TRANSACTIONAL PROGRAMMABILITY AND PERFORMANCE

BY

LEE W. BAUGH
B.S. University of Texas at Dallas, 1997
M.S. University of Texas at Dallas, 2000

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2008

Urbana, Illinois

Doctoral Committee:
Assistant Professor Craig Zilles, Chair
Professor Sarita V. Adve
Associate Professor Steven S. Lumetta
Ravi Rajwar, Intel Corporation
Professor Josep Torrellas
Abstract

Transactional memory is a promising technique for multithreaded synchronization and concurrency which has attracted much interest in recent years. Compared to locks, transactions can offer increased concurrency, composability, and deadlock freedom; yet their speculative nature introduces problems not seen in lock-based code, and these have limited the adoption of transactions as a serious alternative to locks. Specifically, it is not clear how transactions can contain side-effecting actions like I/O, nor how they can scale to arbitrary footprints and durations while retaining acceptable performance and a consistent and desirable programming semantic.

I believe that addressing problems like these is of first importance to programmers considering using transactions. In this work, I seek to extend the programmability of transactions by exploring and addressing these problems. I begin by exploring the domain of hardware support for software transactional memory. I first show how a general-purpose user-mode memory protection system can be used to provide strong isolation to software transactional memory systems with little hardware and minimal performance overhead. Then I show how to augment this strongly atomic software transactional memory system with a ‘best-effort’ hardware transactional memory system to yield a hybrid transactional memory system, the UFO Hybrid, in which high-performance hardware transactions can run concurrently with large, long, nested, or side-effecting software transactions. Compared to previous hybrid transactional memory proposals, my system offers a strongly atomic programming model for both its hardware and its software transactions, low-overhead conflict detection between its hardware and software transactions, and high HTM performance even in the presence
of potentially-conflicting software transactions. I show through experimental analysis that
the UFO Hybrid performs at least as well as competing hybrid transactional memory sys-
tems, and significantly outperforms them when some transactions are forced to fail over to
software.

I also provide an analysis of I/O in lock-based critical sections in large multithreaded
workloads, with observations on how this I/O could be performed in transactional code. In
this analysis, I find that no one of the previously proposed techniques is by itself sufficient for
handling side-effects without sacrificing performance. However, I conclude that the majority
of transactional I/O is likely to be compensatable, and that causing transactions performing
I/O to ‘go nonspeculative’ can be a reasonable choice for uncompensated I/O, provided that
it does not delay non-side effecting transactions.
To my parents, James and Deborah Baugh
Acknowledgements

When, in 2002, I took my Qualifying exam, Dr. Josep Torrellas asked me afterwards, “How did it go?” I replied that I was unsure, adding “You guys are so smart!” “No, no, Lee,” he replied, “We’re just more experienced. In five years, you’ll be there too.” It’s taken six years, and I am quite reluctant to rate myself equal to the exceptional members of my qual committee, but his comment was nonetheless accurate. The Ph.D is a transforming experience, converting those promising students who stick with it into capable, diligent, and resourceful researchers, and no one has been more instrumental to my transformation than my advisor, Craig Zilles. His virtues as a researcher exceed, I fear, my ability to match, but his example – in identifying the essential promise of an idea, in isolating and presenting the correct data to support one’s findings, in crafting a paper or presentation to answer more questions than it raises, and in working hard to get good results – has given me something to strive for, both during my Ph.D work as well as in the work which will come after.

I would like to express my appreciation to my committee for their help along the way. Steve Lumetta has generously shared his time to test my work and discuss its direction; Ravi Rajwar’s difficult questions have helped me produce a stronger dissertation and his support was essential for the work presented in Chapter 3. However, if each year of a Ph.D rests on the years previous to it, then I owe my completion of this degree to Josep Torrellas, who went out of his way to ensure that I felt welcome in the department and that I had the early support I needed.

Computer architecture at UIUC is fortunate to have a thriving community of bright, enthusiastic graduate students, both in CS and ECE. The open atmosphere encourages
communication, and everyone participating benefits from the reading group, practice talks, and other productive communications. Two research groups in particular could always be relied upon to support me in my own explorations and preparations: the i-acoma group, whose members are always available for proofreading, criticism, technical discussions, and moral support; and my own advisor's MSSP group. Of course I enjoyed years of mutual support with the first generation of MSSP students: Pierre Salverda, Naveen Neelakantam, Nicholas Riley, and myself, but the group is now growing, and in good directions. Craig’s guidance has built a strong and diverse group, and one with quite high-minded research principles.

I would also like to acknowledge the support of the CS department, and its staff: particularly Barb Cicone, who was on my side early on; Mary Beth Kelley, who is on my side at the end, and Sheila Clark, who solved so many problems for me during my years with Craig.

Finally, I must acknowledge the profound impact of the support of my friends and family. While there are too many to show here, I’d like to particularly acknowledge, for the direct support they lent this dissertation at various times of my graduate student career, my friends Brian Greskamp, Pierre Salverda, James Tuck, Derek Gottlieb, and Jeff Cook. Michelle Urberg has been a tremendous support in the difficult last months of this Ph.D. But the greatest share of credit must go to my parents, James and Deborah Baugh. Their unstinting support of me – without requiring I follow any particular path – has allowed me to reach this point on my own terms and without resentment. Together, they are uniquely supportive: insightful and enthusiastic in turn; and there is no question that without their powerful presence this document, and my Ph.D, would not exist.
# Table of Contents

List of Tables ................................................................. ix
List of Figures ................................................................. x
Preface ................................................................. 1

Chapter 1 Introduction ......................................................... 15
  1.1 Limitation of Scope .................................................... 17
  1.1.1 Shared Memory Parallel Programming ............................. 17
  1.1.2 Transactional Memory ................................................. 18
  1.1.3 Transactional Memory and Memory Consistency .................. 18
  1.1.4 Transactional Workloads ............................................. 19
  1.2 Transactional Memory Background ..................................... 19
  1.2.1 Transactional Memory Design Space .............................. 20
  1.3 Summary of the Dissertation ......................................... 24

Chapter 2 The UFO Hybrid ................................................... 28
  2.1 Introduction ............................................................. 29
  2.2 Motivation ............................................................... 31
    2.2.1 High Performance .................................................. 31
    2.2.2 A Clean and Complete Programming Model ...................... 31
    2.2.3 The Role of Hardware ............................................. 37
  2.3 Hardware Primitives ................................................... 38
    2.3.1 BTM: a “Best-effort” Hardware TM ............................... 38
    2.3.2 UFO: User-mode Fine-grained Hardware Memory Protection .... 40
  2.4 System Organization ................................................... 43
    2.4.1 USTM: Building a Strongly-Atomic STM ......................... 43
    2.4.2 Making USTM Strongly Atomic ................................... 47
    2.4.3 Bringing It Together: The UFO Hybrid .......................... 48
    2.4.4 Contention Management in a Hybrid TM ........................ 52
  2.5 Performance Analysis ................................................ 54
    2.5.1 Experimental Method .............................................. 55
    2.5.2 Results ............................................................. 58
    2.5.3 Speculative Footprint ............................................. 62
    2.5.4 Software Failover ................................................. 63
2.5.5 Sensitivity Analysis ................................................. 67
2.5.6 Using HTM to Support an STM System ....................... 68
2.6 Concluding Remarks .................................................... 71
  2.6.1 Hardware TM Policies .......................................... 73
  2.6.2 Advanced Transactional Semantics ............................. 75

Chapter 3 I/O in Critical Sections ........................................ 77
  3.1 Introduction ........................................................... 78
  3.2 Types of Actions ..................................................... 79
  3.3 Proposals for Handling Side-Effects in Transactions ............ 80
  3.4 Side-Effects in Critical Sections .................................. 83
    3.4.1 Experimental Method .......................................... 85
  3.5 IO Results ............................................................ 87
    3.5.1 Syscall Sites ................................................... 87
    3.5.2 Syscall Protection – The Advantage of Compensation Code .... 88
    3.5.3 Syscall Context ................................................ 91
    3.5.4 The Applicability of Deferral ................................ 92
    3.5.5 The Overhead of “Going Nonspeculative” ..................... 93
  3.6 Concluding Remarks ................................................ 95

