block sharing between virtual machines [146]. COW allows processes or thread to share data until one attempts a write. At that point, COW saves the old data by making a new copy before applying the update. The advantage of COW is the reduced cache and memory pressure as sharing is maximized. The disadvantage of COW is that it must block the update while data is actually copied. The delay gets longer when the operation involves I/O and is done at large granularities, such as pages. Snapshot enables COW without the need to delay the update as it maintains the old value of the data that can be copied in the background.

In-memory database systems achieve good performance by loading the whole data schema into main memory [52, 107]. They are not only attractive for server-side database systems but also useful for embedded systems such as cell phones that support many features with a limited storage system. Such systems already use software snapshots to generate reports on data usage, replicate the data in a database cluster, and support the isolation-level necessary for database transactions. Hardware-assisted snapshots can significantly improve the performance of such tasks.

6.3 MShot Specification

MShot's goal is to provide hardware-assisted memory snapshot for large datasets. System software mudules can use snapshots to improve concurrency without the need for significant code changes in their code or any applications they are applied to.

In this section, we present the MShot design objectives, its software interface, and a stand-alone implementation.

6.3.1 Design Objectives

There are three major objectives for MShot: usability, performance, and cost- effectiveness. We take the following approach to accomplish these objectives:

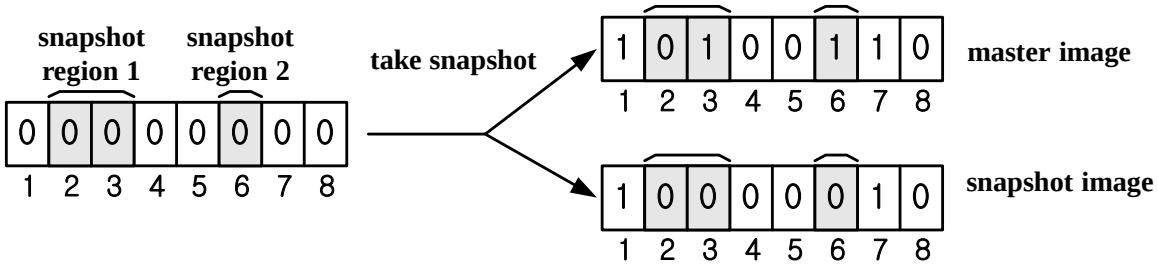


Figure 6.1: An example of using memory snapshot. A snapshot is taken on two regions. The updates to the snapshot regions are applied only to the master image.

- **Versatile interface for usability**: MShot provides a programming tool. It should be easy to use and versatile enough to cover practical applications, including those suggested in Section 6.2.2. This leads to an interface that differs from the one used in traditional software snapshot formulations (scan and update) [8]. We present the simple interface for MShot in Section 6.3.2.

- **Hardware acceleration for performance**: Since MShot is used for performance improvements, it should be fast. Otherwise, its overhead will cancel out any benefits from improved concurrency. Specifically, we target O(1) update time—as fast as a single memory write operation—and O(p) scan time (p is the number of cores). For this purpose, we use additional hardware resources such as per cache-line metadata bits to accelerate data versioning (see Section 6.3.3).

- **Cost-effective implementation on HTM**: To be practical, MShot should be cost-effective to implement. Toward this goal, Section 6.4 shows to implement MShot on top of an HTM system so that hardware resources are shared between the two systems, making them both more cost-effective.

6.3.2 Definition and Interface

MShot provides a snapshot of the memory image at a certain point in time. Multiple disjoint address regions can be part of a snapshot. Once a snapshot is taken on the regions (i.e., equivalent to a scan operation of software snapshots), MShot maintains two memory images for the regions: a master image with the up-to-date data and the

read-only snapshot image. To read from the snapshot image, threads first join the snapshot. Any number of threads can join a single snapshot. Until the user threads leave the snapshot, normal read operations to the snapshot regions return snapshotted data. The separation of taking and joining a snapshot makes it possible to snapshot in a performance critical section and analyze the image later using additional cores. Figure 6.1 shows a simple example of a snapshot with two regions involved, one that covers elements 2 and 3 and one that covers just the 6th element. After the snapshot is taken, update operations are performed on elements 1, 3, 6, and 7. Updates to the elements not covered by the snapshot (elements 1 and 7) are applied to both images. Updates to the snapshot regions (elements 2 and 6) are applied only to the master image.

An MShot image is read-only like software snapshots [8, 2, 70]. User threads are not allowed to write to the snapshot regions. If they do, the system generates a write-protection exception (e.g., SIG_SEGV). Other threads that do not join the snapshot are allowed to write to the snapshot regions (i.e., equivalent to the update operation in software snapshots) and update the master image. Their read operations to the snapshot region return up-to-date data from the master image. In other words, MShot is transparent to any thread or application code that did not explicitly use its services.

MShot also provides an image of the register values in cores at the moment the snapshot is taken. This is useful for analyzing the image later on. There can be multiple snapshots active at any moment. To simplify hardware, MShot does not allow overlapping snapshot regions. Hence, the system must maintain only two versions per address: the up-to-date version and the snapshot version. In the same spirit, MShot supports cache-line granularity in specifying regions to avoid the additional hardware resources required for per-word version management. These design choices did not complicate the applications we studied.

Table 6.1 shows the operations supported by MShot. There are two groups of operations: one to control the snapshot lifetime and the other to access the snapshot. A snapshot is taken by *take_snapshot()* using a simple data structure called *snapshot_info* shown in Figure 6.2. The *snapshot_regions* field points to an array of

Group	Method	Function
Snapshot Control	take_snapshot (snapshot_info*)	Take a snapshot on the address regions specified by snapshot_info.snapshot_regions. New snapshot id (SID) is set at snapshot_info.SID. Saved register values are pointed to by snapshot_info.saved_regs.
	destroy_snapshot (SID)	Destroy the snapshot of SID.
Snapshot Sharing	join_snapshot (SID)	Start using the snapshot of SID.
	leave_snapshot (SID)	Stop using the snapshot of SID.

Table 6.1: MShot interface.

```
snapshot_info {
  short SID;                          // unique snapshot id.
  double[][2]* snapshot_regions;      // array of (start address,
                                      // end address) pairs.
  void* saved_regs;                   // saved register values.
  short TID;                          // optional, Thread id
                                      // to isolate selectively.
}
```

Figure 6.2: `snapshot_info` data structure used for take_snapshot().

pairs of virtual addresses that demarcate the beginning and the end of an address region belonging to the snapshot. The addresses should be aligned to cache lines. Once a programmer sets *snapshot_regions* and invokes *take_snapshot()*, MShot takes a snapshot of the address regions and returns a unique snapshot ID through the *SID* field. Register values are saved in a system-dependent data structure and pointed to by *saved_regs*. In our prototype, we used the *ucontext*, originally for use with OS signal handlers. If the call fails due to any reason, SID is set to 0. Note this reserves SID 0 for an invalid snapshot. The thread ID, *TID*, field is optional and can be used for optimization when the address regions of the snapshot are private to a thread (e.g., stack and thread local storage). For these regions, MShot does not need to interact with other threads. For single-threaded programs, this parameter is ignored since there is just one thread. If programmers are not sure if a snapshot region is thread-local or not, they set TID to -1 to disable the optimization.

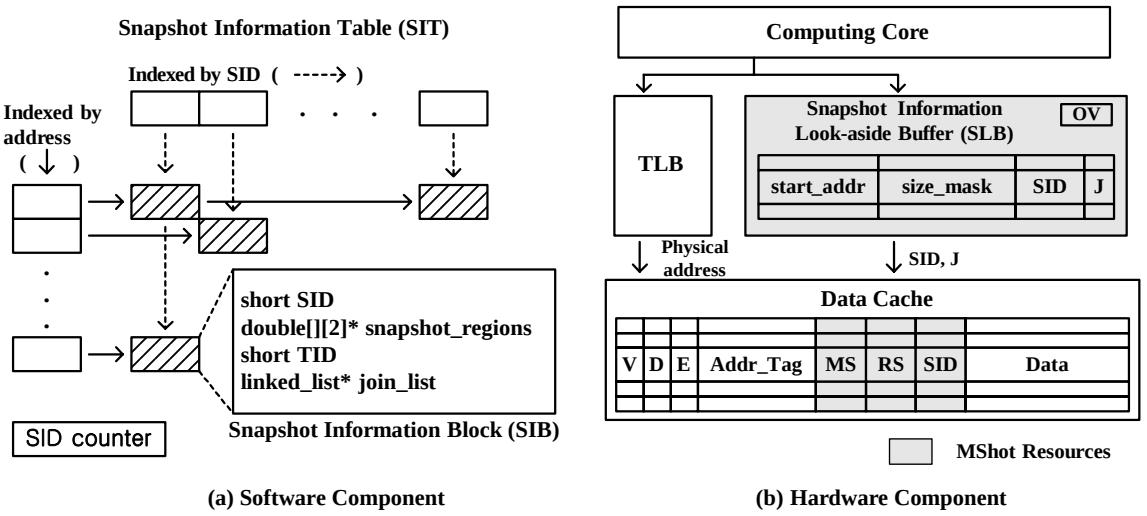


Figure 6.3: MShot hardware and software structures.

Threads can use the snapshot by joining it with *join_snapshot()* and leave it by calling *leave_snapshot()*. The snapshot is destroyed using *destroy_snapshot()*. Destruction is allowed only after all user threads have left the snapshot. MShot checks this condition and returns an error to prevent threads from using a destroyed snapshot.

6.3.3 Base Design

Figure 6.3 shows the base MShot design, including both software and hardware structures. There are three key components: the Snapshot Information Table, the Snapshot information Look-aside Buffer, and Snapshot Metadata Bits.

Snapshot State Management

The **Snapshot Information Table (SIT)** shown in Figure 6.3-(a) is a doubly-indexed hash table for managing all snapshot information. This software structure is indexed either by SID or virtual address. Entries are called *Snapshot Information Blocks (SIB)*, which are similar to the *snapshot_info* structures passed to *take_snapshot()*. However, they do not contain the *saved_regs* field. Instead, they

add a new field called *join_list*, a linked list with the threads that joined a snapshot. MShot uses a simple counter to generate a unique SID.

The **Snapshot information Look-aside Buffer (SLB)** shown in Figure 6.3-(b) is a small, fully-associative, hardware cache for accelerating the retrieval of snapshot information. In our evaluation, we use an SLB with 64 entries. The SLB helps each core identify if a load or store goes to an address included in a snapshot. Snapshot regions are typically large consecutive address ranges. To exploit this fact, we organize SLB entries so that the address tag consists of two fields. The first field is the start virtual address of a snapshot region and the second field is the bit mask that presents the size of the region. The size is a power of two and the bit pattern for the field is obtained by inverting the binary expression of (size - 1). An entry can cover a whole contiguous virtual address space (e.g. 4 Giga-bytes with an entry). The SLB is accessed in parallel with the TLB. A matching SLB entry returns two fields: the SID to which the snapshot region belongs and the *J (join)* bit indicating if the current thread has joined the snapshot. The SLB maintains an overflow *OV* bit to indicate there is an SLB entry that was evicted at some point in time due to capacity issues. If the bit is not set, an SLB miss is ignored—there is no snapshot region associated with the memory address. If the bit is set, the miss is handled by a software refill handler that accesses the SIT. To support the software handler, two SLB instructions are added: *SLBI* to invalidate SLB entries and *SLBLD* to load them. SLBI and SLBLD are similar to TLBI and TLDLD in PowerPC [112].

Two Snapshot Metadata bits and SID bits are added per cache line for data versioning. The *MS (Modified since Snapshot)* bit is set when a cache line in a snapshot region has been modified since the snapshot. This renames the line and separates the up-to-date data in the master image from the old data in the snapshot image. The *RS (Read from Snapshot)* bit is set when a cache line in a snapshot region is refilled with an old data version from the snapshot. Such lines should be invalidated when the snapshot is destroyed. If both bits are reset for a cache line, it implies that the master image and the snapshot image have the same data for the line. SID is set when either the MS bit or RS bit is set to indicate the snapshot for which the bits are set. Both bits cannot be set at the same time. Note that this section assumes that

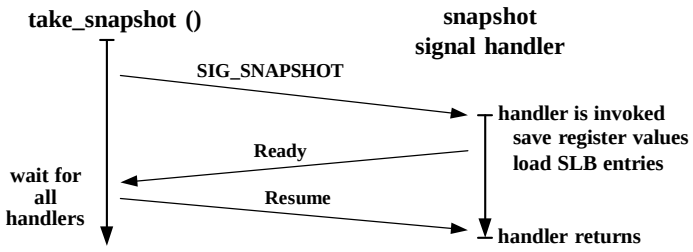


Figure 6.4: Three-way handshaking for snapshot initialization.

the data cache will never overflow snapshot information due to capacity issues. We explain how to handle overflows using HTM virtualization mechanisms in Section 6.4.

Snapshot Operations

take_snapshot() initiates the process of taking a snapshot. To make it thread-safe, we used a simple global lock. Snapshot regions identified in the *snapshot_regions* field of the *snapshot_info* argument are compared with the regions of existing snapshots in SIT to check for overlap. If there is no overlap, the current SID counter value is assigned to the snapshot and the counter is incremented. An SIB is created for the snapshot by copying *snapshot_info* to the block. The *join_list* of the block is initially empty. The block is inserted into the SIT and the addresses of the *snapshot_regions* are loaded into the SLB. A snapshot region can be mapped to multiple SLB entries depending on the size and alignment of the region. Memory allocators or compilers can help adjust the alignment and size of snapshot regions to save SLB entries at the cost of higher virtual address space consumption.

Three-way handshaking is used to build the snapshot gradually with O(p) scan time. While copying is not performed when take_snapshot() is called, MShot must ensure that all cores are aware of the snapshot information before the call returns. This ensures any write to the snapshot after the snapshot is taken triggers data versioning. The snapshot information is exchanged using the three-way handshake shown in Figure 6.4. First, snapshot request messages are sent via inter-core signals to the other threads to invoke snapshot signal handlers. The handlers copy the saved register values of the threads to a pre-allocated data structure pointed to by

	Write to snapshot	**Read from snapshot**
Snapshot Threads	Illegal operation.	Hit if address tag matches and MS bit is not set. Set RS bit and SID when refilled with MS bit piggy-backed.
Other Thread	Hit if address tag matches and RS bit is not set. Set MS bit and SID when hit or refilled.	Hit if address tag matches and RS bit is not set.

Table 6.2: Memory operations with MShot.

the *saved_regs* field of the *snapshot_info* argument and load the snapshot information into the SLB. Next, the handlers send response messages back to the thread that initiated the snapshot and wait for resume messages. On receiving the response messages from all the handlers, the initiating thread sends a resume message that terminates the handlers and allows application threads to resume. This process has O(p) complexity. Note that the scan time in the previous algorithms for software snapshot is O(m) (number of memory elements), since they also provide the wait-free property: processors are not stalled even if messages are lost. This is desirable if snapshots are used to help with fault-tolerance. The base MShot design does not guarantee wait-freedom as it targets primarily concurrency. There are several orthogonal mechanisms to ensure recovery from lost messages or other faults [4, 138]

Local synchronization is performed instead of the global synchronization described above if the TID field of the *snapshot_info* is not -1. The snapshot information does not need to be propagated to the threads other than the thread of the TID, since the memory regions covered by the snapshot are thread-private. If the thread is running, the same three-way handshaking protocol is used but only with one thread. If the thread is not running, the protocol is not used. We simply update the SLB with the snapshot information when the thread is scheduled again.

join_snapshot() works by updating the SIB to remember that the thread is a new *user* of the snapshot. The corresponding SLB entries are reloaded to set the J bits for the thread.