Chapter 4 Conclusion ....................................................... 96

References ................................................................. 99

Author’s Biography ....................................................... 106
List of Tables

1.1 Some Transactional Memory Proposals and Their Salient Properties  27

2.1 The BTM ISA Extension ............................................. 39
2.2 UFO ISA Extensions .................................................. 40
2.3 The USTM API .......................................................... 44
2.4 Simulation parameters ................................................ 56

3.1 Syscalls Seen in Critical Sections S ................................. 87
List of Figures

1. A Race Condition in a Two-threaded Application. ........................................ 6
2. Atomic Regions ........................................................................................................ 7
3. Lock-based Critical Sections .................................................................................. 8
4. Deadlock .................................................................................................................. 9
5. Privatization ............................................................................................................. 11

2.1 Using Fine-grained Memory Protection to Achieve a Strongly-atomic, Efficient Hybrid Transactional Memory System ......................................................... 29
2.2 A Transactional Containment Violation in an Eager-Versioning Transactional Memory System ................................................................. 32
2.3 A Transactional Non-Interference Violation .............................................................. 33
2.4 Privatization Leading to Inconsistency in an Early-Versioning Transactional Memory System Lacking Strong Atomicity .......................... 34
2.5 A Lost Non-Transactional Write In a Word-Granularity, Eager-Versioning Transactional Memory System Lacking Strong Atomicity .......................... 35
2.6 The Memory Hierarchy with UFO Support ................................................................. 42
2.7 The USTM Shared Ownership Table (otable) .......................................................... 45
2.8 Code Example of a USTM Write Barrier ................................................................. 46
2.9 A USTM Write Barrier with Strong Isolation via UFO .......................................... 48
2.10 A UFO Fault Handler ............................................................................................. 49
2.11 Hybrid TM code. ....................................................................................................... 50
2.12 The BTM Abort Handler .......................................................................................... 52
2.13 STAMP Performance of Several TM Schemes ......................................................... 57
2.14 Hardware Abort Reasons ....................................................................................... 57
2.15 A Problematic Transaction in labyrinth ................................................................ 61
2.16 Range of Speculative Cache Footprints ................................................................. 62
2.17 Victim Caches Can Avert Set Overflows ............................................................... 64
2.18 Software Failover Limit ......................................................................................... 65
2.19 Failover Limit Detail for PhTM as the Amount of Work Done in Each Transaction Is Varied ............................................................ 66
2.20 Sensitivity Analysis ................................................................................................. 67
2.21 Code Example of a USTM Write Barrier using BTM for Atomic otable Updates .......................................................................................... 70
2.22 Abort Rates of BTM Transactions in USTM Execution ........................................... 72
2.23 Reasons for BTM Aborts in USTM Execution ......................................................... 72
3.1 A Syscalling-TCS Overlapping with TCSs .......................... 86
3.2 Distribution of Syscalls across Syscalling-TCSs ...................... 88
3.3 Distribution of Syscalling-TCS Durations (in µsec) .................. 90
3.4 Distribution of First Syscalls across Critical Sections ............... 91
3.5 Degree of Overlap of Toplevel Syscalling Critical Sections (syscalling-TCSs) and Toplevel Critical Sections (TCSs) ......................... 93
Preface

Parallel programming – the technique of dividing a computational task into separate units which may be simultaneously executed using independent hardware resources – has long been an important method for improving program performance. Until recently, parallel programming has been largely constrained to applications in scientific and enterprise computing. In these domains, the value of solving a computational problem quickly offsets not only the cost of greater hardware resources, but also of the labor-intensive process of efficiently adapting the problem to the available resources. This adaptation, an expert process requiring detailed knowledge of both the computational problem and the capabilities and limitations of the hardware, seeks not only to improve the code’s performance on a single CPU, but also to partition the code into multiple independent threads which could be executed simultaneously on different processing units. Consumer computing, however, rarely received such parallelization efforts: the economic incentives were much less, the underlying hardware systems too numerous and varied, and in any case very few consumers had access to multiprocessor systems.

Computing has enjoyed a long trend of single-threaded performance improvement. The increasing transistor count promised by Moore’s law supported more advanced superscalar designs, increasing issue width and pipeline depth, and the scaling-down of individual devices permitted increasing clock rates. Recently, however, a number of factors have contributed to the slowing of this trend, including

- **The ‘ILP Wall’**: Programs tend only to expose a limited amount of instruction-level parallelism. Accordingly, increasing processor issue width meets with diminishing
returns at the price of greatly increased hardware complexity.

- **The ‘Memory Wall’**: Memory system performance has not scaled at the same rate as computational performance, so the cost of memory operations relative to the cost of other operations has increased. Accordingly, single-threaded performance is increasingly memory-bound.

- **Power and Heat**: Both the increasing clock rate and the decreasing size of transistors have greatly increased the power density of modern CPUs, to the degree that heat management has become a limiting factor in CPU designs. Drastic increases in frequency (without complex cooling systems) are unlikely.

These factors, combined, appear to spell the end of the steady and significant single-threaded performance improvement which consumers (and application developers) have come to expect. Indeed, processor manufacturers have largely abandoned grand attempts to improve single-core performance, instead offering more cores per chip – so-called multicore architectures – and focusing on shared memory system designs.

These new multicore architectures do not pose a significant problem to developers of scientific and enterprise applications – already used to parallel programming, they need simply to learn how best to take advantage of the new processors. There is, however, a large contingent of developers with little or no experience in parallel programming. Multicore architectures require such developers, grown used to a steady increase in single-threaded performance, to look towards parallelism in order to continue to obtain the performance improvement (or increase in features) that their applications have enjoyed. We find ourselves, then, at a point when parallel programming – particularly shared-memory parallelism – is becoming a very important technique to a much wider variety of software developers.
Shared-memory parallel programming

There are a number of ways to extract parallelism from an algorithm.[46] The best or most natural choice depends on a combination of factors: available libraries, languages, or hardware; the expected value of parallelization; or programmer ability; but most of all it depends on the algorithm itself. Different applications may be more or less suited to parallelization, and different parallelization approaches are more or less suited to different synchronization mechanisms. Some tasks can be trivially and effectively parallelized – in ray tracing, for example, each ray can be computed independently of all other rays. Some tasks can be parallelized, but only with difficulty, or with limited concurrency returns. Other tasks are very difficult to parallelize, either because the problem simply does not expose any parallelism, or because it is very difficult to rule out the possibility of contention on shared data.

In the taxonomy of parallel programming techniques, the broadest division is between message-passing techniques and shared-memory techniques. Message-passing assumes a private memory space for each thread, processor, or node; communication between threads is performed by explicitly sending and receiving messages. This approach is generally used in large, distributed-memory systems. Since I am exploring parallel programming for multicore architectures, I will focus on shared-memory synchronization. In shared-memory synchronization, threads share a common region of memory, and the programmer must ensure that mutual exclusion on shared variables is enforced. Broadly speaking, there are two ways to ensure mutual exclusion: by phasic means such as barriers, or by mutexes like locks.

Phasic synchronization guarantees that shared data is not exposed to race conditions by dividing parallel work into phases. For example, parallelizations which decompose algorithms in a pipelined or systolic fashion might ensure that in any given phase (or pipeline stage), any given shared variable is only accessed by one thread, and are handed off between threads only at phase boundaries. Barriers permit programmers to define these phases by specifying synchronization points in their code. A given barrier is initialized to support $N$ threads;
it will then block the first $N - 1$ threads to wait on it. When the $N^{th}$ thread arrives at the barrier, the waiting threads are all set free and the barrier is reset. In addition to implementing hand-off points in systolic or pipelined algorithms, barriers are useful for coordinating phases in Single Program Multiple Data (SPMD) parallelism. If an algorithm is suited to these kinds of decomposition, no other synchronization may be necessary, if data shared between threads may be proven never to be shared concurrently during a phase.

As a shared-memory synchronization mechanism, barriers have much to commend them. Under barrier-based synchronization, it is very easy for programmers to reason about what can occur concurrently. For many algorithms, pipelining is a relatively simple parallelization, compared to other kinds of parallel decompositions. Also, and importantly, implementing barriers in hardware efficiently is a well-understood problem, so that barriers can be a low-overhead synchronization option.

However, not all algorithms are well-suited to shared-memory parallelization using barriers alone. Mutexes offer a more generic approach to synchronization by allowing programmers to designate regions of code (called critical sections) which can only be executed by one thread at a time. There are several flavors of mutex, including:

- **Semaphores**, which control access to shared resources. A semaphore is lowered when mutual exclusion on the resource it protects is desired, and raised when the work on the resource is done. A thread attempting to lower a semaphore which is already held is blocked until the semaphore is raised again. Semaphores apply naturally to process waiting, shared resource protection, and producer-consumer relationships, and so are commonly used in operating systems.

- **Monitors**, a language-level synchronization construct which provide a form of implicit synchronization by associating shared data with a set of functions permitted to act on the data. Functions belonging to the same monitor execute under mutual exclusion\(^1\),

\(^1\)Monitor functions may wait; multiple monitor functions may be in flight, but only if at most one of them is not waiting.
and the monitor data may not be shared save through the monitor functions. Monitors are particularly well suited to synchronizing operations to shared objects.

- **Locks**, which, like semaphores, permit users to define regions of mutual exclusion. A thread *acquires* a lock when it wishes to perform some action on shared state or resources, and *releases* the lock when it is done. An attempt to acquire a lock which is already held either blocks until the lock is available or fails, depending on the kind of lock used. Because of their generality (barriers, semaphores, and monitors are frequently implemented using locks) and the widespread support in ISAs for lock acquires and releases, locks are perhaps the most common synchronization mechanism in shared-memory applications.

Mutex-based parallel programming is a notoriously difficult endeavor. Parallel programs are subject to new classes of bugs not present in single-threaded programs, such as deadlock and race conditions. Moreover, even a correct parallel program, if poorly parallelized, may not yield any performance improvements over single-threaded execution. Partitioning a program into parallel threads effectively and correctly is a difficult enough task, but the way parallelism is expressed – and in particular, the way accesses to data shared between parallel threads are handled – itself introduces a number of challenges and pitfalls to parallel programmers.

**What mutex programmers want**

**Race freedom**

Parallelizing a program for a shared-memory architecture involves two tasks: partitioning the program into threads, and then protecting operations on resources (if any) shared between two or more of these threads. An example of the need for this protection is shown in Figure 1.
counter has been incremented twice, but due to unfortunate interleaving of the threads, at the end of the sample it contains 1 rather than 2.

In this figure, an unfortunate interleaving of operations from the two threads has resulted in an invalid state: counter has been incremented twice, but appears to have been incremented only once. Such race conditions result from the two threads running concurrently; without some way of grouping the read, modify, and write operations performed by each thread, the programmer cannot achieve race freedom.

Atomic Regions

Central to this grouping are the concepts of atomicity and isolation. Briefly, a group of operations are performed atomically if they are either all performed, or none of them is performed. A group of operations is performed in isolation if no other thread views any intermediate state in the group – that is, from the perspective of other threads, it either

---

2This terminology is borrowed from the transactional database literature, as in Gray and Reuter [25], and will be used (and described in greater detail) throughout this dissertation.
Figure 2: Atomic Regions
Atomic regions such as these avoid the race condition seen in Figure 1.

appears that none of the operations in the group was performed, or all of them were. Taken together, atomicity and isolation permit programmers to safely perform groups of operations on data shared between threads.

A parallel programming paradigm for shared memory must support, explicitly or implicitly, some form of atomic regions (or critical sections). In Figure 3, the read, modify, and write operations have been grouped together in atomic regions which are executed atomically and in isolation. Regardless of thread interleaving, the final value (and any intermediate values) of counter is correct.

Concurrency Expression

While correct, the atomic regions in Figure 3 may be too restrictive. While the isolation property is satisfied by interleaving all atomic regions, this approach may unnecessarily
serialize atomic regions which do not share any state, resulting in a loss of performance. Accordingly, parallel programming techniques for shared memory permit programmers to express, implicitly or explicitly, concurrency in their atomic regions.

Locks, for example, permit programmers to express concurrency by guarding different resources or regions of shared memory by different locks. Since a thread must hold the relevant lock before it can execute any of a given critical section, and only one thread may hold a given lock at a time, atomicity and isolation are guaranteed for a well-formed program; however, two critical sections guarded by different locks are free to execute concurrently, as shown in Figure 3. However, it is the responsibility of the programmer to ensure that every access to a given shared resource is guarded by the same lock; otherwise isolation and atomicity are not guaranteed.

\footnote{That is, a program for which each lock protects a unique region of shared memory, and each region of shared memory is protected by one lock.}
Deadlock Avoidance

Critical sections may be nested; when a programmer wishes to atomically access shared resources guarded by more than one lock, all relevant locks may be acquired, the atomic accesses performed, then all the locks may be released. However, if critical sections can execute concurrently, this raises the danger of deadlock, the situation in which a series of threads form a circular chain of dependences upon locks. Figure 4 shows an example of this situation. In this figure, Thread 1 has acquired lock1 and is waiting on lock2; meanwhile, Thread 2 has acquired lock2 and is waiting on lock1. lock1 will not be released until Thread 1 has completed, which is causally dependent on its acquisition of lock2; a similar situation prevents Thread 2’s releasing lock2. Neither thread may make forward progress.

Deadlock is a significant source of bugs in parallel programs. To avoid it requires intimate knowledge of the locking behavior of the whole program, to ensure a consistent partial ordering of lock acquires. The cost of such bugs, or of the design effort to avoid them, makes concurrent programming techniques which avoid or limit deadlock desirable.
Forward Progress and Starvation

Programmers could, with considerable effort, seek to avoid deadlock by releasing all contended locks if some time has elapsed without successfully acquiring a needed lock. In addition to its complexity, however, this approach risks livelock: a phenomenon in which, while work is being done – here, the acquire and release of locks, whole-program forward progress is not being made.

If, on the other hand, a deadlock detection and recovery system only releases the locks held by one of the deadlocking threads, livelock can be avoided – but at the risk of starvation, or lack of forward progress on one thread. While such starvation does not affect correctness – eventually, other threads will exhaust their work and release the resources the starving thread needs – it can affect performance by limiting the work done by the starving thread, and thus reducing the parallelism of the application.

A good shared-memory concurrent programming method should help programmers avoid livelock and starvation.

Atomic Region Composition

Modular programming, and the separation of concerns, encourages programmers to use modules or libraries as ‘black boxes’, about which little is known except for what the module requires for input and produces as output. However, this technique – widely considered crucial for effective software engineering – directly contravenes the requirement that programmers of lock-based code maintain a whole-program awareness of lock acquires and releases in order to avoid deadlock. In Figure 4, Thread 1’s acquire to lock2 could be in a library call, as could Thread 2’s acquire to lock1; in this case, code which at the context of the outer critical section appears deadlock-free may still deadlock.

Deadlock through composition is not the only pitfall of this approach. For concurrency schemes which allow atomic regions to execute speculatively, composing code with side-
effects into an atomic region can cause problems. As composition is fundamental to modular
programming, concurrent programming techniques which promote it are desirable.

Privatization

Privatization ‘unshares’ a shared variable so that it can be worked on privately, without
need for atomicity or isolation.

Protecting shared data in atomic regions incurs overhead through the ‘opportunity cost’
of limited concurrency. For this reason, a high-performance concurrent programming tech-
nique should provide a way to privatize shared data – to remove it from sharing, and from
the need to protect access to it in an atomic region.