Cache operations with the snapshot metadata bits are summarized in Table 6.2. If a user thread writes to a snapshot, an exception is triggered since the user thread is only allowed to read from the snapshot. The SLB detects this case. A read from the snapshot is a hit if there is a cache line with a matching address tag and its MS bit is 0. This indicates that the line has not been modified since the snapshot was taken. If it is a miss, the SID bits and J bit are piggy-backed to the refill request to indicate that the refill should come from the snapshot image. Receiving the refill request, other caches search for a cache line with matching address tag whose MS bit is 0. If the MS bit of the cache line with matching address is 1, the MS bit is attached to the response message to notify the requester that the master image has deviated from the snapshot for that address. Finding no matching cache line in other caches, the request is sent to main memory and the data is refilled. Because we assumed no cache overflows, this happens only when the master image and the snapshot image have the same data for the address. Hence, there is no need to modify memory controllers. Refilling the cache line, we set the RS bit if the MS bit is piggy-backed. The RS bit is used to when destroying the snapshot to see if the line is to be invalidated.

A read from or a write to the snapshot by a thread not using the snapshot (i.e., J bit is 0) is a hit if there is a cache line with a matching address tag and RS is reset. This indicates that the line does not contain the old data from a snapshot. If it is a write hit, the MS bit is set to indicate that the line no longer belongs to the snapshot image. If the write misses, the MS bit is set after the line is refilled. Note that the function of the MS bit is to rename the up-to-date data in the master image in order to distinguish them from the old data in the snapshot image. This renaming is accomplished system-wide by simply setting the MS bit since the line is exclusive to that processor. If the line is dirty, it is flushed to memory first not to lose the old data. This makes updates O(1). A cache miss is handled similar to the case of a thread that has joined the snapshot. The SID and J bits are piggy-backed on cache line refill requests. The bits are used by the other caches to find a cache line whose address tag matches and whose RS bit is 0.

leave_snapshot() is called to stop using the snapshot. The ID of the thread is removed from the SIT and the SLB entries are reloaded to clear the corresponding J

	HTM resource	**Usage in MShot**
Data Versioning	Transactional metadata bits per cache line	Snapshot metadata bits per cache line
	Transaction ID (TID) per cache line	Snapshot ID (SID) per cache line
	Transactional metadata bit gang clear logic	Snapshot metadata bit gang clear logic
Virtualization	Shadow page table in SW, table access logic in HW	Providing access to master/snapshot images
	Home/shadow pages	Storage for snapshot data that overflow the cache

Table 6.3: Hardware resource mapping between HTM and MShot.

bits.

destroy_snapshot() destroys the snapshot. It starts with invalidating the SLB entries. Cache lines with the SID of the snapshot are invalidated if the RS bit is set. Then all metadata bits of the snapshot are gang-cleared. It completes with removing the SIB of the snapshot from the SIT.

Note that, even though we present one design point in this section, it is obviously possible to lower the cost by exploring other design points such as page-granular snapshot and limiting accessibility to snapshot to a single thread at a time. We choose to maintain full functionality and reduce hardware cost by integrating with HTM.

6.4 MShot on Hardware Transactional Memory

HTM systems use dedicated hardware resources to accelerate transactional execution. In this section, we propose a scheme to lower the hardware cost of MShot dramatically by sharing the hardware resources for HTM.

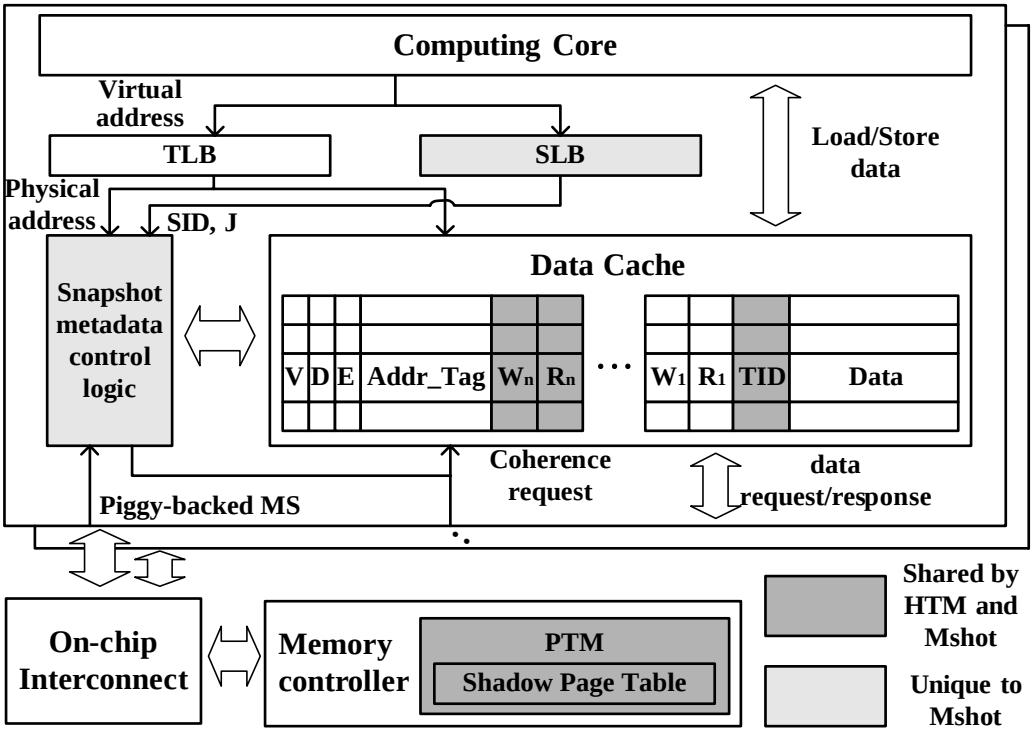


Figure 6.5: Resource sharing between HTM and MShot.

6.4.1 Resource Sharing between MShot and HTM

To a TM system, a memory snapshot looks like a read-only transaction that never aborts. The versioning mechanism for snapshot is similar to that for HTM as it also uses per cache-line metadata bits. Most HTMs use a pair of metadata bits to record the addresses written or read by a transaction. Additional pairs of metadata bits are also available to support composable nested transactions [84]. MShot uses one of these additional pairs to implement its versioning mechanism. The remaining pairs are used for nested transactions as usual. A cache line with transactional or MShot metadata may be evicted due to a cache capacity overflow. We can reuse a TM virtualization mechanism to also handle MShot overflows by pretending that an snapshot overflow is a read-only transaction that experiences cache capacity issues. No changes are necessary in the virtualization mechanism to support this case.

In our evaluation, we use the Page-based TM (PTM) which uses additional hardware to ensure that overflow processing is fast [29]. PTM maintains in hardware

a shadow page table when transactional data overflow hardware caches. When a cache-line with transactional data or metadata is evicted, PTM allocates a new page (shadow page) to store the last committed version of the data and uses the old page to store the overflowed data. The shadow page table maintains the proper mapping information including a per page write summary vector that indicates which blocks in the page have overflowed. PTM refers to the vector to determine which blocks it serves a memory request with.

Table 6.3 summarizes the resource sharing between HTM and MShot. HTM uses metadata bits to record transactional memory accesses: W bit for data in the write-set and R bit for data in the read-set of a transaction. Essentially, the W bit is used to rename an address and distinguish a new value from the last committed one. For MShot, we map the MS bit to the W bit and use it to distinguish the master image from the snapshot. Similarly, we map the RS bit to the R bit of the HTM system to mark old data from the snapshot image that should be invalidated when the snapshot is destroyed. MShot uses the W and R bit pair for the highest nesting level transaction as shown in Figure 6.5. This reduces the number of nesting level the underlying HTM supports by one. When used by MShot, the highest nesting level should not generate any conflicts. We can filter such conflicts either in software (using a violation handler) or in hardware (by modifying the conflict detection logic). It checks the bits against the accesses from the local core in addition to snooped memory requests from the remote cores. PTM includes transaction ID in each cache line to allow transactions from many threads to share the cache [29]. MShot uses these bits for the SID to allow multiple snapshots to share the cache. HTMs have gang-clear logic to reset the metadata bits. MShot uses the same logic to reset the metadata bits at the end of snapshot.

On an overflow, PTM uses the original physical page (home page) to store the new version of data and an additional physical page (shadow page) to buffer the last committed version. A shadow page table maintains the metadata and provides access to both pages. MShot uses the home page to store the master image and the shadow page to buffer the snapshot image. When a cache line with an MS bit is evicted, the MS bit and SID of the line are piggy-backed. PTM uses the SID as the transaction

ID and the MS bit as the W bit. The snapshot data of the cache line is copied to a shadow page and the evicted up-to-date data of master image goes to the home page. MShot evicts cache lines with the RS bit set silently since the RS bit indicates that the line should be invalidated anyway.

To reload an overflowed cache line, the hardware attaches the J bit and SID bits with the request. PTM hardware uses the J bit to select if the home page or the shadow page is read. If the bit is 1, the shadow page is read. If not, the home page is read. PTM releases the shadow pages at the end of an overflowed transaction. At the end of a snapshot, MShot sends out the transaction commit message with the SID of the snapshot pretending that it is the end of a read-only transaction. PTM releases the shadow pages with the snapshot image. As is the case with metadata in hardware caches, PTM must ignore any conflict detected for the read-only transactions of MShot. The distinction can be made easily by checking if the transaction ID starts with a special bit prefix that MShot adds for this purpose.

Of course, there are hardware resources required for MShot that are not available in HTM. They include the SLB structure and the additional information MShot must associate with cache coherence messages (SID, J bit, MS bit). The size of the SLB is a tradeoff between cost and performance. The width of coherence messages is increased by two bytes in the system we evaluated. Note that MShot does not require a full-fledged HTM for integration. More specifically, MShot does not require the HTM components such as conflict detection logics, transaction abort logic, and software handler logics.

6.4.2 Running with Transactions

We do not allow take_snapshot() to be called within a transaction to avoid including uncommitted data in a snapshot. The simplest way is to wait for all outstanding transactions to commit, but this can cause a long delay in the presence of long transactions. An alternative is to abort the transactions. This approach is cheap and fine as long as snapshots are not taken too frequently. We use this scheme for the evaluation in Section 6.6.

Transactions may begin and abort within a snapshot lifetime. If they do not abort or overflow, MShot does nothing for transactions since there is no difference for MShot between transactional execution and non-transactional execution in this case. If they abort, the TM system rolls back the modified data and transactional metadata bits for the aborted transactions but does nothing for snapshot metadata bits. A straightforward solution is for MShot to take a checkpoint of snapshot metadata bits whenever transactions start, but it is costly to add extra storage for the checkpointed bits. An alternative is to do nothing as these bits cannot affect correctness. The MS bits set by the aborted transaction can be thought of as being set by idempotent writes. The RS bits set by the transaction just cause extra cache line invalidations.

If a transaction overflows during the lifetime of a snapshot, there are two cases to consider. If the overflowed cache lines are not modified from the snapshot image, there is no snapshot data and metadata to manage. The TM virtualization mechanism deals with the lines regardless of MShot. If the lines are modified, the TM virtualization mechanism maintains three versions: snapshot version, last committed version, and transactional version. It deals with the case as if there are two overflowed transactions. The overflowed transaction is handled as usual. The overflowed snapshot is handled as a transaction whose last committed version is the version of the data in the snapshot. Supporting this scheme is problematic for PTM since it manages only two versions per address. For our evaluated system, MShot detects this case and gives priority to the overflowed transaction so that it is guaranteed not to abort. Hence, PTM can ignore the last committed version, and use its capabilities to buffer the snapshot version and the latest version produced by the transaction. This approach leads to no performance issues if overflows are rare.

Join_snapshot() and leave_snapshot() select the memory image accessed by a thread (master image or snapshot image). If one of the two is called in a transaction that aborts, we call the other one in the abort handler in order to compensate. It is hard to undo destroy_snapshot() since it discards the snapshot data and metadata. If it is called in a transaction, we defer the actual execution of the function until the transaction commits. This is acceptable because destroy_snapshot() is not a timing-critical operation unlike take_snapshot().

6.4.3 System Issues

Just like the TLB, the SLB must be reloaded on context-switches because it is indexed by virtual addresses. The SLB entries do not need to be saved since the SIT is always more up-to-date than SLB. On rescheduling, the SLB can be loaded eagerly or lazily. We use the eager approach in our evaluation.

MShot relies on the TM virtualization mechanism to maintain the overflowed data and metadata over context switches. It also uses the same mechanism to deal with paging the snapshot. Working with PTM, MShot flushes out the page from the cache before paging. PTM pages to disk the home and shadow pages but remembers the relation between the two in order to restore it when the corresponding data are accessed again [29].

I/O operations within transactions are troublesome because they can expose uncommitted data. On the contrary, it is safe to perform I/O operations on snapshot regions. The I/O access is served by the memory image and the snapshot is not affected.

6.4.4 Discussion

MShot can use also any other TM virtualization mechanism to handle cache overflows for snapshot data [31, 120]. MShot uses a TM virtualization mechanism as hardware-acceleration for data storage in virtual memory. The basic requirement is that MShot should be able to use SID and the address as the key to index the data structures in the virtualization scheme.

There has been research on implementing hybrid TM systems that provide hardware acceleration for some critical operations in software TM environments. MShot can reuse the hardware resources in such systems such as the additional metadata bits in each cache line [126]. There are also hybrid TM proposals that use signatures to compress the transactional metadata information. While signatures are cost-effective and performs well for TM, they cause correctness issues if used for MShot due to their imprecise nature. Implementing MShot on top of an software TM would make MShot only as fast as software implementations for memory snapshots.

6.5 Using MShot in System Modules

To evaluate the effectiveness of MShot, we used it to increase the concurrency in three important software modules. Starting with their straight-forward implementations that either stop the world or serialize the accesses to the shared datasets in favor of algorithmic simplicity and easy code management, we added just a few MShot methods to enable them to run on idle cores concurrently with other software modules. And this is the key benefit of MShot to allow for algorithmic simplicity, easy code management, and performance at the same time.

6.5.1 Snapshot-based Garbage Collection

Snapshot GC takes advantage of the fact that "once garbage always garbage." If a garbage object is found in snapshot image, it is garbage in the master image as well. Snapshot GC starts with take_snapshot() to take a snapshot of the regions it wants to collect. Initiating the snapshot introduces a small pause but, beyond this point, mutator threads (application threads) can resume immediately and run in parallel with the collector threads if sufficient cores are available. All collector threads join the snapshot as well. Collector threads work in parallel to build the root-set using the snapshot image and the saved register values. Neither the GC nor the application programmers need to worry about synchronization between collectors and mutators since they work on different images. The snapshot is destroyed when collection is done.

For our evaluation, we implemented snapshot GC on top of Boehm GC [16, 43], a conservative collector for C and C++ programs. We started from a parallel mark-sweep collector code and used snapshot to run its threads concurrently with the mutator. We added less than 100 lines of code to the GC to turn the original code into a concurrent collector. Most of the lines were to prepare snapshot_info data structure. Only four call sites were added for MShot interface. The availability of the snapshot allowed us to ignore synchronization issues between collector and mutator threads. Nevertheless, attention was given to the allocation of GC metadata, such as the data structure for free blocks. Boehm GC uses the same heap for the application

and the GC itself. The GC collects garbage objects for both at the same time. While this seems desirable, it complicates setting up the snapshot regions because the GC metadata objects need to be updated during collection. Hence, we cannot simply take a snapshot of the whole heap. To address the problem, we reserved a separate memory space for GC metadata. We did not do anything special to garbage collect TM programs. Note that the snapshot GC misses garbage created after the snapshot is taken. Assuming spare cores, the GC can run frequently enough so that it claims the missed objects the next time it is invoked.

6.5.2 Snapshot-based Profiler

The snapshot call-path profiler periodically walks the stack to obtain information on the calling relationship between functions in the application code. The process starts by taking a snapshot of the stack of the thread to be analyzed using take_snapshot(). The TID field of the snapshot_info argument is set to the ID of the thread to let MShot know that the snapshot region is thread-private. Then, the SID and the saved register values are passed to a profiling thread through a stack analysis request queue. The profiling thread joins the snapshot and analyzes it in parallel with the scheduled application thread. It obtains the stack pointer and frame pointer from the saved register values and starts walking the stack following the sequence of return addresses. The return addresses found are used as keys to look up the symbol table of the analyzed program to locate the corresponding function names. Adding the snapshot calls to the profiler was trivial and no special synchronization code was needed in the function that calls the stack.