Figure 5 gives an example of privatization; in it, both threads 1 and 2 atomically remove
an element from a shared linked list, but then perform work on the element privately, without
needing to operate on it atomically or in isolation.

Transactional Memory

Historically, the dominant form of synchronization for application-level (as opposed to system-
level) parallel programming has been locks. This is in part due to their generality – more
sophisticated synchronization mechanisms may be efficiently built from locks – and in part to the ubiquity of support for lock acquires and releases among many diverse architectures. However, lock-based parallel programming is extremely challenging due to its risks of deadlock and livelock, and the difficulty of abstracting lock-based critical sections to enable easy composition.

The source of these difficulties stems from the requirement to associate a specific lock, or set of specific locks, with every critical section. Transactional memory, in contrast, permits programmers to specify transactions, or atomic regions unassociated with a specific lock resource. To permit concurrent execution of several transactions, transactions execute speculatively: transactional memory systems observe memory accesses within transactions, and should a conflict between two concurrent transactions arise, one of them is aborted.

Transactional memory has many advantages over lock-based synchronization, which have made it the target of a considerable amount of research scrutiny in recent years. Transactional programmers can avoid race conditions by simply marking atomic regions as such. The degree to which transactions may execute concurrently is in theory limited only by the contention they experience. While deadlock is still possible under transactions\(^4\), it is extremely unlikely to arise in most transactional codes. Accordingly, transactions offer a more modular approach to composing atomic regions than do locks. These benefits, taken together, seem to indicate that system- and language-level support for transactional memory may see the technique employed in a new generation of parallelized applications.

These advantages have spurred considerable research interest into transactional memory in recent years, but current TM proposals still fall short of programmer expectations – at the time of this writing, no major multithreaded applications using transactional memory have been demonstrated. I believe that transactional memory’s acceptance is hindered by two factors: its performance, and its programmability. Specifically,

\(^4\)Deadlock in transactional code is trivially demonstrable by observing that transactions may be used to implement locks.
• Several proposals supporting transactional memory in hardware exist; however, without considerable evidence that programmers consider transactional memory valuable, processor firms are very unlikely to spend design effort and chip resources supporting it. Moreover, it is not yet clear that the correct transactional semantics have been established; this threatens to obsolete hardware transactional memories which attempt to provide a complete transactional solution.

• Transactional memory systems implemented (as a library or a language construct) in software can be easily deployed, and may be easily changed to adapt to developing transactional semantics. However, due to the overhead of transactional conflict detection and transactional data version control, these software systems tend to suffer from poor performance. Moreover, many such software transactional memory systems are prone to subtle race conditions created by a lack of strong atomicity. These race conditions can, for example, affect programmers’ ability to privatize data. Strongly atomic STMs have been proposed, but their strong atomicity comes at significant overhead.

• Hybrid transactional memory systems combining limited hardware support for transactions with a software transactional memory system have been proposed. Such systems attempt to achieve good performance by executing most transactions in hardware, but to permit arbitrary transactions and flexible semantics by allowing transactions unable to complete in hardware to ‘fail over’ to software. These hybrid systems, however, tend to perform poorly whenever any transaction is running in software, and they are generally prone to the same lack of strong atomicity as STMs.

• The concurrency benefits of transactional memory stem from the speculative nature of transactions. Transactions, however, only extend isolation and atomicity to accesses to shared memory within them; side effects such as I/O are neither performed in isolation, nor are they undone in the case of a transactional abort. This has an effect upon transactional composability; if a programmer unwittingly composes a module
which performs side-effects within a transaction, the program’s consistency may become compromised.

Research Goals

I believe that before it sees acceptance by parallel programmers, transactional memory will need to address the aforementioned problems with transactional programmability and performance. This dissertation aims to do this in two ways:

- First, addressing the performance, programmability, and hardware complexity shortcomings of previous transactional memory proposals, I propose a new hybrid transactional memory system which couples a strongly-atomic software transactional memory system with best-effort hardware support for transactional memory. Uniquely, this hybrid system, called the UFO Hybrid, can run hardware transactions at full speed even while potentially-conflicting software transactions are in flight. Furthermore, the strong atomicity of both the hardware and software components of the UFO Hybrid permit a consistent, sensible programming model to its users. Through performance analysis of my implementation of the UFO Hybrid, I explore the space of policies for hardware conflict resolution, software conflict resolution, and software failover policy.

- Second, addressing the programmability challenges posed by side-effects in transactions, I offer an analysis of side effects in critical sections and their ramifications on transactional memory. Through profiling the (lock-based) critical sections in large, multithreaded applications, I show that programmers do use I/O in critical sections, and not merely to provide mutual exclusion on external resources like files. I then analyze the kinds of I/O used in critical sections and the kinds of critical sections which perform I/O, and these data to reason about the applicability and performance of the proposed techniques for handling side-effects in transactions.
Chapter 1
Introduction

As with other synchronization schemes, transactional memory has both a semantic component which concerns the behavior programmers can expect from transactions, and an implementation component which concerns the mechanisms, hardware or software, by which transactional support is provided. A good transactional memory system design must address both of these components, as a poor choice of transactional semantics will compromise the ease and clarity of transactional programming on the system, and a poor implementation may compromise not only the performance of the system, but even its ability to support reasonable transactional semantics. Programmers are unlikely to use a transactional memory system whose behavior is unintuitive or difficult to reason about, or a system whose performance is poor.

In this dissertation, I will be exploring transactional memory from both a semantic and an implementation perspective, addressing both transactional programmability and performance. The key questions motivating this dissertation include:

- **What simple and flexible hardware and software mechanisms can provide strong atomicity (Section 1.2.1) to a software or hybrid transactional memory system without requiring nontransactional code to be instrumented?**

- **What simple hardware can be used to accelerate the performance of a software or hybrid transactional memory system?**

- **Will support for side-effects such as I/O in transactions be required? If so, what kind of support?**
Addressing these questions, the key contributions of this dissertation include:

- I show that hardware memory protection may be used to make a strongly atomic software transactional memory which does not require instrumentation or recompilation of nontransactional code. (Section 2.4.1)

- I show that such a strongly atomic software transactional memory system may be paired with a best-effort hardware transactional memory system to produce a high-performance hybrid transactional memory system, the UFO hybrid, which permits hardware transactions to run uninstrumented, at full speed, even while potentially-conflicting software transactions are in flight. (Section 2.4.3)

- Using the STAMP benchmarks [48, 49], I compare the UFO Hybrid against previous hybrid transactional memory schemes, and show that it offers superior performance when software transactions are in flight. (Section 2.5.2)

- Using the STAMP benchmarks to measure performance, I consider a number of variations of this hybrid’s hardware conflict resolution policy, its software conflict resolution policy, and its software failover policy, determining which design decisions have a significant performance or forward progress impact. (Section 2.5.5)

- I give an analysis of I/O in lock-based critical sections in two large, modern, multi-threaded workloads, characterizing the I/O that occurs in critical sections by kind, frequency, and distribution within the critical section, as well as characterizing the critical sections which contain I/O. Using this information, I gauge the feasibility and likely efficacy of the proposed techniques for handling I/O and side-effects within transactions. (Chapter 3)

In this Introduction, I first define the scope with which this dissertation is concerned (Section 1.1). Then, in Section 1.2, I review the historical context, key concepts, and related
works of this dissertation. Finally, in Section 1.3, I summarize this dissertation, emphasizing my findings and results.

1.1 Limitation of Scope

The field of parallel programming is enormously varied. The exigencies of distributed system design have spawned a number of parallel programming paradigms, supporting architectures with varying interprocessor or interprocess communication costs, with varying arrangements of memory, or supporting different synchronization primitives. Different problems may be parallelized in different ways rendering them more or less suitable to different parallel architectures. Many scientific workloads, like protein folding, can benefit from computing power far in excess of the ability of a single shared-memory computer to provide – Stanford’s Folding@Home [38], for example, can claim the world’s most powerful computing cluster, at any time consisting of tens or hundreds of thousands of PC clients working in parallel; while the cost of interprocess communication in this case is enormously high, such communication is quite rare because the work can be effectively partitioned into numerous large, independent tasks. Parallel programming for distributed-memory machines, or the architecture of those machines, is not germane to transactional memory, and so does not fall under the purview of this dissertation.

1.1.1 Shared Memory Parallel Programming

Other problems are not so easily parallelized, or they don’t offer as much to be gained from so much parallelism. For such problems – much more in the scope of most programmers than highly-parallel high performance computing code – the shared memory paradigm is a natural fit. A number of mechanisms exist for synchronization and concurrency in a shared-memory system, including not only transactional memory but also semaphores, locks, monitors, and barriers, each lending itself to different hardware configurations and different parallelizations.
Moreover, a barrier parallelization of a given code may be fundamentally different than a lock parallelization of the same code, and both may differ entirely from a transactional parallelization of the code. My interest in this dissertation is not to compare different ways of extracting parallelism from an application, but to explore transactional memory. For this reason, I do not quantitatively compare transactional memory to any other synchronization paradigm.

### 1.1.2 Transactional Memory

Transactional memory is not itself a monolithic technique. Section 1.2.1 describes a number of design parameters which characterize any given transactional memory system. Though I will examine some of them, like the transactional conflict resolution policy, I do not purpose to provide an exhaustive analysis of each of the design choices (some of which has been done by prior work, e.g. Bobba et al. [12]), but instead to fix a congenial set of design parameters (in this case, dictated by hardware) and use it as a platform for study and improvement.

### 1.1.3 Transactional Memory and Memory Consistency

Transactional memory’s interaction with various memory consistency models is not yet completely clear. As software TMs generally depend upon mutexes to access and update their ownership structures, their interaction with memory consistency generally devolves to their constituent mutexes’ interaction with memory consistency, which is typically well understood. However, just as mutexes must take into account the underlying hardware’s memory consistency model, so must hardware transactional memory. The aim of this dissertation is not to explore the interaction of HTM and memory consistency models or to formalize the consistency requirements of particular HTM implementations. Accordingly, throughout this dissertation, except where otherwise noted, I assume a sequential consistency model.
1.1.4 Transactional Workloads

The lack of availability of diverse suitable transactional workloads is a significant lacuna in the field. The problem merits, and will no doubt receive, much research attention: but not, however, in this dissertation. The Stanford STAMP benchmark suite [48, 49], including a number of transactional small applications and execution kernels, has been made available for public use. In this dissertation, I neither examine techniques for parallelizing applications using transactional memory nor do I characterize such applications themselves; instead I use the STAMP benchmarks, with modifications to support hybrid execution, to compare transactional memory implementations.

1.2 Transactional Memory Background

Transactional memory developed from several historical trends. Databases have long supported transactions to perform work atomically [25], and a mechanism in the IBM 801 extended hardware support for database transactions using perhaps the first hardware transactional memory [15]. The ACI properties of transactional memory – Atomicity, Consistency, Isolation – are derived from database transaction literature, as are the terms for transactional actions which I use in Section 3.2. Eswaran et al. [22] introduced the concepts of two-phased locking and well-formed transactions, which modern transactional memory systems still employ.

Some very early work on the semantics of atomicity was done by Lomet [40], who developed an atomic programming paradigm which, though it did not support concurrency with dynamic conflict detection or resolution, did explore a number of concepts related to atomic semantics, including composition, transactional waiting, explicit aborting, and strong atomicity.

Early research on modern transactional memory, including the papers which coined the terms transactional memory [33] and software transactional memory [67], was largely in pur-
suit of non-blocking shared data structures – that is, shared data structures which could be
operated on atomically, but which, unlike locks, could not be tied up indefinitely by threads
which then get swapped out. To this end, early transactional memory systems were presented
as multi-word load linked/store conditional or compare-and-swap operations.

With the recent dominance of multicore processors, it is clear that the path to increas-
ing application performance involves increasing parallelization. In response, transactional
memory has emerged as a compelling alternative to traditional locks. When transactions
are considered as a general-purpose synchronization and concurrency primitive, however,
transactional memory systems have very different constraints and responsibilities than ap-
ply when transactions are used only in support of a single application class like databases,
or only for operations on some kinds of data structure. In Table 1.1, a number of representa-
tive transactional memory proposals, and their properties, are shown. Section 1.2.1 explains
these properties, and attempts to flesh out the design space of contemporary transactional
memory systems.

1.2.1 Transactional Memory Design Space

The primary division I make among transactional memory systems is between hardware
and software. Transactional memory can be implemented as an architectural feature, as in
hardware transactional memory systems (HTMs), or as a software library or language feature,
as in software transactional memory systems (STMs). HTMs, in which speculative versioning
and conflict detection are performed in hardware, can offer excellent performance – with
low contention, hardware transactions incur almost no overhead over equivalent sequential
code. However, bounded HTMs, which typically augment the cache to store transactional
metadata, can only support transactions of limited size before their transactional storage
overflows, or of limited duration before a context switch aborts them. Meanwhile, unbounded
HTMs generally demand significant hardware complexity – on the order of, or in excess of,
that required for virtual memory.
STMs are naturally unbounded\textsuperscript{1}, as they employ dynamic memory-resident data structures to store transactional state and ownership information. Software transactional memory systems are also more easily extended than HTMs to provide advanced transactional semantics like nesting, open nesting [54] or paused regions [77], retry and orElse [31], compensation code, or other semantics under active research. However, the lack of hardware support in STMs means that they incur significant overhead over HTMs on startup, transactional accesses, or commit. All STM proposals have one crucial advantage over most HTMs, though: they are available for use, and testing, immediately.

A third class of transactional memory system, hybrid transactional memory systems, typically combine a bounded HTM with an unbounded STM. This are thought to permit near-hardware performance for smaller transactions while still supporting arbitrarily-sized transactions and demanding less hardware complexity than unbounded HTMs.

Transactional memory systems are also categorized by granularity, versioning, conflict detection, conflict resolution policy, nesting, blocking, and atomicity.

Granularity: Every transactional memory system detects conflicts at some minimum granularity. As with cache coherence, finer granularity reduces false conflicts while coarser granularity can reduce overhead. HTMs, which typically piggyback off of the cache coherence system, often use cache-line granularity; STMs often operate on a per-word granularity. Some transactional memory systems operate at object granularity, extending transactional protection to an entire object at once.

Versioning: Transactional writes require two copies of the written location to be stored: the original value (to be used by other code while the transaction is executed, and if the transaction aborts) and the speculative value (to be used when the transaction commits). Where the speculative data is stored is the transactional memory system’s versioning policy. A lazy versioning transactional memory system stores its speculative state privately, and

\textsuperscript{1}Software transactional memory systems are, like all other software, of course bounded by available virtual memory.
sometimes requires a serialized commit phase to update architected state – a fact that gives rise to several performance pathologies. By contrast, *eager versioning* transactional memory systems eagerly store speculative data to memory, eliminating the need for a complex commit phase and optimizing the (hopefully common) commit case. Either technique can be affected by pathological cases, as shown in [12].

*Conflict detection:* Eagerly conflict detecting transactional memory systems detect and act on conflicts as soon as they occur, while lazy conflict detecting transactional memory systems defer either the detection of or the response to conflicts to some later time, often just before commit.

*Conflict resolution policy:* When transactions conflict, a number of behaviors are available. A fixed policy, such as *conflictor aborts* or *younger aborts*, is often employed by HTMs due to its simplicity, but may be prone to pathologies. [12] Unbounded HTMs and STMs, however, often employ an adaptive *software policy*. Such a policy may, for instance, choose to attempt to stall the conflicting transaction in the hopes that the conflictee will retire, and then after some timeout period abort the transaction deemed to have performed the least work.