The snapshot memory profiler takes a snapshot on memory to find classes and instances of classes and to analyze relations among objects. For our evaluation, we implemented a snapshot memory profiler by modifying Boehm GC. Basically, the profiler works in the same way that snapshot GC does by traversing live objects in a snapshot image during the mark phase. On finding a live object, it records the type of the object instead of setting a live bit for it. We changed the memory allocator so that objects in C have type information.

6.5.3 Snapshot-On-Write (SOW)

Snapshot makes copy-on-write (COW) seamless. COW makes a copy of the shared data when a data update is detected. For correctness, it blocks the write until the copy operation completes. With SOW, instead of blocking the write, a memory snapshot is taken on the shared data using take_snapshot(). The TID is set to that of the thread that is making the update. The write operation can complete right after the snapshot is taken. The snapshot is held until the copy operation completes and is destroyed by a copy helper thread. This technique is beneficial for use cases of COW including pages sharing in fast fork and disk block buffer sharing in virtual machines [146].

We developed a microbenchmark to emulate fast fork with SOW and evaluate its performance. It launches an application with write protection on the heap, data, and bss segments. Receiving a SIGSEGV, a signal handler takes a snapshot on the write-protected page instead of copying the page right away. The page copy request is sent through a queue to a copy helper thread running on another core. The write protection on the page is removed and the application resumes without waiting for the page to be copied.

6.6 Evaluation

6.6.1 System and Applications

We implemented MShot on top of an HTM system as explained in Section 6.4. The base HTM detects conflicts eagerly based on the MESI protocol [90]. It uses eager data versioning but defers logging of old data until transactional data overflows the cache. As long as transactions fit in the cache, it commits fast by gang-clearing the transactional bits and rolls back fast by invalidating the cache lines modified by the transaction [60]. When transactional data overflow the cache, the log entries are saved in a thread-private region. The HTM has multiple pairs of transactional metadata bits per cache line to support nested transactions and uses PTM for virtualization [29]. PTM is in an on-chip directory controller located next to shared L2 cache. A pair of W and R bits per cache line is dedicated to MShot. The snapshot metadata control

Feature	Description
CPU	2GHz, single-issue, in-order x86 core
SLB	64 entries
L1 Cache	64 KB, 4-way, 32B line, MESI, write-back 1 cycle hit time, private
L2 Cache	8 MB, 8-way, 32B block, MESI, write back 10 cycle latency, shared, 8 banks bit vector of sharers
Memory	4 GB, 100 cycle latency
Interconnect	Tiled network, 32B links, 3 cycles per hop

Table 6.4: Parameters for the simulated CMP system.

logic is augmented to the L1 cache. The SLB has 64 entries and is placed next to TLB. We use 14-bit SIDs. Hence, attaching the SID, J bit, and MS bits on coherence requests introduces 2 additional bytes. We implemented a library to provide the MShot interface as explained in Section 6.3.2. If take_snapshot() is called within a transaction, we roll back all active transactions to ensure that the snapshot does not include uncommitted data.

Table 6.4 shows the evaluation environment. We used a detailed simulator for x86 CMPs with 16 cores. The number of cores used by applications varies depending on the test and the application. Each core has a 64KB private L1 cache with transactional metadata bits. The cores are interconnected using a tiled network with 3-cycle link delay per hop. The CMP includes an 8-MByte shared L2 cache and an on-chip directory.

We used 12 applications and one micro benchmark. W3M is a client-side web browser [148]. Pypy is a python interpreter [116]. Gzip is a compression tool [58]. Mpeg2 is a MPEG-2 decoder [12]. Cfrac performs continuous fraction factorization for integers [21]. Nullhttpd is an HTTPD web server [105] . Vacation mimics an e-commerce system and spends more than 90% of execution time in transactions [23]. Vacation-L is a locking version of Vacation. Mp3d and Radix are from SPLASH2 [155]. Tomcatv, Equake, and Swim are from SPEComp [141]. RBtree is a micro benchmark that adds, searches, and deletes objects into and from a red-black tree in transactions.

	Null httpd	RB tree	Vacat tionL	Vaca tion	Cfrac	Gzip	Mpeg2	Pypy	W3M
Snapshot Data	5.25	0.02	0.35	0.10	0.00	0.59	6.46	0.29	0.79
Shadow Page	5.38	0.28	0.40	0.77	0.04	0.64	6.52	0.38	0.97

Table 6.5: Memory requirement to manage overflowed snapshot data in Mbyte.

We set up the experiments to evaluate MShot for garbage collection, call-path profiling, memory profiling, and copy-on-write. Each experiment compares the two software implementations for the same component: non-concurrent implementations without snapshot and snapshot-based version. The non-concurrent versions stop the application while the system software component runs. We use a parallel GC, parallel memory profiler, single-threaded call graph profiler, and single-threaded copy-on-write handler for the non-concurrent case. The snapshot-based versions run the components concurrently with the application using additional cores in the CMP system. We observe how efficiently they use these cores to support the system service with minimal overhead to the application runtime. The ultimate goal is to use MShot and the additional cores to give applications the illusion of using an ideal system with zero overhead for GC, profiling, and COW.

We run Vacation-L, Nullhttpd, RBtree, and Vacation in parallel with 8 cores and provide two additional cores for the snapshot-based components. We run the other applications with one core and provide one additional core for the system components. To provide a fair comparison, when the non-concurrent components run they use all available cores to accelerate parallel GC and parallel profiling. The code for call-path profiling and copy-on-write is single-threaded.

6.6.2 Garbage Collection Tests

We used 8 applications and the microbenchmark to compare parallel GC and the snapshot-based one. To observe meaningful behavior within a reasonable simulation time, a 32MB heap is used. While this is smaller than what a real environment would

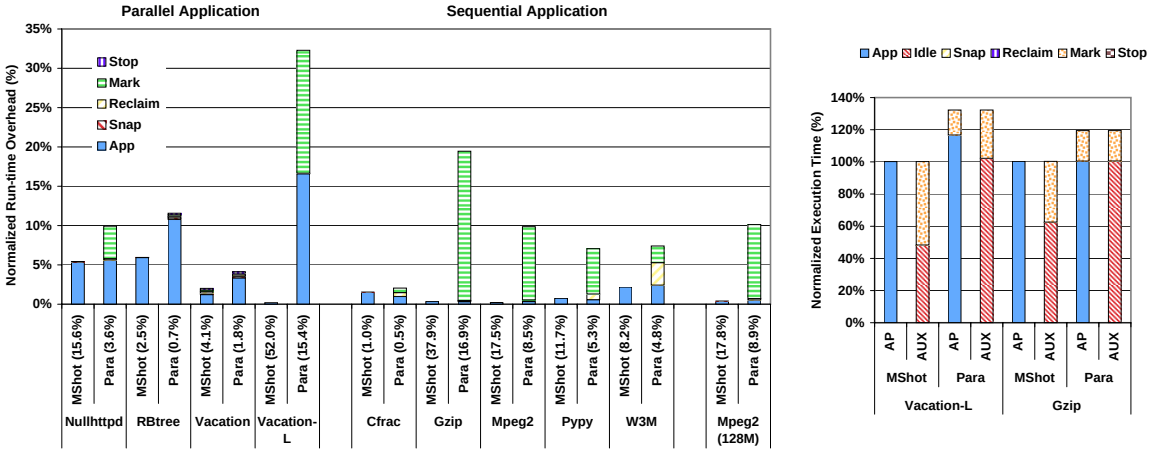


Figure 6.6: The graph on the left shows the run-time overhead caused by parallel GC (Para) and snapshot GC (MShot). It is normalized to application execution time without GC. The graph on the right shows the total execution time of Vacation-L and Gzip. The AP bars are for the cores running applications. The AUX bars are for the auxiliary cores used for garbage collection. Parallel applications runs with 8 cores. Parallel GC stops the applications and uses 10 cores for the mutators. Snapshot GC concurrently runs with 2 cores. Sequential applications run with 1 core. Parallel GC stops the sequential applications and uses 2 cores for collection. Snapshot GC runs concurrency with the sequential applications using on 1 core.

use, the applications still exhibit reasonable ratios of GC time over the total execution time (from 1% for Cfrac to 18% for Gzip). We also show results for running MPEG2 with a 128MB heap.

Figure 6.6 presents the run-time overhead added to the application execution time normalized to the execution time when running without GC using a very large heap. Lower overhead is better. In each bar, *Stop* is the time spent for stopping the system to start GC, *Mark* the time for the mark phase, *Reclaim* for the reclaim phase, and *Snapshot* the time to initiate a snapshot. *App time* is the time spent in the application itself. The MShot bar represents snapshot-based GC while the Para bar represents the original parallel GC (non-concurrent). The numbers in parenthesis is the utilization of the additional cores for each case. 10% utilization means that the extra cores run GC for 10% of the total execution time and are idle the rest of time. Note that, compared to the parallel GC, software-only concurrent GC typically

shows shorter pause time at the cost of throughput loss since it has to deal with synchronization between application threads and GC threads using software barriers for load and store accesses [55]. The measured performance of parallel GC can be thought of as a loose upper bound of software-only concurrent GC.

For all applications except Nullhttpd and RBtree, snapshot GC eliminates most of the run-time overhead experienced with parallel GC. Nullhttpd and RBtree show an increase of the application execution time itself. For Nullhttpd, this is because of its significant memory requirements in order to load up new data to create HTML files for HTTP responses. In the snapshot GC case, we observe increased contention for the L2 cache between the GC and the application itself. For RBtree, it is due to conflicts between transactions allocating memory and the GC around some GC variables. The TM contention manager assigns a higher priority to GC, which makes application transactions abort more often. Vacation, the other TM application, does not show this effect since it has longer transactions than RBtree and the application itself has a high conflict ratio. On average, snapshot GC reduces the overhead down to 1.5%, which makes garbage collection essentially cost-free. Snapshot GC scales well with a 128MB heap for Mpeg2. Take_snapshot() adds negligible overhead to applications and snapshot GC shows better utilization of the auxiliary cores than parallel GC in all cases.

To further investigate the effects of garbage collection with and without snapshot, we used Vacation-L and Gzip to measure the time breakdown for both the cores that run application threads and the cores that run user threads. Figure 6.6 (right graph) shows the breakdown normalized to the application execution time without GC. The AP (application) bar is the one for the cores running application threads, and the AUX (auxiliary) bar is for the additional cores running GC threads. For both applications, snapshot moves the GC related cycles from the AP bar to the AUX bar, allowing the application cores to process application threads faster. It is interesting to note that the App time of Vacation-L has increased with parallel GC on the application core (i.e., higher AP bar). This is because alternating between Vacation-L threads and parallel GC threads in the non-concurrent case leads to thrashing in the L1 cache. This is not the case for the concurrent case using snapshots, as the GC threads share

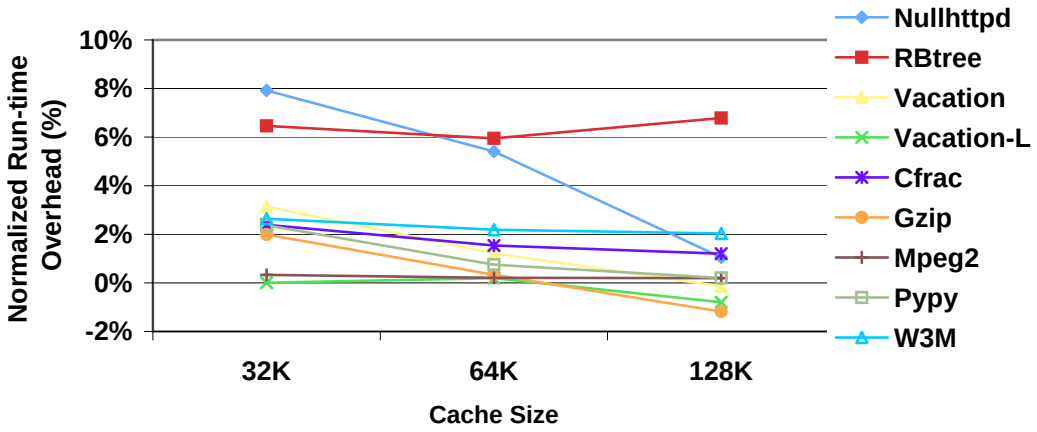


Figure 6.7: The sensitivity of snapshot GC to different L1 cache sizes. It is normalized to the application execution time with a 64K L1 cache and without GC.

only the L2 with the application threads.

Table 6.5 shows the memory requirements to maintain the overflowed data for the snapshot-based GC. Since we use PTM for TM virtualization, additional physical pages are required for shadow pages even if a single cache line overflows within each page. In the table, Snapshot Data is for the actual snapshot data at cache line granularity and Shadow Page is for the number of shadow pages times the 4-KByte page size. For all applications except Nullhttpd and Mpeg2, the memory requirements for snapshot data is under 1 MBytes. The worse case is still under 7 MBytes for the 16-core CMP.

We also examined the sensitivity of snapshot GC to L1 cache size, varying it from 32KB to 64KB and 128KB. Figure 6.7 shows the run-time overhead added to application execution time. It is normalized to the total execution time of the applications running with a 64KB L1 cache and without the GC. While the applications show better performance with a bigger L1 cache in general, the variance is under 4% for all cases except for Nullhttpd. The higher sensitivity for Nullhttpd is because it is cache-bound when generating HTML pages.

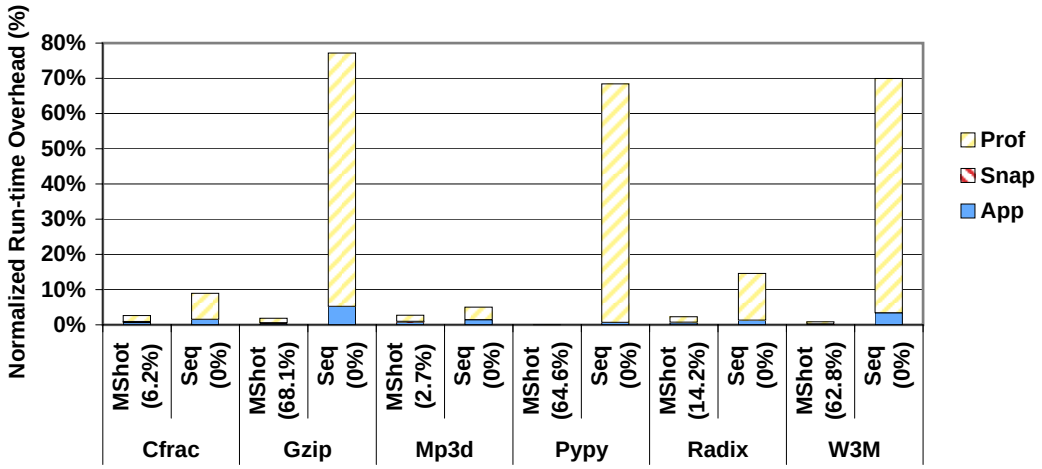


Figure 6.8: The run-time overhead caused by sequential call path profiler (Seq) and snapshot call path profiler (MShot). It is normalized to application execution time without profiling. For the sequential call path profiler, we use 1 core for both application and profiling. For the snapshot call path profiler, we concurrently use 1 core for the application and 1 core for profiling.

6.6.3 Profiler Tests

The profiler evaluations consist of call-path profiling and memory profiling. Figure 6.8 shows the overhead added due to call-path profiling. It is normalized to the application execution time without profiling. *Prof* is the time for running the profiler, *App* for the execution of the application itself, and *Snap* the time to take a snapshot. The percentage in the parenthesis is the utilization of the auxiliary cores. The non-concurrent implementation runs on the same core as the application and does not use the additional core. The profiler is triggered 50K cycles after the last profiling has completed to guarantee reasonable forward progress of the applications with the non-concurrent implementation. Gzip, Pypy, and W3M experience the largest performance improvements with snapshot profiling ranging from 68% (Pypy) to 75% (Gzip). This result is important for the programs written in object-oriented language since they typically have deeper call depth. This is due to the deep call graph (average depth of 14) for these applications. On the contrary, Mp3d, Radix, and Cfrac have an average depth of 4 to 6 functions. Regardless of the applications' call depth,

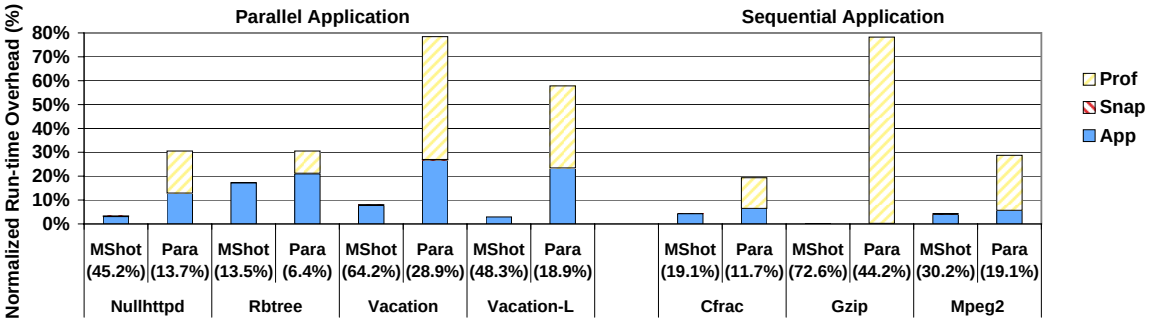


Figure 6.9: The run-time overhead caused by the parallel memory profiler (Para) and the snapshot memory profiler (MShot). It is normalized to application execution time without profiling. Parallel applications run with 8 cores. The parallel memory profiler stops the applications and run with 10 cores. The snapshot memory profiler runs concurrently with 2 cores. Sequential applications run with 1 core. The parallel memory profiler stops the applications and run with 2 cores. The snapshot memory profiler run concurrency with 1 cores.

the snapshot call-path profiler adds negligible overhead to the application (less than 3%). Note that the profiling period we used here is shorter than a typical call-path profiling period. In a 2GHz system, if the profiler is invoked every 1msec, it then it runs every 2,000,000 cycles. This is 40× less often than the the frequency we used in our tests. Still, we were able to maintain low profiling overhead which implies that, given spare cores, profilers can run more often to provide more accurate information.

Figure 6.9 shows the overhead due to the memory profiler. It is normalized to the application execution time without profiling. The profiler run every 10M cycles. The snapshot memory profiler reduces the run-time overhead by 12% (RBtree) to 77% (Gzip). In this test, the App time itself has increased noticeably with both the parallel memory profiler and the snapshot-based memory profiler. This is because of contention for cache resources between application and profiler threads. The parallel memory profiler uses the same L1 cache with the application and shows higher overheads. The snapshot profiler shares only the L2 with application threads.

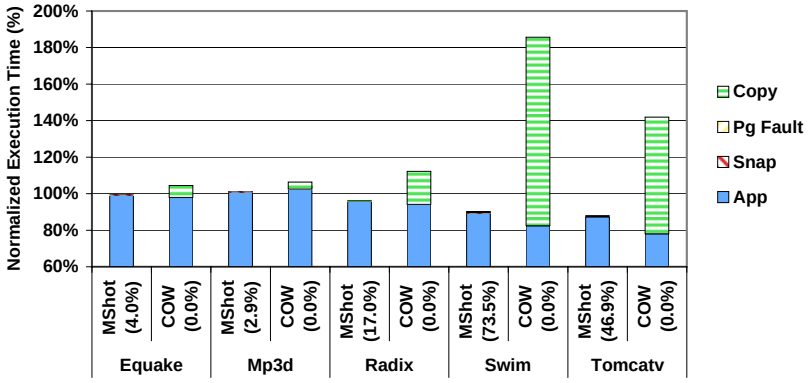


Figure 6.10: The total execution time with the copy-on-write handler (COW) and the snapshot-on-write profiler (MShot). It is normalized to application execution time without write protection (100For copy-on-write, we use 1 core for both application and copying. For snapshot-on-write, we concurrently use 1 core for application and 1 core for copying.

6.6.4 Snapshot-On-Write Test

To compare snapshot-on-write (SOW) using MShot to conventional copy-on-write (COW), we used five scientific applications that stress both systems through accesses to large data structures. Figure 6.10 shows the time breakdown for the applications normalized to the total execution time of the applications running without copy-on-write. In the figure, *Copy* is the time to copy a page, *Pg Fault* the time to invoke the page fault handler, *Snap* the time to take a snapshot, and *App* the time to execute the application code. The figure shows a very interesting result. SOW eliminates the overhead of copy operations, providing improvements ranging from 5% (Equake) to 85% (Swim). Moreover, it allows Radix, Swim, and Tomcatv to run faster than they do without write protection as the SOW threads essentially provide L2 cache prefetching for the application threads. Note that we used scientific applications with high spatial locality in order to not penalize COW, which works poorly if only a small number of words are used after copying a whole page. It turns out that high spatial locality increases the benefits of prefetching from SOW as well. With a 100 cycle memory latency in our system, the penalty of an L2 miss is comparable to that of invoking a page fault handler.

6.7 Related Work

Other than the related works described in Section 6.2.2, there are recent software implementation proposals for mostly concurrent and parallel garbage collectors [156, 44, 66]. While they are competitive to MShot in terms of low runtime overhead, MShot delivers more important benefit of allowing for algorithmic simplicity and easy code management. There are recent advances in dynamic profiling such as SuperPin and Shadow Profiling [152, 92]. However, SuperPin still reports overheads of 100% because it uses heavy-weight cloning/forking and because of the DBT overheads. MShot might be able to help it with a light-weight consistent view on memory. Shadow Profiling reports overheads but has only been evaluated with simpler analyses (e.g. count basic blocks, rather than actually scanning through heap and stack).

6.8 Conclusion

In the chapter, we propose MShot as a system that provides a hardware-assisted memory snapshot. The goal of MShot is to allow for algorithmic simplicity, easy code management, and performance at the same time. MShot allows us to increase the concurrency in software systems without introducing complex code to synchronize accesses to shared data. We showed that MShot can be implemented in a cost-effective manner when combined with a hardware TM architecture. The evaluation result using snapshot for garbage collection, call-path profiling, memory profiling, and copy-on-write show that MShot allows us to exploit additional cores in a CMP to improve the system performance by eliminating almost all overheads due to such system services.

Chapter 7

Accelerating SW Solutions Using TM

7.1 Introduction

Computing systems have become an essential part of modern infrastructure, underlying communication, commerce, government, health care, education, and scientific research. In many environments, *reliability*, *security*, and *debugging* are as important as performance. In the past decades, there have been efforts to develop solutions that independently provide system support for reliability, security, or debugging in software and hardware [138, 114, 117, 36, 144, 160, 121, 122]. Software solutions can flexibly provide various useful features at no hardware cost such as kernel protection with virtual memory protection [144]. However, they have performance issues since they have to rely on the commodity hardware primitives. Hardware solutions add dedicated hardware resources to accelerate a specific feature. Still, the more highly-tuned the hardware resources are for a specific feature, the harder they are to be adopted to commercial systems due to the lack of generality.

In this chapter, we propose a scheme to accelerate software solutions for reliability, security, and debugging with hardware resources for transactional memory. Our proposal is based on two key observations. First, the software solutions for reliability,

security, and debugging frequently require similar low-level features that are implemented with software techniques such as virtual memory protection and dynamic binary translation. Second, TM hardware resources can be reused to support the same features with superior performance than the software techniques. The hardware resources for TM have been designed for concurrency management and cannot substitute a whole solution for reliability, security, and debugging. Nevertheless, the key point of our proposal is to maintain the software solution for functionality and use TM hardware to accelerate only their common case behavior.

We support four specific acceleration primitives for software solutions by leveraging hardware resources for TM:

- **Thread-wide isolated execution** separates the result of thread execution from the rest of system state. The primitive is used to isolate the execution of faulty or untrusted code until its properties are verified.
- **Fine-grain access tracking** detects memory accesses that read from or write to a specified address at cache-line granularity. The primitive allows for tracking and preventing memory accesses to security critical data.
- **User-level software handler** is invoked when a software specified event occurs such as an access to a watched memory address. In collaboration with access tracking, this primitive provides a light-weight signaling mechanism.
- **Process-wide checkpoint** takes a register checkpoint in hardware and builds memory checkpoint gradually by versioning data at cache-line granularity. The primitive is used to roll back the execution of a program to a pointer of interest or to a known safe point.

Compared to conventional software techniques such as virtual memory protection and instrumentation using dynamic binary translation, the acceleration primitives provide the following advantages for reliability, security, and debugging solutions. First, they minimize the time and area overhead of data versioning by providing support in hardware and at cache-line granularity. Second, they avoid the performance

and accuracy problems of false-sharing due to page-level tracking by tracking accesses granularity of cache lines. Third, they eliminate the need for dynamic instrumentation of the application code in order to safely handle the concurrent execution of the application with the reliability, security, or debugging solution. Finally, user-level handlers eliminate the overhead of using OS mechanisms for signaling.

The rest of the chapter is organized as follows. Section 7.2 reviews software solutions for reliability, security, and debugging. Section 7.3 explains the acceleration primitives built over TM hardware. Section 7.4 explains how these primitives accelerate the software solutions, and Section 7.5 quantifies the performance advantages. Section 7.6 discusses related works and Section 7.7 concludes the chapter.

7.2 Software Solutions

There has been significant research on software solutions for reliability, security, and debugging. This section reviews these solutions and their performance bottlenecks. Related hardware techniques are discussed in Section 7.6.

7.2.1 Reliability

Reliability solutions for backward error recovery reduce mean-time-to-recovery (MTTR) by taking checkpoints periodically and by restoring the latest checkpoint when a fault is detected [19, 78, 111, 27]. To avoid the runtime overhead of checkpointing the whole program state, incremental techniques build the checkpoint gradually using virtual memory protection [19, 78]. When a checkpoint is initiated, all pages in the program's address space are write protected. If the program attempts to modify a page later, a protection error occurs and the page is copied to a log (copy-on-write). The checkpoint overhead is reduced further by keeping the log in memory with diskless checkpointing [111] or by using the fork system call [110]. Nevertheless, checkpointing techniques suffer from the time and area overhead introduced by virtual memory protection. Page fault exceptions take hundreds of cycles to handle and cause significant performance issues for memory intensive applications. Tracking modifications

at page granularity causes unnecessary logging of significant amounts of data if only a small portion of each page is modified.

Software recovery techniques such as Swift-R and Trump recover from a single bit fault in the functional units and pipeline registers of a processor by triplicating the original instruction stream or by duplicating it with additional error recovery code at the risk of widening the window of vulnerability [123, 26]. The results from the replicated streams are compared and the correct result is decided by voting. While requiring no dedicated hardware, these techniques suffer from the runtime overhead of replicated execution.

Software fault isolation assigns a portion of the virtual address space (i.e., a fault domain) to a software module and allows the module to access only its own domain with address sandboxing [150]. While it effectively isolates a fault from the rest of system, it introduces additional overhead for cross-domain communication. Since each module can access only its own domain, a remote procedure call (RPC) is used to copy data cross domains. The overhead increases in proportion to the size of shared data, which is likely to grow as parallel programming becomes commonplace.

7.2.2 Security

There have been proposals to protect systems from buffer overflow attacks, a well-studied but still common threat. StackGuard uses a software canary mechanism to detect buffer overflow attacks [36]. Figure 7.1 shows how to detect a stack smashing attack by inserting canaries adjacent to potential targets on the stack such as return addresses or function pointers. Due to the monotonic nature of the string and buffer operations used for stack smashing, an attack must overwrite the canary before changing the protected data. A software implementation of canaries involves checking if the value of the canary has been modified before every use of the protected variable. Propolice is a similar idea but also performs stack variable relocation to hide stack variable layout [113]. PointGuard protects function pointers in stack with the same mechanism [35]. These are all practical solutions against buffer overflows which introduce the runtime overhead of checking and manipulating software canaries. The

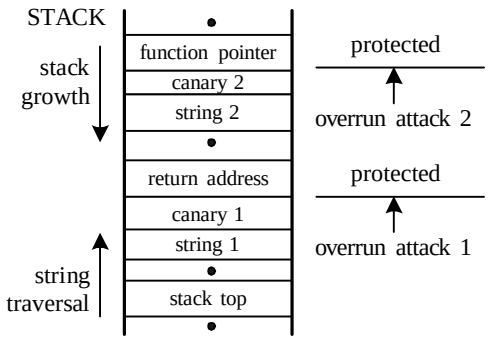


Figure 7.1: Buffer overflow detection using canaries in StackGuard [36]. Canaries 1 and 2 protect the return address and function pointer from attacks 1 and 2 respectively using the corresponding string variables.

overhead is proportional to the number of protected variables and can be low if only a small number is protected.

Several proposals attempt to secure the OS kernel from faulty or malicious drivers and plug-ins. Nooks uses virtual memory mechanism to provide protection domains per driver in order to limit access to kernel data structures [144]. Illegal accesses by an attacker are caught by page fault exceptions. Nooks introduces runtime overhead when drivers modify kernel data legitimately. To guarantee the integrity of kernel data, Nooks uses RPC to provide a copy of the kernel data for the driver and to copy the final data back to the kernel domain when the driver completes its operation in a secure and fault-free way. Depending on the size and complexity of kernel data structures, the copy operation may be expensive (e.g., when copying a large B-tree). Instead of using RPC, VINO and KeyKOS use object-based software transactional memory to save safe data versions before modification [132, 50]. They introduce the performance overhead of data versioning in software. Moreover, they increase the complexity of code as they also require undo methods for all data modifications.

7.2.3 Debugging

Watchpoints provide a useful tool to track and catch incorrect memory accesses. GDB leverages the small number of hardware monitors (e.g., 4 monitors in x86 systems) to support low overhead watchpoints [56]. For a larger number of watchpoints,

GDB traps at every memory access and searches a software structure that tracks watched addresses. To avoid the significant slowdown of trapping at every memory access, EDDI [159] uses dynamic binary translation to introduce checking code that searches the software structure and traps into the debugger only on actual accesses to watchpoints. While more scalable than the original GDB approach, EDDI still suffers from a 3× slowdown due to the additional instrumentation. EDDI also provides an optimization that uses virtual memory protection for coarse-grain detection before instrumentation code is used to detect if the access is to an actual watchpoint. The data breakpoint scheme uses virtual memory protection in a way similar to EDDI [149]. The bottleneck of these techniques is false sharing, as the debugger or instrumentation code is invoked on any access to the page with a watched variable.

During debugging, programmers tiptoe around the point of failure in search of the actual bug. Unfortunately, they often step past an interesting point before they realize its importance. A *bookmark* allows users to mark a starting point before they start to tiptoe. Having missed an important event, they can rollback execution to the bookmark and resume execution without a complete restart. Step-back allows for reverse execution to reach back to the interesting point. It uses the last bookmark and automatically replays from there to the proper point of execution. Recap [108], Bdb [17], and Jockey [127] use a fork system call to take a register checkpoint and use copy-on-write at page granularity to log the memory image of the bookmark. They log external events for replay as well. The longer a bookmark is maintained, the larger the memory footprint they generate due to unnecessary data versioning at page granularity.

Systems such as Memcheck [101] and Purify [115] have been proposed to help detect memory errors. They use dynamic binary translation to introduce metadata and code that check the validity of a memory access to every byte of memory. For example, Memcheck adds a valid bit per memory byte and sets the bit when the byte is allocated and resets it when freed. An access to freed memory is detected by checking the corresponding valid bits. While providing a flexible and versatile platform for debugging, these schemes introduce the runtime overhead of instrumentation, which causes a slowdown of up to 20× [101]. Safemem uses the ECC bits in memory to

detect memory leaks and corruptions [117]. To detect leaks, ECC bits are set during resource allocation to enable the collection of usage statistics. Resources that are not used for a certain period of time after allocation indicate a potential leak. Memory corruption is detected by using the ECC bits to mark write-protected addresses. The disadvantage of Safemem is that it takes 1 to 2 microseconds to set and reset an ECC bit [117]. Exterminator uses software canaries with randomized values for fine-grain detection of memory errors experiencing the common overhead of software canary [104].