*Nesting:* Most transactional memory systems support transaction nesting, if only by flattening – subsuming all child transactions into their top-level ancestor. This is trivial to support, requiring only a ’nesting depth’ counter to be maintained so that the system knows when the top-level transaction is closing. Some transactional memory systems support full nesting, which maintains child transactions separately from their ancestors, merging the two only when the child commits. Transactions which support full nesting may be able to support open nesting, a proposed technique for permitting transactional side-effects. Another flavor of nesting, open nesting, permits the state written by open-nested transactions to become architecturally visible upon their commit, even if their ancestor transactions have not yet committed.[53, 54] This approach permits programmers to avoid physical conflicts on shared state while still providing logical conflict detection.
Blocking: As mentioned earlier, transactional memory was originally studied as a tool supporting nonblocking data structures. Not all recent transactional memory system proposals, however, are nonblocking. Typically HTMs are nonblocking: since hardware manages the state of all transactions, one transaction can cause another to be aborted and rolled back at-demand.\footnote{Moreover, most bounded HTMs are not even capable of being stalled, as they abort on a context switch.} However, some STMs may block: if an eager-versioning transaction has write-acquired an address and then stalled, or a lazy-versioning transaction stalls during its commit phase, a later transaction desiring access to that address may have to wait for it to return. Ennals has recently argued that STMs should be blocking to preserve performance\cite{21}, and current STM research appears to concur.

Atomicity: All transactional memory systems guarantee conflict detection between concurrent transactions. However, not all transactional memory systems have strong atomicity\footnote{Strong atomicity is also called strong isolation in the literature; the property itself is perhaps best described as both strong atomicity and strong isolation}, meaning that they can detect conflicts between a transaction and nontransactional code (Section 2.2.2). HTMs, which listen to coherence traffic to detect conflicts, are generally strongly isolating. By contrast, most STMs, whose accesses to transactional data must be guarded by ownership lookups, are weakly isolating. The designations strong and weak atomicity were introduced by Blundell et al.\cite{9} and further examined in the context of a high-level language’s memory model in Grossman et al.\cite{26}, and the semantics of strong and weak atomicity were formally provided by Moore et al.\cite{51}. Generally, hardware transactional memory systems are strongly isolating, since they are not aware of whether conflicting accesses originate from within a transaction or outside of one. Software transactional memory systems are on the other hand typically not strongly isolating, though a recent proposal provides strong atomicity by instrumenting nontransactional code.\cite{68}

I find that while many of the proposals in Table 1.1 offer interesting insights and techniques, none of the proposals presents a system which hardware designers will want to support and concurrent programmers will want to use. Bounded HTMs like TM, LTM,
and Bulk place size or duration restrictions on transactions which would seem arbitrary in higher-level language code. STMs like STM, DSTM, HSTM, and TL2 incur considerable overhead for maintaining transactional state. Unbounded HTMs like UTM, TCC, VTM, LogTM, and PTM balance unbounded transaction size and duration with low performance overhead, but require unrealistic hardware complexity. While very few transactional memory proposals explicitly address supporting I/O, outlawing I/O and going nonspeculative upon I/O (described in Section 3.3) can apply to most transactional memory proposals. However, compensating for speculative I/O is likely to benefit from support for true (not flattening) nesting. Bounded and unbounded HTMs are generally strongly isolating (the benefits of which are described in Section 2.2.2), but most STMs and Hybrids (STM, DSTM, RSTM, McRT-STM, TL2, HyTM, and Intel Hybrid) are not strongly isolating; those which are (HSTM, SA-STM) depend on instrumenting all nontransactional code in which accesses to transactional data cannot be disproven, a technique which can introduce performance overhead in nontransactional code. Similarly, hybrid transactional memory systems like HyTM and Intel’s Hybrid support conflict detection between hardware and software transactions by instrumenting hardware transactions, increasing their performance overhead.

1.3 Summary of the Dissertation

This dissertation makes contributions in two broad categories, both pertaining to the theme of expanding transactional performance and programmability. In Chapter 2, I motivate, describe, and analyze a new hybrid TM system, the UFO Hybrid, which permits good performance, offers reasonable semantics, and requires only moderate hardware support (and that useful beyond this one application). In Section 2.2, I motivate the need for such a hybrid system in three ways: I claim that an effective transactional memory system will have to offer minimal performance overhead on uncontested transactions; that a usable transactional memory system will need to present unsurprising semantics to the programmer; and
that a practical transactional memory system cannot demand considerable special-purpose hardware. On the second point, I elaborate at length, giving examples of surprising semantics and showing how most previous software and hybrid transactional memory systems can behave in unexpected – and undesirable – ways.

In Section 2.3, I describe the hardware primitives from which the UFO Hybrid is composed. I rely on two primitives: BTM, a best-effort bounded hardware transactional memory system which relies on transactional caches for versioning and cache coherence for conflict detection; and UFO, a fine-grained, user-mode hardware memory protection system. In Section 2.4 I describe how these primitives are assembled into the UFO Hybrid. I first show how memory protection like UFO can be used to make a strongly atomic software transactional memory system, then show how I compose such an STM with a hardware transactional memory system like BTM to obtain the UFO Hybrid.

The results I obtained from my simulations of the UFO Hybrid are presented in Section 2.5. There, I show the UFO Hybrid’s performance on transactional benchmarks as well as limit studies relative to its constituent STM, two prior hybrid TM proposals, and an unbounded variant of its constituent HTM. I also describe the performance impact of several design choices in the hybrid, including its hardware conflict resolution policy and its software failover policy. I find that the UFO Hybrid tends to offer HTM-like performance at low software failover rate, and tends to resemble STM performance only to the degree to which it uses the STM. I argue that these are ideal performance characteristics for a hybrid transactional memory system, permitting programmers to target transactional code for high performance while still gracefully supporting the occasional large transaction or advanced transactional technique.