7.2.4 Opportunities for Acceleration

The solutions discussed provide useful functionality for reliability, security, and debugging. Table 7.1 summarizes the software techniques used to implement the basic features and the performance overhead they introduce. In this work, we provide hardware acceleration for these features by leveraging the hardware resources for transactional execution. Of course, the main challenge is to provide performance improvements without compromising the flexibility and usefulness of these solutions.

7.3 Acceleration Primitives using TM Hardware

This section provides an overview of TM hardware and the acceleration primitives we build on top of TM resources.

7.3.1 Acceleration Primitives

We propose to use hardware resources for TM to accelerate software solutions for reliability, security, and debugging. Table 7.2 summarizes the interface and functionality of the four primitives we suggest. While TM resources may also support other functions, we narrow the focus of this work to these primitives. *Thread-wide isolated execution* separates the result of thread execution from the rest of system. *Fine-grain access tracking* detects memory accesses to a specified address at cache-line granularity. *User-level software handlers* are invoked on events such as the completion of

Software Technique	Usage	Performance Issues	Used by
Virtual Memory Protection	Process-wide data versioning	False sharing, Page-fault exception, Data versioning at page granularity	Incremental, Diskless, Probabilistic checkpoint
	Memory access tracking	False sharing, Page-fault exception	Efence, EDDI, Data breakpoint
Dynamic Binary Translation	Instruction stream replication	Additional replicated instructions	Swift-R, Trump
	Memory access tracking	Additional checking instructions	Memcheck, purify, EDDI, GDB
Software Canary	Memory access tracking	Canary check and manipulation	Exterminator, StackGuard, Propolice, PointGuard
Object-based SW TM	Isolated execution	Logging safe data per write	Vino, KeyKOS
RPC	Cross-domain communication	Copying argument and return value	Software fault isolation, Nooks
Fork	Register checkpoint, Process-wide data versioning	False sharing, System call overhead, Data versioning at page granularity	Libckpt, Recap, Bdb, Jockey
ECC	Memory access tracking	ECC manipulation, ECC-error interrupt	SafeMem

Table 7.1: Software techniques used by software solutions for reliability, security, and debugging and their performance issues.

isolated execution, the aborting of isolation execution, and the detection of an access to a tracked address. *Process-wide checkpoint* takes register checkpoints in hardware and builds memory checkpoint gradually by versioning data at cache-line granularity.

By glancing at the interface of the primitives, it is not surprising that TM hardware resources can be reused to support their functionality using the architecture shown in Figure 7.2. The rest of this section explains the exact use of TM resources for each primitive. We attempt to support their functionality without modifying TM hardware whenever possible.

Primitive	Interface	Function
Thread-wide Isolated Execution	IS_begin ()	Start isolating the current thread execution from the rest of system.
	IS_end ()	Merge the result of the isolate thread execution to the rest of system.
	IS_abort ()	Discard the isolated thread execution.
Fine-grain Access Tracking	AT_set (*addr*, *R/W*)	Monitor memory access to the cache line including *addr*. R/W is 0 to set read protection, and 1 to set write protection.
	AT_reset (*addr*, *R/W*)	Stop monitoring the cache line including *addr*. R/W is 0 to reset read protection, and 1 to reset write protection.
User-level Software Handler	Register_hdr (*hdr_ptr*, *event_type*)	Register handler pointed by *hdr_ptr* for *event_type*. *event_type* is either IS_end, IS_abort, or access tracking.
	Unregister_hdr (*hdr_ptr*, *event_type*)	Unregister handler pointed by *hdr_ptr* for *event_type*.
Process-wide Checkpoint	CKP_take ()	Take a checkpoint of the current process
	CKP_release ()	Discard the current checkpoint
	CKP_restore ()	Roll back to the current checkpoint

Table 7.2: Four primitives for acceleration of software solutions

7.3.2 Thread-wide Isolated Execution

Thread-wide isolated execution is trivially implemented with TM resources since isolation is one of the basic properties of transactional execution. IS_begin, IS_end, and IS_abort are directly mapped to transaction begin, end, and abort, respectively. The challenge is to support the use of this primitive concurrently with memory transactions in user code. It is reasonable to assume that user-defined transactions and isolated code blocks are perfectly nested (i.e., the one fully encloses the other). The rationale is that it is difficult to construct practical scenarios where sets of operations must be atomic with respect to concurrency but non-atomic with respect to reliability, or vice versa. If TM hardware provides support for nested transactions, we can use these resources to allow for independent rollback of nested transactions and isolated blocks. If there is no nesting support in the TM hardware, subsuming the code block

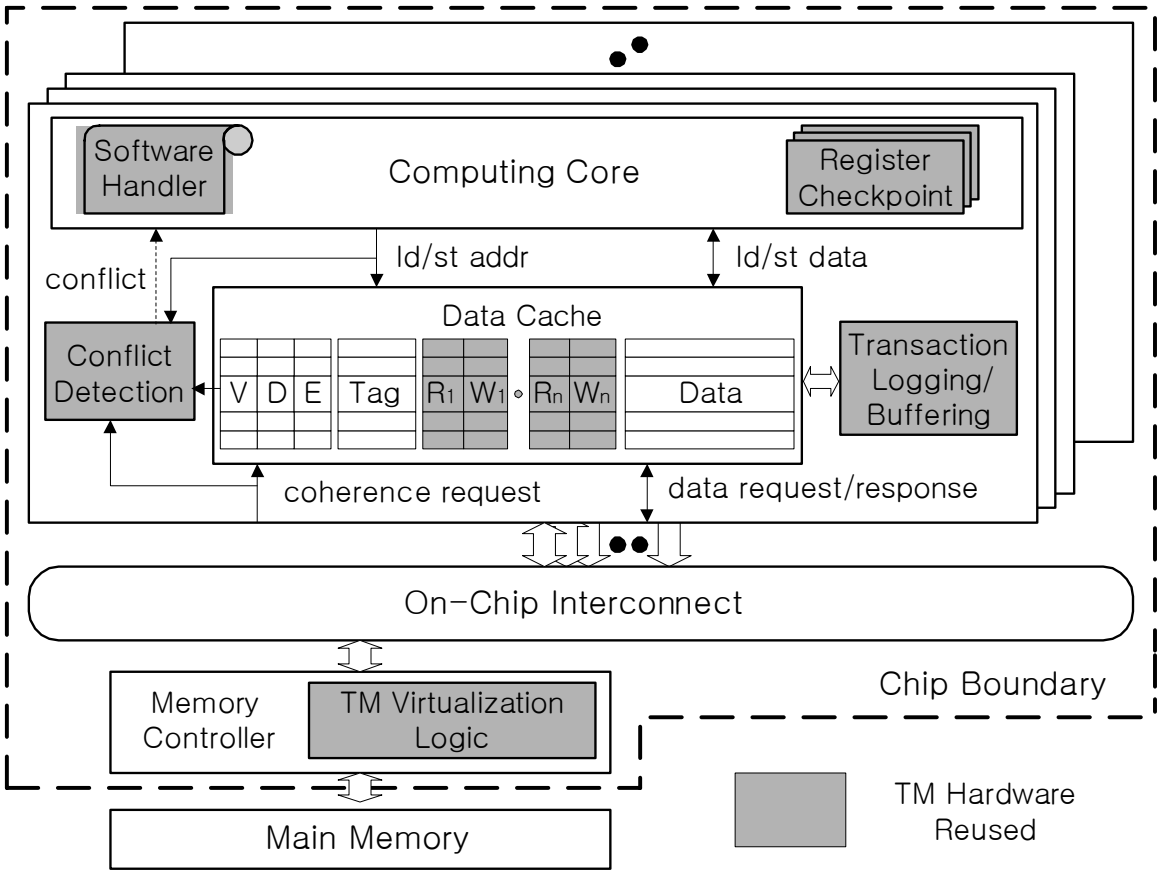


Figure 7.2: TM hardware resources reused for accelerating software solutions. The location of TM virtualization logic may change depending on virtualization mechanisms.

within the transaction or the other way around is sufficient as long as software can determine whether a conflict was due to an address accessed within the transaction or within the isolated code block. If the code block is long enough to cause overflow, we either rely on the TM virtualization mechanism or abort the isolated block and falls back to the conventional techniques used for isolation in the software solutions for reliability (e.g., checkpointing through virtual memory).

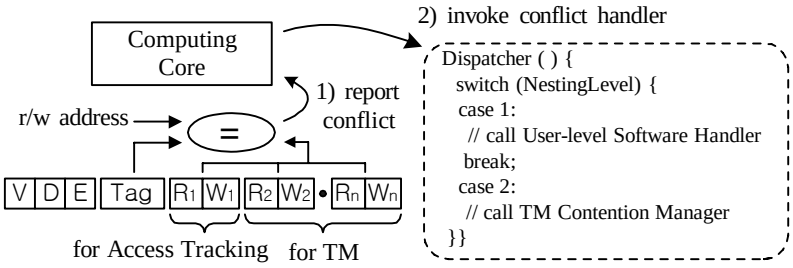


Figure 7.3: Leveraging TM nesting support and handlers to implement fine-grain access tracking.

7.3.3 Fine-grain Access Tracking

We use the TM conflict detection mechanism to implement fine-grain access tracking. We take advantage of nesting support in TM hardware and dedicate the metadata bits for one nesting level to address tracking as shown in Figure 7.3. Metadata bits of the dedicated nesting level are set for cache lines of the monitored addresses with the W bit set for writes and the R bit set for reads. An access to a monitored address is detected similarly to a transactional conflict. Software can easily separate transaction conflicts from monitored accesses by checking if the conflict is against the dedicated nesting level. Since the metadata for TM and access tracking are separate, it is simple to set and reset them independently as transactions begin/end or address protection is turned on/off for various addresses. The only disadvantage is that the number of hardware nesting levels available to TM is reduced by one. In Figure 7.3, we use the lowest nesting level (i.e., level 1) for access tracking. This allows the transaction nesting support to work exactly in the same way as before except for starting from the next lowest nesting level (i.e., level 2).

If the TM hardware does not provide support for nested transactions, it is not practical to implement access tracking on a core that is also executing a transaction since the same metadata bits may be set and reset for both TM and access tracking. To avoid aliasing, we suggest one software and one hardware solution. The software solution is to reserve a core in the CMP system for access tracking. The metadata bits available for conflict detection in this core are used exclusively for access tracking. This may be the core that supports operating system and I/O tasks, which are likely

to require separate core(s) in future CMPs [61]. The hardware solution is to add a single hardware signature per core to separate address protection metadata from transactional metadata. If the hardware resources are not enough to contain all metadata bits set for access tracking, we either rely on the proposed mechanisms for TM virtualization or fall back to the software techniques for access tracking for the remaining addresses (e.g., virtual memory protection).

Most hardware TM designs, including hybrid schemes, provide instructions or functions that map directly to the AT_set and AT_reset functionality. *Early release* can be used for AT_reset as well [84]. In the worst case, it should not be hard to add the interface to set and reset metadata bits since the combinational logic manipulating metadata bits already exists in TM hardware. There is no need to execute AT_set or AT_reset operations in the same thread or processor that runs the monitored programs since the coherence protocol guarantees that conflicts will be detected correctly across the system. However, we need to have the option to turn on conflict detection within a processor/thread to allow a memory access by a processor to find the metadata bits set in its local caches and trigger a conflict. This feature is not needed by the base TM functionality.

If hardware signatures are used to track the read-sets or write-sets of transactions [25], software has to filter out false positives due to aliasing of monitored addresses. While signatures are only gang-cleared for transactional processing, the AT_reset operation needs to remove individual entries as well. If the signatures are not based on counting Bloom filters [48], software should maintain a list of the monitored address and reconstruct the signatures as needed when AT_reset is invoked. Reconstruction can be delayed to balance its overhead with the performance impact of false conflicts. If there is a need for finer granularity than cache line for security or debugging purposes, the hardware detects an access to monitored addresses at cache-line granularity and software is used to provide word- or byte-level granularity.

Note that there is an interesting difference between the typical use of cache metadata bits for transactions and for access tracking. The metadata bits for transaction are short-lived and related to recently accessed data. Because transaction overflow mechanisms lead to increased overhead for TM systems, we should tune the cache

eviction policy to avoid replacement of cache lines with active metadata. The metadata bits for access tracking are long-lived and attached to data accessed infrequently or not supposed to be accessed. It indicates retaining the cache lines with the metadata may lead to poor utilization of the cache. An optimal eviction policy must take into account the nature of the metadata bits when making replacement decisions.

7.3.4 User-level Software Handler

User-level software handlers are directly mapped to TM software handlers [84]. The software handler for IS_end is mapped to the commit handler. The handler for IS_abort is mapped to the abort handler. And the handler for access tracking is mapped to the conflict handler. The same interface that registers and unregisters TM handlers is used for Register_hdr and Unregister_hdr. Figure 7.3 shows how the conflict handler is used as a handler for access tracking. When a conflict is reported, a default handler checks the nesting level the conflict is detected against. If it is against the nesting level dedicated to access tracking, the registered access tracking handler is called. If not, the register handler for TM contention management is invoked.

7.3.5 Process-wide Checkpoint

Process-wide checkpointing is more complex than the other primitives since it is not directly mapped to TM features. Our scheme uses a small software library that synchronizes threads to build a checkpoint. At the beginning of a checkpoint triggered by CKP_take, all threads are synchronized with per-thread signals and use the lowest nesting level transactions to take register checkpoints. If a thread already runs a memory transaction, we either abort it or wait for it to complete.

The memory checkpoint is built gradually using the TM data versioning mechanism. If TM hardware logs old values of data updated within a transaction, we build the checkpoint by accumulating transaction logs. If a user transaction is encountered, we simply initiate a new log to use in case the transaction is aborted. The old log is not discarded and the new log is appended to the old one when the transaction commits. No special support for nesting is necessary to support transaction after a

checkpoint is initiated. Nevertheless, we must track the serializable commit order of transactions across threads so that we can undo a collection of transactions if needed. The checkpoint is restored with CKP_restore by applying the accumulated logs in the reverse order by taking into account the commit order of transactions across threads

If the TM hardware buffers new values until the transaction commits, we use caches as temporal storage to contain the changes since the checkpoint. Conflicts against the lowest nesting level metadata bits are ignored by the TM contention manager since the bits are set only to distinguish data accessed after the checkpoint. If a user transaction is encountered after the checkpoint is initiated, nesting support is required. If the changes overflow the caches, we use the TM virtualization mechanism to store new values in virtual memory [120, 29]. An alternative is to use a simplified virtualization mechanism that simply logs in virtual memory the value in main memory about to be overwritten by the eviction due to the cache overflow. There is a single process-wide log. The checkpoint is restored by invalidating all caches and overflowed data. It is released by committing the buffered new data to the shared state of the system in a manner similar to committing a transaction.

Some software solutions require escape actions that survive through CKP_restore. For example, the debugging solutions supporting reverse execution of programs may want to keep the information about how to replay to the interesting point after the checkpoint restored [108, 17, 127]. They can reserve a small memory area to store such information and access the area with non-cacheable access. Non-cacheable accesses bypass the TM hardware for data versioning in cache, and CKP_restore does not revert the modifications made to the area.

Note that the hardware support for checkpointing guarantees only restoration of registers and memory. It is up to software to deal with external events. For instance, software can use a framework like ReviveIO that provides revocable I/O operations [94]. Exceptions such as page faults are expected and reproducible, but interrupts are not. Hence, software must either take a checkpoint on each interrupt or record them for replay purposes. In some case, software can exploit high-level properties of the I/O system in order to simplify integration with the checkpointing

mechanism. For TCP/IP connections, for example, software does not need to checkpoint on interrupt or log incoming packets. Assuming a long enough timeout interval, it may be sufficient to just defer sending acknowledgments until the next checkpoint. If a checkpoint is restored, both the record of the received packets and the queue of pending acknowledgments are rolled back, causing the system to behave as if the network failed to deliver the packets. TCP/IP's robust retransmission mechanism will cause the packets to be retransmitted.