Providing a suitable transactional memory system does not, however, guarantee easy development of transactional codes. Due to the speculative nature of transactional execution, many programming idioms in common, unremarkable use in lock-based code are deeply troublesome, or even downright impossible, in transactional code. Among the most significant
of these hazards is the problem of side-effects within transactions. In Chapter 3, I explore this problem by examining lock-based critical sections in large, parallel codes, and extrapolating the kinds and contexts of side-effects in these critical sections to previously proposed techniques to handle transactional side effects. In Sections 3.2 and 3.3, I give background to the problem by introducing terminology for side-effecting operations and surveying prior proposals for handling side-effects in transactions. After describing my experimental method in Section 3.4, I give results in Section 3.5, including data on which side effects were invoked within critical sections, where they were situated within their critical sections, and the kinds of critical sections in which they were found. I find that if transactions resemble the critical sections used in the workloads I examined, they will certainly contain side-effecting operations; but that between a transaction-aware system library, the ability of side-effecting transactions to go nonspeculative, and the ability for programmers to specify compensation code, virtually all of these side-effecting operations can be handled, the vast majority of them without requiring the compliticy of the application developer.
<table>
<thead>
<tr>
<th>Transactional Memory System</th>
<th>Year</th>
<th>Type</th>
<th>Granularity</th>
<th>Vers.</th>
<th>Confl.</th>
<th>On Conflict</th>
<th>Nest</th>
<th>Block</th>
<th>Strong Atomicity</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TM [33]</td>
<td>1993</td>
<td>HTM</td>
<td>cache line</td>
<td>lazy</td>
<td>eager</td>
<td>conflictor aborts</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>LTM [4]</td>
<td>2004</td>
<td>HTM</td>
<td>cache line</td>
<td>lazy</td>
<td>eager</td>
<td>conflictor aborts</td>
<td>flat</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>UTM [4]</td>
<td>2004</td>
<td>UHTM</td>
<td>cache line</td>
<td>lazy</td>
<td>eager</td>
<td>younger aborts</td>
<td>flat</td>
<td>no</td>
<td>yes</td>
<td>1</td>
</tr>
<tr>
<td>TCC [28]</td>
<td>2004</td>
<td>UHTM</td>
<td>cache line</td>
<td>lazy</td>
<td>lazy</td>
<td>committer wins</td>
<td>flat</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>VTM [60]</td>
<td>2005</td>
<td>UHTM</td>
<td>cache line</td>
<td>lazy</td>
<td>eager</td>
<td>software policy</td>
<td>flat</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Bulk [14]</td>
<td>2006</td>
<td>HTM</td>
<td>word</td>
<td>lazy</td>
<td>lazy</td>
<td>decided by BDM</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>LogTM [52]</td>
<td>2006</td>
<td>UHTM</td>
<td>cache line</td>
<td>eager</td>
<td>eager</td>
<td>software policy</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>2</td>
</tr>
<tr>
<td>PTM [16]</td>
<td>2006</td>
<td>UHTM</td>
<td>cache line</td>
<td>eager</td>
<td>eager</td>
<td>younger aborts</td>
<td>flat</td>
<td>no</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>STM [67]</td>
<td>1995</td>
<td>STM</td>
<td>word</td>
<td>eager</td>
<td>eager</td>
<td>'helping'</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>3</td>
</tr>
<tr>
<td>DSTM [32]</td>
<td>2003</td>
<td>STM</td>
<td>object</td>
<td>lazy</td>
<td>eager</td>
<td>software policy</td>
<td>flat</td>
<td>no</td>
<td>no</td>
<td>4</td>
</tr>
<tr>
<td>HSTM [31]</td>
<td>2005</td>
<td>STM</td>
<td>word</td>
<td>lazy</td>
<td>lazy</td>
<td>committer wins</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>5</td>
</tr>
<tr>
<td>ASTM [43]</td>
<td>2005</td>
<td>STM</td>
<td>object</td>
<td>lazy</td>
<td>1/e</td>
<td>software policy</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>10</td>
</tr>
<tr>
<td>TL2 [19]</td>
<td>2006</td>
<td>STM</td>
<td>object</td>
<td>lazy</td>
<td>lazy</td>
<td>committer wins</td>
<td>flat</td>
<td>no</td>
<td>no</td>
<td>7</td>
</tr>
<tr>
<td>McRT-STM [63]</td>
<td>2006</td>
<td>STM</td>
<td>word/object</td>
<td>eager</td>
<td>1/e</td>
<td>conflictor aborts</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>11</td>
</tr>
<tr>
<td>RSTM [44]</td>
<td>2006</td>
<td>STM</td>
<td>object</td>
<td>lazy</td>
<td>1/e</td>
<td>software policy</td>
<td>flat</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>HyTM [18]</td>
<td>2006</td>
<td>Hybrid TM</td>
<td>word</td>
<td>lazy</td>
<td>eager</td>
<td>software policy</td>
<td>flat</td>
<td>yes</td>
<td>no</td>
<td>8</td>
</tr>
<tr>
<td>Intel Hy [36]</td>
<td>2006</td>
<td>Hybrid TM</td>
<td>object</td>
<td>lazy</td>
<td>eager</td>
<td>software policy</td>
<td>flat</td>
<td>no</td>
<td>no</td>
<td>9</td>
</tr>
<tr>
<td>HaSTM [64]</td>
<td>2006</td>
<td>HaSTM</td>
<td>word/object</td>
<td>eager</td>
<td>lazy</td>
<td>committer wins</td>
<td>flat</td>
<td>yes</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>SA-STM [68]</td>
<td>2007</td>
<td>STM</td>
<td>word/object</td>
<td>eager</td>
<td>1/e</td>
<td>conflictor aborts</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>12</td>
</tr>
<tr>
<td>SigTM [49]</td>
<td>2007</td>
<td>Hybrid TM</td>
<td>cache line</td>
<td>lazy</td>
<td>eager</td>
<td>conflictee aborts</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>6</td>
</tr>
</tbody>
</table>

1. UTM has lazy versioning for data in cache, and eager versioning for data in memory.
2. LogTM receives nesting support in [53].
3. Shavit & Touitou’s STM executes transactions nonspeculatively after acquiring ownership on the entire dataset. When a thread attempts to execute a transaction but is stymied by an executing, possibly swapped-out, transaction holding data it wishes to acquire, it can help the executing transaction by attempting to run its transaction, guaranteeing that no transaction is held up indefinitely by another, stalled transaction.
4. DSTM exhibited obstruction freedom, a weak nonblocking policy.
5. Haskell STM featured strong atomicity by forbidding nontransactional code from accessing transactional data. It also introduced retry, a method for programmers to explicitly avoid some blocking situations.
6. SigTM, while named as a hybrid transactional memory system, is perhaps more accurately a software-assisted HTM. It does not block on stalled transactions, but can only support a limited number of concurrent transactions. An object-granularity variant is described.
7. TL2 forbids sharing data between transactional and non-transactional code.
8. HyTM’s best-effort HTM is a limited LogTM, and inherits LogTM’s properties. HyTM can either forbid concurrent execution of HW and SW transactions, or execute HW transactions instrumented to examine the STM’s OREC_TABLE.
9. Intel’s hybrid transactional extends the DSTM STM.
10. ASTM improves on DSTM by removing a level of indirection when looking up a transactionally-held object.
11. McRT-STM has a blocking implementation, but appears nonblocking via thread scheduler modifications and deadlock detection.
12. SA-STM extends McRT-STM to provide strong atomicity.

Table 1.1: Some Transactional Memory Proposals and Their Salient Properties
Chapter 2

The UFO Hybrid

A transactional memory system can be characterized by three orthogonal attributes: the programming interface that it provides, its performance, and its hardware requirements (discussed in more detail in Section 2.2). The ideal transactional memory system would provide clean semantics and a complete programming model, introduce no overhead to transactions, and require no special-purpose hardware support. While such an ideal transactional memory system is likely unobtainable because of the tension between these attributes in the implementation, I contend that existing proposals fall well short of the ideal in one or more of the attributes.

I believe that hybrid transactional memory systems represent the most compelling approach to supporting transactional memory, but that existing proposals have significant weaknesses. A hybrid transactional memory system executes most transactions using a simple HTM, but handles large, long-running, or otherwise uncommon transactions using a STM [18]. This approach promises the performance of an HTM in the common case and permits arbitrary transactions without the complex hardware required by unbounded HTMs. It also permits much of the system’s transactional semantics to be defined in software rather than hardware. However, existing hybrid transactional memory proposals suffer from two notable drawbacks: they are subject to the non-intuitive semantics resulting from not detecting conflicts between transactional and non-transactional code, and the performance of their common-case hardware transactions is sacrificed to ensure that hardware transactions do not violate the atomicity of software transactions. In this chapter, I propose a new hybrid transactional memory design that addresses these shortcomings of the previous work.
2.1 Introduction

In this proposal, the STM transactions protect the memory locations that they are reading and writing using a hardware fine-grained memory protection mechanism (e.g., Blizzard [65], Mondriaan Memory Protection [72], iWatcher [76]), as shown in Figure 2.1. This prevents accesses from non-transactional code from violating the atomicity of software transactions, because a hardware fault will be raised before a conflicting non-transactional read or write completes.\footnote{Software transactions disable these faults upon beginning and re-enable them upon commit.} In this way, I provide the STM with strong atomicity [9], which, as I discuss in Section 2.2.2, is a property that provides clean transactional semantics in a straightforward way.