7.4 Accelerating SW Solutions

This section explains how the TM-based acceleration primitives in Section 7.3 are used within software solutions for reliability, security, and debugging and what the expected benefits are. The basic idea is to rely on software for flexible functionality and to use the proposed primitives to accelerate the performance of the common case behavior. Table 7.3 shows the common low-level features that the solutions require for functionality and the hardware-assisted primitives that support the features with superior performance than the alternative software techniques. If the hardware resources used for the primitives are exhausted, we either rely on TM virtualziation mechanisms or the original software techniques.

7.4.1 Process-wide Data Versioning

Process-wide data versioning is a key feature required by the reliability solutions that recover errors backward with periodic checkpointing [111, 78, 27]. It builds directly upon process-wide checkpoint in our scheme. CKP_take is used to start the first checkpoint. The next checkpoints are composed of CKP_release followed by CKP_take. Between checkpoints, data versioning is performed at cache-line granularity in hardware out of the critical path of program execution. The hardware exhaustion for checkpoint can be handled either by TM virtualization mechanisms [120, 29] or by shortening the checkpointing interval. A short checkpointing interval reduces the recovery time after restoring a checkpoint as well. However, it increases the checkpointing overhead

Common Feature Requirement	Hardware-assisted Acceleration Primitive				Alternative Software Technique
	Thread-wide Isolated Execution	Fine-grain Access Tracking	User-level Software Handler	Process-wide Check-point	
Process-wide Data Versioning	√			√	Virtual memory protection, fork
Memory Access Tracking		√	√		Virtual memory protection, dynamic binary translation, ECC, software canary
Isolated Execution	√		√		Object-based SW TM
Instruction Stream Replication	√		√		Dynamic binary translation
Cross-domain Communi-cation	√		√		Remote procedure call

Table 7.3: Common feature requirements supported by the hardware-assisted acceleration primitives with superior performance than the alternative software techniques.

with frequent execution of CKP_take. The software solutions should determine the checkpoint interval that balances the TM virtualization overhead, MTTR, and the checkpoint overhead. It is desirable to use an interval that causes the hardware exhaustion rarely to accelerate the solutions fully in hardware. Compared to virtual memory protection, process-wide checkpoint accelerates the solutions by removing most of false sharing, eliminating page fault exception, and reducing the log size significantly. We compare process-wide checkpoint and virtual memory protection for checkpoint quantitatively in Section 7.5.2.

While process-wide checkpoint supports process-wide data versioning efficiently,

there are opportunities to reduce the checkpointing overhead further for certain computation faults with thread-wide isolated execution. Programmers or compilers declare a code fragment to be an isolated code block enclosed with IS_begin and IS_end. During the execution of the code block, its output is isolated from the rest of the system; other threads cannot see its memory updates until it completes. If an error that affects a single isolated block is detected, the block can be rolled back with a much lower cost than that of restoring a process-wide checkpoint. This dramatically reduces the MTTR. Thread-wide isolated execution is useful for recovery from both hardware and software errors. The isolated code blocks coexist with process-wide checkpoints, so recovery is still possible for faults that encompass multiple isolated blocks or that cannot be detected quickly. We show the performance benefit of using thread-wide isolated execution for reliability in Section 7.5.2.

Process-wide checkpoint accelerates the debugging solutions that support bookmark and reverse execution [110, 108, 17, 127]. A bookmark is created with CKP_take, restored with CKP_restore, and discarded with CKP_release. In comparison to the fork system call, process-wide checkpoint supports faster register checkpointing and memory data versioning in hardware. Reverse execution is done by restoring a bookmark and moving forward to the point of interest. Escape actions to remember where to move forward after restoring a checkpoint are done through non-cacheable access as explained in Section 7.3.5

7.4.2 Memory Access Tracking

Memory access tracking is an essential feature for the security solutions that prevent untrusted code from accessing important data illegally. It builds upon fine-grain access tracking and user-level software handler in our scheme. For the solutions using a software canary to prevent buffer overrun attacks, an aligned cache-line-sized space is allocated next to return addresses and pointers in the stack in the same way as a software canary is allocated [104, 36, 113, 35]. The space is write-protected with AT_set. A buffer overrun attack to compromise the return addresses and pointers triggers the user-level handler registered for access tracking as explained in Figure 7.3.

Inside the handler, the security solutions take a proper measure to deal with the attack (e.g., killing the process). This mechanism eliminates the additional code to check software canary values and improves performance as shown in Section 7.5.3.

Fine-grain access tracking helps debugging solutions to support a scalable watchpoint. Instead of using virtual memory protection [159, 149], watchpoint solutions set read/write protection to watched addresses using AT_set. This eliminates false sharing at page granularity and page fault exception handling overhead. If the hardware resources are exhausted due to lots of watchpoints, we either rely on TM virtualization mechanism or fall back to virtual memory protection for the overflowed portion of watchpoints. Some watchpoint solutions instrument trap instructions per memory access after scanty hardware monitors are exhausted [56]. Fine-grain access tracking provides scalable hardware watchpoints for the solutions to avoid trap instruction instrumentation. We compare fine-grain access tracking, virtual memory protection, and trap instruction instrumentation for watchpoint implementation in Section 7.5.4.

Some solutions instrument monitoring code with dynamic binary translation to detect memory errors such as accesses to freed memory [101, 115]. Fine-grain access tracking easily detects such accesses by setting write protection to freed memory regions with AT_set. The protection is reset with AT_reset when the memory regions are reallocated. This approach improves performance significantly by eliminating the monitor code. We compare fine-grain access tracking, virtual memory protection, and dynamic binary translation for watchpoint implementation and memory error debugging in Section 7.5.4. For the solutions that use ECC bits for memory error detection [117], fine-grain access tracking provides metadata bits in cache for faster bit manipulation.

7.4.3 Isolated Execution

Security solutions using object-based software transactional memory for isolated execution take direct advantage of the hardware support for isolated execution in our scheme [132, 50]. Instead of software transactions, they use faster thread-wide isolated

execution. Since transactions are short-lived in the common case [32], most transactions should be able to run with hardware-assisted isolated execution. If the resources are exhausted, they either rely on the TM virtualization mechanism [120, 29] or fall back to the original object-based software TM [41]. Using the object-based software TM as a backup system demands two versions of code: one for hardware TM and the other with software barriers for software TM. An alternative is to use page-based software TM that supports software transactions transparent to TM programs.

7.4.4 Instruction Stream Replication

Our scheme does not provide a feature to replicate instruction streams, but helps reduce the number of instructions to be replicated for software fault recovery [26]. The basic idea is to mix forward recovery of software fault recovery and backward recovery with thread-wide isolated execution. Assuming that faults are rare, we start with backward recovery. A program is split into small isolated code blocks, each of which starts with IS_begin. The code blocks have duplicated instruction streams for fault detection. If the results from duplicated instruction streams coincide, IS_end is called. If they differ, a fault is detected and IS_abort is called to restart. Only for the blocks with frequent faults, is forward recovery applied. The code blocks have triplicated instruction streams that vote for the right result. The frequency of failure for a block is tracked by IS_abort handler. Switching to forward recovery is done either by dynamic binary transaction [123] or with another version of the code block generated at compile time if the instruction replication happens at source code level [26]. Using dynamic binary translation, we can adjust the size of large code blocks to avoid the hardware resource exhaustion for isolated execution. This approach helps avoid code triplication in most cases.

7.4.5 Cross-domain Communication

Some security solutions provide multiple domains to isolate software modules [150, 144]. They use remote procedure call for cross-domain communication to preserve

Feature	Description
CPU	2GHz, single-issue, in-order x86 core
L1 Cache	64 KB, 4-way, 32B line, MESI, write-back, 1 cycle hit time, private
L2 Cache	8 MB, 8-way, 32B block, MESI, write back, 10 cycle latency, shared, 8 banks, bit vector of sharers
Memory	4 GB, 100 cycle latency
Interconnect	Tiled network, 32B links, 3 cycles per hop

Table 7.4: Parameters for the simulated multi-core system.

data integrity of each domain with call-by-value semantics. Our scheme uses thread-wide isolated execution and user-level software handler to accelerate cross-domain communication. It allows call-by-reference semantics that hands over pointer to data directly instead of copying data for faster cross-domain communication. A method invocation to another domain starts with IS_begin. During the isolated execution, old safe values to be overwritten are saved through thread-wide isolated execution. After the method invocation completes successfully, IS_end is called. If a fault is detected, IS_abort is called to restore the safe values. IS_abort handler is used to initiate recovery procedure for the fault. This approach eliminates the runtime overhead of copying data, which can be significant for complex data structures such as large trees. It simplifies call/return stubs for cross-domain communication as well.

7.5 Performance Evaluation

This section evaluates the acceleration primitives quantitatively in the context of reliability, security, and debugging. We explain our methodology of using a cycle-accurate simulator to compare the primitives with software techniques and to analyze them.

7.5.1 Methodology

We used a cycle-accurate CMP simulator for an 8-core x86 CMP system. Table 7.4 summarizes the basic parameters of the environment. All operations except loads

and stores have a CPI of 1.0. However, all details of the memory hierarchy timings, including contention and queuing events, are modeled. The shared L2 cache also acts as a directory as each entry includes a bit-vector that identifies the L1 caches that share the cache line. It takes 200 cycles to trigger page fault exception handler.

We implemented the acceleration primitives over a simulated TM system that has the following hardware resources. It versions data lazily by using local caches as a write buffer for an ongoing transaction [60]. Every entry in the L1 cache for a core contains four pairs of metadata bits (R and W) that indicate if the line has been read or written by a transaction at the corresponding nesting level [84]. Coherence messages are looked up against these metadata bits to detect conflicts. A conflict occurs when a shared request finds any of the W bits set or an exclusive request finds any of the R or W bits set. If a transaction overflows the L1 cache, the transactional state is virtualized in the following manner. The metadata bits are summarized in a set of conservative signatures implemented by Bloom filters. Each core has four pairs of signatures to summarize overflowed metadata (one R and one W signature per nesting level) An overflowed transaction may experience some false conflicts due to the inexact nature of Bloom filters. If a cache line that holds write-set data is evicted (W bit set), it is written back to the L2 and/or main memory only after the old values are copied into a per-thread undo-log in memory. This approach provides full write buffer virtualization but introduces some runtime overhead when aborting transactions that have overflowed their caches.

To implement the four primitives, we reserve the metadata bits of the lowest nesting level for our scheme. If the primitives are rarely used, one can use a dynamic scheme that allocates a nesting level for them on demand at each core. We modified the software conflict handler to check the nesting-level of conflicts and to invoke a user-level software handler as shown in Figure 7.3. Process-wide checkpoint flushes dirty lines in caches during checkpointing and buffers modifications after checkpointing in caches. The checkpoint restoration invalidates caches. If there are active transactions when CKP_take is called, they are allowed to complete before the checkpoint is taken.

We used 16 applications and one microbenchmark. Eight parallel applications

Implementation Technique	MP3D	RADIX	TOMCATV	SWIM	EQUAKE
Process-wide Checkpoint	5.06%	1.96%	2.95%	1.66%	4.67%
Software Checkpoint	30.19%	794.37%	556.89%	777.23%	301.49%

Table 7.5: Runtime overhead of hardware-assisted process-wide checkpoint and software checkpoint with virtual memory protection.

were used for reliability and debugging experiments: *mp3d* and *radix* from SPLASH-2 [155]; *swim*, *tomcatv*, and *equake* from SPEComp [141]; and *genome*, *kmeans*, and *vacation* from STAMP [23], a TM benchmark suite. The STAMP programs use transactions for up to 95% of their execution time. For debugging, we used three applications: *cfrac* (factorization), *mpeg2* (multimedia), and w3m (test-based web browser). A microbenchmark is used to measure the overhead of randomly placed watchpoints. For security experiments, four older versions of open source applications with known vulnerabilities to buffer overflow attack were used: *gzip* (compression utility), *polymorph* (filename rewriter), *ghttpd* and *nullhttpd* (httpd servers). We compiled the applications with GCC 4.3 on x86 Linux 2.6.9.

7.5.2 Evaluation of Reliability Support

We first compare our process-wide checkpoint with software checkpoint based on virtual memory protection. The software checkpoint fields out write protection on all pages and builds the checkpoint gradually by copying the pages to be written when page fault exceptions are triggered. Checkpoints are taken periodically at every 50K cycles. Table 7.5 shows the overheads of two checkpoint techniques. Software checkpoint adds 490.03% runtime overhead on average mainly due to page copying overhead, which makes it impractical to use it at short intervals for many applications. On the other hand, hardware checkpoint adds only 3.26% runtime overhead.

Figure 7.4 shows further analysis on process-wide checkpoint with fault injection, thread-wide isolated execution, and TM programs. It shows the normalized execution time of 8 applications in 4 configurations. Lower bars are better. The *BASE* bar shows

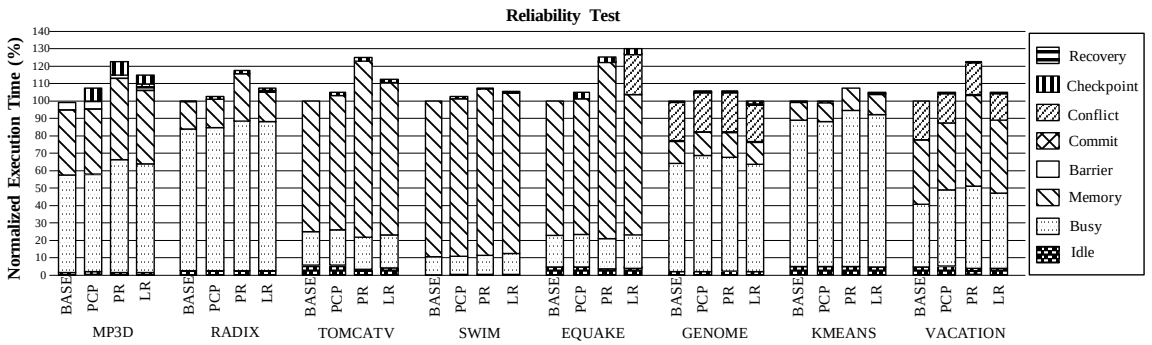


Figure 7.4: Normalized execution time for checkpoint test in the context of reliability support with 4 configurations: no checkpointing and no faults (BASE), process-wide checkpointing only (PCP), recovery with process-wide checkpoint against faults (PR), and additional thread-wide isolated execution for local recovery (LR). The execution times are normalized to BASE which is always 100%. Checkpoints are taken at every 50K cycles. Faults are injected at every 1M cycles.

baseline performance without checkpointing and no injected faults. It is always 100%. The *PCP* bar shows the overhead when a process-wide checkpoint is taken every 50K cycles, the same configuration as in the previous test. No faults were inserted in this case. The *PR* bar represents process-wide checkpoints at every 50K cycles and transient faults injected randomly, one per 1M cycles on the average. The fault model for the test assumes that main memory, cache, and registers used for checkpointing are protected with ECC [42, 114]. The transient faults in computation or communication (e.g., packet loss or logic error) are handled by restoring a checkpoint.

The *LR* bar is similar to PR, but also uses thread-wide isolated execution to support local recovery within each thread. The faults are recovered locally within an code block isolated with thread-wide isolated execution only when fault detection occurs before the code block completes. For the 5 non-transactional programs, isolated code blocks were added to cover inner loop bodies. For the TM programs, user transaction boundaries were used to define the isolated code blocks.

Figure 7.4 shows that the overhead of process-wide checkpoints (PCP) is less than 3.5% for most applications. At an interval of 50K cycles, the number of dirty lines in the CMP is small and flushing them to memory is fast. Moreover, logging introduces virtually no overhead as it happens rarely with 8MB L2 cache. When errors

Implementation Technique	Gzip	Polymorph	Ghttpd	Nullhttpd
Fine-grain Access Tracking	0.2%	0.4%	0.3%	0.3%
StackGuard	3.1%	3.6%	3.3%	2.0%

Table 7.6: The runtime overhead of buffer overflow detection with fine-grain access tracking and StackGuard.

are inserted every 1M cycles, the overhead of process-wide recovery from the latest checkpoint (PR) is 20% to 30% for five programs and less than 10% for the rest. The recovery overhead is primarily due to the requirement that caches should be invalidated during checkpoint restoration, causing all caches to be cold after recovery. The overhead varies depending on the size of the working set that needs to be refetched and the application's inherent locality. When errors can be handled using the local recovery with thread-wide isolated execution, the recovery overhead is dramatically reduced as caches remain warm. Equake is an exception as the use of the isolated execution leads to frequent false conflicts across threads, introducing significant runtime overhead that outweighs the recovery benefits. Our experiments show that process-wide checkpoint and thread-wide isolated execution with hardware assist provide fast mechanisms for error recovery in a cost-effective manner.

7.5.3 Evaluation of Security Support

We used user-level software handler and fine-grain access tracking to implement a runtime scheme for buffer overflow detection. We modified GCC 4.3 to emit AT_set and AT_reset instead of the software canaries provided by StackGuard [36]. Both approaches detected the same vulnerabilities in our application set. Table 7.6 shows the runtime overhead of each approach. The overhead of fine-grain access tracking is almost an order of magnitude lower as it requires only two instructions per function. In contrast, StackGuard introduces two instructions in the prologue and five in the epilogue of every function call. The performance difference would be even more dramatic if we also used canaries to protect other data such as function pointers for improved security coverage. StackGuard requires several instructions for each read or write access to a protected pointer. In contrast, fine-grain access tracking requires

Implementation Technique	Cfrac	Gzip	Mpeg2	W3M
Fine-grain Access Tracking	1.02×	1.01×	1.08×	1.06×
Dynamic Binary Translation	44.69×	44.24×	55.98×	32.15×

Table 7.7: The slowdown for debugging memory errors with fine-grain access tracking and dynamic binary translation.

Duration	L2 Size	Tomcatv	Swim	Equake	Vacation
1M cycles	**1MB**	0.68MB	0.27MB	0.53MB	0.01MB
	8MB	0.22MB	0.01MB	0.01MB	0.00MB
10M cycles	**1MB**	5.17MB	8.38MB	2.76MB	0.36MB
	8MB	1.20MB	3.95MB	0.22MB	0.00MB

Table 7.8: The storage overhead of parallel bookmark in main memory.

only one instruction when the pointer is allocated and one at deallocation time.

7.5.4 Evaluation of Debugging Support

We compared fine-grain access tracking and dynamic binary translation to catch memory errors that access freed memory bytes. A valid bit per memory byte is assigned. The bit is set when the byte is allocated and reset when freed. A memory access to a byte with the bit reset indicates a memory error. With fine-grain access tracking, the bit is manipulated with AT_set and AT_reset and an illegal access is notified through the user-level access tracking handler. For dynamic binary transaction, we added instructions to check the bits on every memory access. The applications we used do not have memory errors so that the test measures the runtime overhead without errors. In Table 7.7, fine-grain access tracking shows negligible slowdown of 4% on average since the applications do not allocate and free memory frequently so that the overhead to set and reset the bits is low. On the other hand, the additional checking instructions with dynamic binary translation are executed per memory access, which leads to 44× slowdown.

We implemented and evaluated parallel bookmark with process-wide checkpoint. The longer a parallel bookmark is maintained, the more memory is required to maintain the undo log for overflowed cache lines. Table 7.8 shows the main memory overhead of parallel bookmark for the applications with the largest undo logs, as a

Implementation Technique	100 points	200 points	500 points	1000 points
Fine-grain Access Tracking	1.02	1.06	1.19	1.39
Virtual Memory Protection	1.46	1.81	2.56	3.02
Dynamic Binary Translation	23.75	23.79	24.08	24.18

Table 7.9: The normalized execution time of watchpoint microbenchmark with three watchpoint implementation: fine-grain access tracking, virtual memory protection, and dynamic binary translation. The number of watchpoints per core has changed from 100 to 200, 500, and 1000. The execution time is normalized to the execution without watchpoint.

function of checkpoint interval and L2 cache size. It shows that the storage overhead is reasonable in all cases. Even if a bookmark is maintained for 10M cycles for a system with a 1MB cache, the debugger's footprint increases by only 8MB in the worst case. A larger L2 cache size helps significantly as it reduces the number of off-chip evictions that lead to logging. With an 8MB L2 cache, the worst case requirement drops to 4MB. This is encouraging as the current trend is toward increased on-chip cache capacity for CMP systems.

To test watchpoints, we created a microbenchmark that places watchpoints and accesses memory according to a pre-generated random sequence of addresses. Three techniques to implement watchpoint are tested: fine-grain access tracking, virtual memory protection that write-protects a whole page including a watched address, and dynamic binary translation that instruments trap instructions to look up an accessed address in the list of watched addresses stored in virtual memory space. The number of watchpoints per core was changed from 100 to 200, 500, and 1000. The execution time is normalized to the execution without watchpoint. Hardware watchpoint scales well and shows under 40% runtime overhead for all cases. Watchpoint with virtual memory protection shows up to 3× slowdown mainly due to the page fault exception overhead. Watchpoint with dynamic binary translation suffers from frequent traps and shows up to 24× slowdown on average. The trap overhead dominates the execution time so that the overall overhead does not change much over

different numbers of watchpoints per core.

7.6 Related Work

There have been efforts on hardware solutions for reliability, security, and debugging. Unlike our scheme that targets common requirements, they are best tuned for a specific feature with dedicated hardware resources.

SafetyNet [138] provides low-overhead protection from transient faults with reduced mean-time-to-recovery (MTTR). It supports multiple global checkpoints without stopping execution to initiate one. A checkpoint is built gradually by logging data in additional storage next to caches and in memory. After detecting a fault, it rolls back to the last validated checkpoint by applying the logs. Revive [114] organizes main memory similarly to RAID in order to deal with both transient and permanent errors in a directory-based system. It stops the system to start building a global checkpoint. All dirty data in caches are flushed to memory to preserve data safely. Data logging happens whenever dirty data are evicted to memory. Revive recovers from a fault by invalidating dirty data in caches and applying the undo logs. ReviveI/O [94] supports recovery of I/O resources such as network connections and disks. It introduces a scheme to undo and redo I/O operations and builds upon Revive to rollback memory state [114]. A monitor-and-recover programming paradigm is used to enhance the reliability of software in [106]. The original code is executed speculatively after taking a thread-wide local checkpoint. The code is allowed to finish only when a corresponding check code running in parallel confirms its safe execution. Otherwise, the original code rolls back to the local checkpoint.

Mondrian [154] provides a finer-grain memory protection mechanism that provides multiple protection domains in a flexible way. It allows arbitrary permission control down to the granularity of 32-bit words. Space overhead is reduced by permission table compression, and runtime overhead is minimized by the use of a two-level permission cache. The iWatcher system [160] is a good example of hardware support for debugging. It provides a large number of hardware-assisted watchpoints and a flexible software handler mechanism to eliminate the overhead of dynamic code rewriting,

helping to find more software bugs.

There have been proposals to use transactional memory for something other than parallel programming. A recent study used TM to simplify the implementation of sophisticated compiler transformations [96]. Transactions simplify compensation code, allow lazy verification of fast-path assertions, and provide aggregate fences.

7.7 Conclusion

In this chapter, we propose a scheme to accelerate software solutions for reliability, security, and debugging with hardware resources for transactional memory. We provide four acceleration primitives leveraging TM hardware resources: thread-wide isolated execution, fine-grain access tracking, user-level software handler, and process-wide checkpoint. We qualitatively explain how the primitives accelerate software solutions and quantitatively analyze the effectiveness of the primitives for acceleration. We expect TM designers to consider the interesting use cases in the paper , which show that TM hardware resources are beneficial not only for parallel programming but also for software solutions designed for other important purposes.

Chapter 8

Conclusion

This thesis address the challenges to building an efficient and practical TM system and explore the opportunities to use it to support system software and to improve important system metrics other than performance. The contributions of the thesis are the following.

- We analyze the common case transactional behavior of multithreaded programs. We measure common-case characteristics of transactions to provide key insights on the architectural support for efficient TM systems.
- We present eXtended Transactional Memory (XTM), a software-base TM virtualization system using virtual memory support in the operating system. It virtualizes all aspects of transactional execution (time, space, and nesting depth) for practical TM systems.
- We suggest a practical solution to use TM for correct execution of multithreaded programs within DBT frameworks. Memory transactions are used to eliminate races involving metadata by encapsulating the data and metadata accesses in a trace, within one atomic block.
- We present MShot, a hardware-assisted memory snapshot system using TM to allow for algorithmic simplicity, easy code management, and performance at

the same time. We use the hardware resource for TM to accelerate a memory snapshot of arbitrary lifetime that consist of multiple disjoint memory regions.

- We propose a scheme to accelerate software solutions for reliability, security, and debugging with hardware resources for TM. Four primitives are provided for acceleration: thread-wide isolated execution, fine-grain access tracking, user-level software handler, and process-wide checkpoint.

Overall, this thesis provides important information and novel techniques towards the design of an efficient and practical TM system useful for multiple purposes. System designers can utilize these techniques to address TM implementation challenges and justify the cost of hardware support for TM across multiple applications.

For future work, it is interesting to re-design the TM hardware resources for multiple purposes other than concurrency control. As shown in the thesis, the resources are also useful for other important system features such as reliability, security, and debugging. What if we design them from the scratch to serve other purposes? What are the common set of hardware resources for the purposes? Do we provide the interface to the resources other than transaction begin and end? There are many challenges to design the versatile resources.

Another interesting work is to provide consistency to TM systems. The current hardware TM proposals provide atomicity and isolation. While they are enough to produce a serializable transaction history, there are cases that a strict serializability is not required for correct execution of applications. For example, if an application provides a set of rules that define consistent(or correct) states, a correct execution of the application can be guaranteed by enforcing transactions to change the application's state from a consistent state to another consistent state. This may help TM systems ignore unnecessary conflicts and improve performance.

Bibliography

[1] A.-R. Adl-Tabatabai, B. Lewis, et al. Compiler and Runtime Support for Efficient Software Transactional Memory. In *the Proc. of the 2006 Conf. on Programming Language Design and Implementation*, June 2006.

[2] Y. Afek, H. Attiya, et al. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.

[3] H. Akkary et al. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *the Proc. of the 36th Intl. Symp. on Microarchitecture*, San Diego, CA, Dec. 2003.

[4] K. Alnas, E. H. Krstiansen, et al. Scalable coherence interface. In *CompuEuro '90. Proc. of the 1990 Intl. Conf. on Computer Systems and Software Engineering*, 1990.

[5] B. Alpern and S. Augart. The Jikes Research Virtual Machine project: Buliding an open-source research community. *IBM Systems Journal*, 2005.

[6] C. S. Ananian, K. Asanović, et al. Unbounded Transactional Memory, Feb. 2005.

[7] C. S. Ananian and M. Rinard. Efficient Software Transactions for Object-Oriented Languages. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. October 2005.

[8] J. Anderson. Composite registers. In *PODC '90: Proc. of the 9th ACM symp. on Principles of distributed computing*, 1990.

[9] Apache HTTP Server Project. http://httpd.apache.org/.

[10] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *the Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, Vancouver, Canada, June 2000.

[11] L. Baugh and C. Zilles. An analysis of i/o and syscalls in critical sections and their implications for transactional memory. In *ISPAA '08: Proc. of Intl. Symp. on Performance Analysis of Systems and Software*, 2008.

[12] A. Bilas, J. Fritts, and J. P. Singh. Real-time parallel mpeg-2 decoding in software. In *Proc. 11th Intl. Parallel Processing Symp.*, 1997.

[13] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of ACM*, 13(7), July 1970.

[14] C. Blundell, J. Devietti, et al. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2), 2007.

[15] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Workshop on Duplicating, Deconstructing, and Debunking (WDDD)*, June 2005.

[16] H.-J. Boehm. Space efficient conservative garbage collection. In *PLDI '93: Proc. of the ACM SIGPLAN 1993 conf. on Programming language design and implementation*, 1993.

[17] B. Boothe. Efficient algorithms for bidirectional debugging. In *PLDI '00: Proc. of ACM SIGPLAN 2000 Conf. on Programming Language Design and Implementation*, 2000.

[18] E. Borin, C. Wang, et al. Software-based Transparent and Comprehensive Control-flow Error Detection. In *the Proc. of the 4th Intl. Symp. Code Generation and Optimization (CGO)*, New York, NY, March 2006.

[19] N. S. Bowen and D. K. Pradhan. Processor and memory-based checkpoint and rollback recovery. *Computer*, 26(2):22–31, 1993.

[20] Efficient Belief Propagation for Early Vision. http://people.cs.uchicago.edu/ pff/bp/.

[21] R. P. Brent. Recent progress and prospects for integer factorisation algorithms. In *COCOON '00: Proc. of the 6th Intl. Conf. on Computing and Combinatorics*, 2000.

[22] D. Bruening, E. Duesterwald, and S. Amarasinghe. Design and Implementation of a Dynamic Optimization Framework for Windows. In *the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO)*, Austin, TX, December 2001.

[23] C. Cao Minh, M. Trautmann, et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *the Proc. of the 34th Intl. Symp. on Computer Architecture*. June 2007.

[24] B. D. Carlstrom, A. McDonald, et al. The Atomos Transactional Programming Language. In *the Proc. of the Conf. on Programming Language Design and Implementation*, June 2006.

[25] L. Ceze, J. Tuck, et al. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *the Proc. of the 33rd Intl. Symp. on Computer Architecture (ISCA)*, June 2006.

[26] J. Chang, G. A. Reis, and D. I. August. Automatic instruction-level software-only recovery. In *DSN '06: Proc. of Intl. Conf. on Dependable Systems and Networks*, 2006.

[27] H. chang Nam, J. Kim, S. Hong, and S. Lee. Reliable probabilistic checkpointing. In *PRDC '99: Proc. of the 1999 Pacific Rim Intl. Symp. on Dependable Computing*, 1999.

[28] A. Chernoff, M. Herdeg, et al. FX!32 A Profile-Directed Binary Translator. *IEEE Micro*, 18(2), Mar. 1998.

[29] W. Chuang, S. Narayanasamy, et al. Unbounded Page-Based Transactional Memory. In *the Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. Oct. 2006.

[30] Computer Hardware Understanding Development Tools. http://developer.apple.com/tools/performance/.

[31] J. Chung, C. Cao Minh, et al. Tradeoffs in Transactional Memory Virtualization. In *the Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. Oct. 2006.

[32] J. Chung, H. Chafi, et al. The Common Case Transactional Behavior of Multithreaded Programs. In *the Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture*, Feb. 2006.

[33] J. Clause, W. Li, and A. Orso. Dytan: A Generic Dynamic Taint Analysis Framework. In *the Proc. of the Intl. Symp. on Software Testing and Analysis (ISSTA)*, London, UK, July 2007.

[34] M. Costa, J. Crowcroft, et al. Vigilante: End-to-End Containment of Internet Worms. In *the Proc. of the 20th ACM Symp. on Operating Systems Principles (SOSP)*, Brighton, UK, Oct. 2005.

[35] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguardtm: protecting pointers from buffer overflow vulnerabilities. In *SSYM'03: Proc. of the 12th Conf. on USENIX Security Symp.*, 2003.

[36] C. Cowan, C. Pu, et al. Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In *SSYM'98: Proc. of the 7th Conf. on USENIX Security Symp.*, 1998.

[37] J. R. Crandall and F. T. Chong. MINOS: Control Data Attack Prevention Orthogonal to Memory Model. In *the Proc. of the 37th Intl. Symp. on Microarchitecture (MICRO)*, Portland, OR, December 2004.

[38] The DaCapo Benchmark Suite. http://www-ali.cs.umass.edu/DaCapo/gcbm.html.

[39] L. Dagum. Openmp: A proposed industry standard api for shared memory programming, October 1997.

[40] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A Flexible Information Flow Architecture for Software Security. In *the Proc. of the 34th Intl. Symp. on Computer Architecture (ISCA)*, San Diego, CA, June 2007.

[41] P. Damron, A. Fedorova, et al. Hybrid transactional memory. In *the Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, Oct. 2006.

[42] T. Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory. Nov 1997.

[43] A. Demmers, M. Weiser, et al. Combining generational and conservative garbage collection: framework and implementations. In *POPL '90: Proc. of the 17th ACM SIGPLAN-SIGACT symp. on Principles of programming languages*, 1990.

[44] D. Detlefs, C. Flood, et al. Garbage-first garbage collection. In *ISMM '04: Proc. of the 4nd intl. symp. on Memory management*, 2004.

[45] D. Dice, O. Shalev, et al. Transactional Locking II. In *the Proc. of the 20th Intl. Symp. on Distributed Computing*, Sept. 2006.

[46] D. Doligez and G. Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL '94: Proc. of the 21st ACM SIGPLAN-SIGACT symp. on Principles of programming languages*, 1994.

[47] T. Domani, E. K. Kolodner, et al. Implementing an on-the-fly garbage collector for java. In *ISMM '00: Proc. of the 2nd intl. symp. on Memory management*, 2000.

[48] L. Fan, P. Cao, et al. Summary cache: a scalable wide-area web cache sharing protocol. *SIGCOMM Comput. Commun. Rev.*, 28(4), 1998.

[49] http://sourceforge.net/projects/fastdb/.

[50] W. S. Frantz and C. R. Landau. Object oriented transaction processing in the keykos microkernel. In *moas'93: USENIX Symp. on USENIX Microkernels and Other Kernel Architectures Symp.*, 1993.

[51] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *ICS '05: Proc. of the 19th intl. conf. on Supercomputing*, 2005.

[52] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[53] R. Garg, V. K. Garg, and Y. Sabharwal. Scalable algorithms for global snapshots in distributed systems. In *ICS '06: Proc. of the 20th intl. conf. on Supercomputing*, 2006.

[54] D. Gay and B. Steensgaard. Fast Escape Analysis and Stack Allocation for Object-Based Programs. In *9th Intl. Conf. on Compiler Construction (CC)*, Berlin, Germany, Mar. 2000.

[55] http://java.sun.com/docs/hotspot/gc1.4.2/.

[56] http://sourceware.org/gdb/.

[57] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proc. of the 1982 SIGPLAN symp. on Compiler construction*, 1982.

[58] http://www.gzip.org/.

[59] T. Haerder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2):111–120, 1984.

[60] L. Hammond, V. Wong, et al. Transactional Memory Coherence and Consistency. In *the Proc. of the 31st Intl. Symp. on Computer Architecture (ISCA)*, Munich, Germany, June 2004.

[61] R. A. Hankins, G. N. Chinya, J. D. Collins, P. H. Wang, R. Rakvic, H. Wang, and J. P. Shen. Multiple instruction stream processor. *SIGARCH Comput. Archit. News*, 34(2), 2006.

[62] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *the Proc. of the 18th Conf. on Object-oriented Programing, Systems, Languages, and Applications*, Oct. 2003.

[63] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[64] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proc. of the twenty-second Symp. on Principles of distributed computing*, pages 92–101, New York, NY, USA, July 2003. ACM Press.

[65] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *the Proc. of the 20th Intl. Symp. on Computer Architecture*, May 1993.

[66] R. L. Hudson and J. E. B. Moss. Sapphire: copying gc without stopping the world. In *JGI '01: Proc. of the 2001 joint ACM-ISCOPE conf. on Java Grande*, 2001.

[67] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. Mcrt-malloc: a scalable transactional memory allocator. In *ISMM '06: Proc. of the 5th Intl. Symp. on Memory management*, 2006.

[68] E. Iwata and K. Olukotun. Exploiting coarse-grain parallelism in the mpeg-2 algorithm, 1998.

[69] Java Grande Forum, *Java Grande Benchmark Suite*. `http://www.epcc.ed.ac.uk/javagrande/`, 2000.

[70] P. Jayanti. An optimal multi-writer snapshot algorithm. In *STOC '05: Proc. of the 37th ACM symp. on Theory of computing*, 2005.

[71] JDBC Technology. http://java.sun.com/products/jdbc/.

[72] R. Kalla, B. Sinharoy, and J. Tendler. Simultaneous multi-threading implementation in POWER5. In *Conf. Record of Hot Chips 15 Symp.*, Stanford, CA, August 2003.

[73] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution via Program Shepherding. In *the Proc. of the 11th USENIX Security Symp.*, San Francisco, CA, August 2002.

[74] P. Kongetira. A 32-way multithreaded Sparc processor. In *Conf. Record of Hot Chips 16*, Stanford, CA, August 2004.

[75] S. Kumar, M. Chu, et al. Hybrid Transactional Memory. In *the Proc. of the 11th Symp. on Principles and Practice of Parallel Programming (PPoPP)*, New York, NY, Mar. 2006.

[76] B. C. Kuszmaul and C. E. Leiserson. Transactions Everywhere. MIT Laboratory for Computer Science, Research Abstract, 2003.

[77] J. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Synthesis Series, 2007.

[78] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):874–879, 1994.

[79] D. E. Lowell and P. M. Chen. Free Transactions with Rio Vista. In *Proc. of the 16th Symp. on Operating systems principles*, Saint Malo, France, October 1997.

[80] C. Luk, R. Cohn, et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *the Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, Chicago, IL, June 2005.

[81] E. Lusk and R.Overbeek. *Portable Programs for Parallel Processors.* Holt, Rinehart and Winston, Inc., 1987.

[82] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *19th Intl. Symp. on Distributed Computing*, September 2005.

[83] A. McDonald, J. Chung, et al. Characterization of TCC on Chip-Multiprocessors. In *the Proc. of the 14th Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2005.

[84] A. McDonald, J. Chung, et al. Architectural Semantics for Practical Transactional Memory. In *the Proc. of the 33rd Intl. Symp. on Computer Architecture*, June 2006.

[85] P. McKenney and J. Slingwine. Read-Copy Update: Using Execution History to Solve Concurrency Problems. In *Parallel and Distributed Computing and Systems*, Las Vegas, NV, October 1998.

[86] C. McNairy and R. Bhatia. Montecito: The next product in the Itanium processor family. In *Conf. Record of Hot Chips 16*, Stanford, CA, August 2004.

[87] M. Milovanovic, R. Ferrer, et al. Transactional Memory and OpenMP. In *the Proc. of the Intl. Workshop on OpenMP*, June 2007.

[88] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. of The IEEE Intl. Symp. on Workload Characterization*, September 2008.

[89] M. Moir. Hybrid transactional memory. Unpublished manuscript, July 2005.

[90] K. E. Moore, J. Bobba, et al. LogTM: Log-Based Transactional Memory. In *the Proc. of the 12th Intl. Conf. on High-Performance Computer Architecture*, Feb. 2006.

[91] M. J. Moravan, J. Bobba, et al. Supporting Nested Transactional Memory in LogTM. In *the Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

[92] T. Moseley, A. Shye, V. J. Reddi, D. Grunwald, and R. Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *CGO '07: Proceedings of the Intl. Symp. on Code Generation and Optimization*, 2007.

[93] MPEG Software Simulation Group, mpeg2encode/mpeg2decode (mpeg2play) version 1.2. http://www.mpeg.org/MSSG/, 1996.

[94] J. Nakano, P. Montesinos, et al. Revivei/o: efficient handling of i/o in highly-available rollback-recovery servers. In *HPCA '06: Proc. of the 12th Intl. Symp. on High-Performance Computer Architecture*, 2006.

[95] NASA Advanced Supercomputing Parallel Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.

[96] N. Neelakantam, R. Rajwar, et al. Hardware atomicity for reliable software speculation. *SIGARCH Comput. Archit. News*, 35(2), 2007.

[97] http://memprofiler.com/.

[98] http://profiler.netbeans.org/.

[99] N. Nethercote and J. Seward. How to Shadow Every Byte of Memory Used by a Program . In *the Proc. of the 2007 ACM Intl. Conf. on Virtual Execution Environments (VEE)*, San Diego, CA, June 2007.

[100] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of the Conf. on Programming Language Design and Implementation (PLDI)*, San Diego, CA, June 2007.

[101] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE '07: Proc. of the 3rd Intl. Conf. on Virtual Execution Environments*, 2007.

[102] J. Newsome and D. X. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *the Proc. of the Network and Distributed System Security Symp. (NDSS)*, San Diego, CA, February 2005.

[103] A. Nguyen-Tuong, S. Guarnieri, et al. Automatically Hardening Web Applications using Precise Tainting. In *Proc. of the 20th IFIP Intl. Information Security Conf.*, Chiba, Japan, May 2005.

[104] G. Novark, E. Berger, and B. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *PLDI '07: ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2007.

[105] http://nullwebmail.sourceforge.net/httpd/.

[106] J. Oplinger and M. S. Lam. Enhancing software reliability with speculative threads. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 184–196, October 2002.

[107] http://www.oracle.com/technology/products/timesten/index.html.

[108] D. Z. Pan and M. A. Linton. Supporting reverse execution for parallel programs. *SIGPLAN Not.*, 24(1), 1989.

[109] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *the Proc. of the Recent Advances in Intrusion Detection Symp.*, Seattle, WA, Sept. 2005.

[110] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: transparent checkpointing under unix. In *TCON'95: Proc. of the USENIX 1995 Technical Conf.*, 1995.

[111] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[112] PowerPC Assembler Reference.

[113] Gcc extension for protecting applications from stack-smashing attacks.

[114] M. Prvulovic, Z. Zhang, and J. Torrellas. Revive: cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA '02: Proc. of the 29th Intl. Symp. on Computer Architecture*, 2002.

[115] Purify: Fast detection of memory leaks and access errors.

[116] http://pypy.org/.

[117] F. Qin, S. Lu, and Y. Zhou. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA*, 2005.

[118] F. Qin, C. Wang, et al. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *the Proc. of the 39th the Intl. Symp. on Microarchitecture (MICRO)*, Orlando, FL, December 2006.

[119] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X: Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, New York, NY, USA, October 2002. ACM Press.

[120] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *the Proc. of the 32nd Intel. Symp. on Computer Architecture (ISCA)*, Madison, WI, June 2005.

[121] M. Rebaudengo, M. S. Reorda, et al. Soft-error detection through software fault-tolerance techniques. In *Proc. of the 14th Intl. Symp. on Defect and Fault-Tolerance in VLSI Systems*, 1999.

[122] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *Proc. of the 27th Intl. Symp. on Computer Architecture*, 2000.

[123] G. A. Reis, J. Chang, D. I. August, R. Cohn, and S. S. Mukherjee. Configurable transient fault detection via dynamic binary translation. In *Proc. of the 2nd Workshop on Architectural Reliability (WAR)*, 2006.

[124] Rosetta. `http://www.apple.com/rosetta/`.

[125] B. Saha, A.-R. Adl-Tabatabai, et al. A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *the Proc. of the 11th Symp. on Principles and Practice of Parallel Programming*, Mar. 2006.

[126] B. Saha, A.-R. Adl-Tabatabai, et al. Architectural Support for Software Transactional Memory. In *the Proc. of the 39th Intl. Symp. on Microarchitecture (MICRO)*, Dec. 2006.

[127] Y. Saito. Jockey: a user-space library for record-replay debugging. In *AADEBUG'05: Proc. of the 6th Intl. Symp. on Automated analysis-driven debugging*, 2005.

[128] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight Recoverable Virtual Memory. *ACM Transactions on Computer Systems*, 12(1), 1994.

[129] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: a Low Overhead, Software-only Approach for Supporting Fine-grain Shared Memory. In *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, October 1996.

[130] K. Scott and J. Davidson. Safe Virtual Execution Using Software Dynamic Translation. In *the Proc. of the 18th Annual Computer Security Applications Conf. (ACSAC)*, Las Vegas, NV, December 2002.

[131] Security-Enhanced Linux. `http://www.nsa.gov/selinux/`.

[132] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *OSDI '96: Proc. of the 2nd USENIX Symp. on Operating Systems Design and Implementation*, 1996.

[133] J. Seward and N. Nethercote. Using Valgrind to detect Undefined Value Errors with Bit-Precision. In *the Proc. of the USENIX 2005 Annual Technical Conf.*, April 2005.

[134] N. Shavit and D. Touitou. Software transactional memory. In *Proc. of the 14th ACM Symp. on Principles of Distributed Computing*, pages 204–213, August 1995.

[135] T. Shpeisman, V. Menon, et al. Enforcing Isolation and Ordering in STM. In *the Proc. of the Conf. on Programming Language Design and Implementation*, Mar. 2007.

[136] A. Shriraman, M. Spear, et al. An Integrated Hardware-Software Approach to Flexible Transactional Memory. In *the Proc. of the 34th Intl. Symp. on Computer Architecture (ISCA)*, San Diego, CA, June 2007.

[137] J. M. Smith and G. Q. Maguire, Jr. Effects of copy-on-write memory management on the response time of UNIX fork operations. *Computing Systems*, 1(3):255–278, 1988.

[138] D. J. Sorin, M. M. K. Martin, et al. Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. of the 29th Intl. Symp. on Computer Architecture*. May 2002.

[139] SourceForge.net. http://sourceforge.net/.

[140] Standard Performance Evaluation Corporation, *SPECjbb2000 Benchmark*. `http://www.spec.org/jbb2000/`, 2000.

[141] Standard Performance Evaluation Corporation, *SPEC OpenMP Benchmark Suite*. `http://www.spec.org/omp`.

[142] S. Sridhar, J. Shapiro, et al. HDTrans: an Open Source, Low-Level Dynamic Instrumentation System. In *the Proc. of the ACM/USENIX Intl. Conf. on Virtual Execution Environments (VEE)*, Ottawa, ON, June 2006.

[143] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-write Network. In *Proc. of the 16th Symp. on Operating Systems Principles*, Saint Malo, France, 1997.

[144] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *Proc. of the 19th Symp. on Operating Systems Principles*, 2003.

[145] S. Thrun, D. Fox, W. Burgard, and F. Dellaert. Robust Monte Carlo Localization for Mobile Robots. *Artificial Intelligence*, 128(1-2):99–141, 2001.

[146] http://user-mode-linux.sourceforge.net/.

[147] VMware. `http://www.vmware.com/`.

[148] http://w3m.sourceforge.net/.

[149] R. Wahbe. Efficient data breakpoints. In *ASPLOS-V: Proc. of the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1992.

[150] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP '93: Proc. of the 14th ACM Symp. on Operating Systems Principles*, 1993.

[151] C. A. Waldspurger. Memory Resource Management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[152] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *CGO '07: Proceedings of the Intl. Symp. on Code Generation and Optimization*, 2007.

[153] J. H. Wharton. Intel 80960ca superscalar microprocessor. 1992.

[154] E. Witchel, J. Cates, and K. Asanovic. Mondrian memory protection. In *ASPLOS-X: Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2002.

[155] S. C. Woo, M. Ohara, et al. The SPLASH2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, Santa Margherita, Italy, June 1995.

[156] M. Wu and X.-F. Li. Task-pushing: a scalable parallel gc marking algorithm without synchronization operations. In *IPDPS '07: Proc. of intl. Parallel and Distributed Processing Symposium*, 2007.

[157] W. Xu, S. Bhatkar, and R. Sekar. Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks. In *the Proc. of the 15th USENIX Security Conf.*, Vancouver, Canada, Aug. 2006.

[158] K. C. Yeager. The mips r10000 superscalar microprocessor. *IEEE Micro*, 16(2), 1996.

[159] Q. Zhao, S. Amarasinghe, R. M. Rabbah, L. Rudolph, and W. F. Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. In *Proc. of 17th Intl. Conf. of Compiler Construction*, 2008.

[160] P. Zhou, F. Qin, et al. iwatcher: Efficient architectural support for software debugging. In *ISCA '04: Proc. of the 31st Intl. Symp. on Computer Architecture*, 2004.