This technique, however, also permits hardware transactions to detect conflicts with concurrently-executing software transactions without slowing hardware transactions that do not conflict. When a hardware transaction attempts to perform an access that

Figure 2.1: Using Fine-grained Memory Protection to Achieve a Strongly-atomic, Efficient Hybrid Transactional Memory System

Software transactions (a) use hardware memory protection to write protect values that they read and read/write protect values that they write, as shown in (b). Accesses to these values by non-transactional execution (c) will raise a protection violation, preventing the transaction’s atomicity from being violated. This same protection mechanism also prevents conflicting hardware transactions (d) from violating the atomicity of software transactions, without the hardware transactions having to check STM metadata; they can execute without software overhead.
conflicts with an in-flight software transaction, it receives a protection fault, permitting the hardware transaction to avoid the conflict by backing off or aborting. Since software transactions are required to protect their transactional data with hardware memory protection, hardware transactions can run at full speed because no software checks are required to detect conflicts with STM transactions. In this way, this proposal adopts a pay-per-use philosophy, where the costs of uncommon cases (such as overflowed transactions or I/O) only affect overall performance in proportion to their frequency.

My approach also follows the precept that hardware should provide primitives and not solutions [73]. This hybrid transactional memory system uses a minimum of special-purpose hardware, instead consisting of two hardware primitives — a best-effort hardware transactional memory (Section 2.3.1) and fine-grained memory protection (Section 2.3.2) — that have many compelling applications unrelated to transactional memory. This approach also avoids architecting in hardware the hybrid transactional memory’s semantics; instead, the STM is free to define (and evolve) its full feature set, of which the hardware transactional memory accelerates a (likely common) subset. This approach is particularly compelling as it is in the dark corners of transactional memory (e.g., system calls, I/O, waiting, open nesting) where the desired semantics have yet to be determined, but where programs are likely to spend only a fraction of their time.

This chapter makes the following contributions:

• I show how hardware memory protection enables low-overhead strong atomicity for STMs (Section 2.4.1).

• I demonstrate that this strong atomicity enables a hybrid transactional memory system with zero-overhead hardware transactions that can execute concurrently with potentially-conflicting software transactions (Section 2.4.3) and characterize the performance of this hybrid transactional memory (Section 2.5).

• I identify policies for managing contention between STM and HTM threads and switch-
ing transactions to software that achieve robust performance (Section 2.4.4).

2.2 Motivation

In this section, I discuss the three attributes of an ideal transactional memory system mentioned above — high performance, a clean and complete programming model, and little dependence upon special-purpose hardware — and pitfalls encountered by prior transactional memory proposals.

2.2.1 High Performance

As the goal of multi-core programming is to increase program performance, many programmers will ignore transactions if they significantly underperform locks, even if they offer a cleaner programming model. For example, there has been little adoption of software transactional memory because its significant single-thread overhead causes them to underperform lock-based code in any circumstance where lock contention is not the only bottleneck.

I strongly believe that, to achieve widespread use, uncontended transactions will need to incur little – if any – performance overhead relative to non-transactional execution. If transactions incur negligible single-thread overhead, programmers will be free to use them for concurrency control in almost any piece of code, without having to consider whether that code represents a scalability bottleneck.

2.2.2 A Clean and Complete Programming Model

To be accepted by programmers, transactional memory should expose a programming model that is no more complex than locks. This poses a problem for bounded HTM systems, for programmers should not have to reason about how their transactions fit into the cache geometry and the time quantum of all machines upon which they might run. Furthermore, the transactional memory system’s semantics should be intuitive — however, some proposed
transitional memory systems have exposed non-intuitive semantics, particularly through the lack of strong atomicity.

Under strong atomicity (also called strong isolation) transactions are atomic and isolating even with respect to nontransactional code. Following Blundell et al., strong isolation features two components: non-interference and containment. [9] Both of these components involve how transactional work and nontransactional work interleave.

Transactions operating under a policy of containment do not allow any other code to see their internal state until they commit, at which point all of it is visible. In Figure 2.2, I show an example of a containment violation in an eager-versioning transactional memory system lacking strong atomicity. A transaction in thread 1 speculatively updates the shared value \( \text{val} \). The new value is read by thread 2 – as the access is nontransactional, no conflict is detected – but a subsequent abort of the transaction in thread 1 reverts \( \text{val} \) to its original state. At the end of this sequence, thread 2 has read a value of \( \text{val} \) which should never have existed. In a transactional memory system which offers containment, transactions’ writes
appear to all other threads to happen at one point.

Non-interference is the complementary case to containment: where containment places bounds on when other threads may see a transaction’s writes, non-interference places bounds on when a transactional thread may see other thread’s writes. In Figure 2.3, I show an example of a non-interference violation. In this example, a non-transactional write to val from thread 2 interposes itself between two transactional reads to val in thread 1; this causes an abort which should never be reached to execute.

A transactional memory system which offers non-interference does not cause all of a transaction’s reads to appear to other threads to happen at one point. Instead, all of the transaction’s reads to a given location appear to all other threads to happen at one point; but reads to different locations may appear to happen at different points to other threads. This means that consistency may still be violated under strong atomicity; if the programmer expects all of a transaction’s reads to appear to happen at one point, all writes to those read locations must be guarded by transactions.
In theory, for a well-formed\textsuperscript{2} transactional program, a transactional memory system lacking containment and non-interference should still provide correct execution. However, practical considerations of transactional memory system implementation can lead to consistency violations even in well-formed code, if containment and non-interference are not provided.

For example, a common practice with locks is to privatize a shared object to one thread, typically by making all pointers to it inaccessible from other threads. This allows the newly-private object to be accessed outside of a critical section. Intuitively, privatization should work as well with transactions, but in transactional memory systems which are not strongly atomic, it can lead to race conditions which yield inconsistent behavior. The STAMP benchmarks whose data I report do not use privatization to remove data from shared state, but recent work has shown that privatization can be a very effective performance optimization for transactional code running on software TMs, finding that aggressively privatizing STAMP –

\textsuperscript{2}Here, I use \textit{well-formed} to mean that all references to shared objects in the program are either inside of transactions, or are demonstrably not shared at the point of the reference.
Figure 2.5: A Lost Non-Transactional Write In a Word-Granularity, Eager-Versioning Transactional Memory System Lacking Strong Atomicity

if the underlying STM were to support it – would give significant performance improvements for several of these benchmarks.[74]

Figure 2.4 gives an example of such inconsistent behavior in an eager-versioning transactional memory system which lacks strong atomicity. In this example, threads 1 and 2 have concurrent transactions; thread 2’s writes a value to the shared object head, but a subsequent privatization by thread 1 causes thread 2 to abort. After thread 1’s transaction completes, it operates on the previously shared value once held in head. However, with an eager-versioning transactional memory system, the undo phase is not instantaneous; the original values of speculatively written transactional state need to be replaced. In Figure 2.4, thread 1 believes that it has successfully privatized the element which, before its execution, had been the head of the linked list, but after it has acted on that supposedly-private data, the rollback of the transaction in thread 2 loses thread 1’s privatized write. While this example only holds for eager-versioning transactional memory systems, lazy-versioning systems are susceptible to a complementary hazard on commit.

Another way in which even well-formed transactional programs can become inconsistent
without strong atomicity involves the granularity at which transactional data is logged. When the granularity for logging writes is larger than the minimum-sized write (a common state of affairs for transactional memory systems) and conflicts between transactional and nontransactional code are not detected, there is the potential for lost nontransactional writes. Figure 2.5 demonstrates this problem in the context of neighboring accesses to a byte array. The transaction in thread 1 speculatively updates $S.a$, then thread 2 speculatively updates $S.b$. Then the transaction in thread 1 aborts (for unrelated reasons, perhaps a conflict with another transaction not shown) and begins to roll back. When it restores the original value of $S.a$, it also reverts the original value of $S.b$, leaving the program in an inconsistent state.

As before, a complementary hazard exists for lazy-versioning transactional memory systems. While these unintuitive behaviors can be addressed in a piecemeal fashion – for example, Privatization can be supported by stalling commit until all conflicting transactions complete the abort process [3], and false conflicts can be avoided by preventing data within the same transactional word from being accessed both inside and outside of transactions (easier promised than guaranteed) – they can also be addressed by making the transactional memory system strongly atomic. Strong atomicity — requiring transactions to serialize with conflicting non-transactional (nonT) accesses — has been identified as a sufficient condition to avoid these and other problems [9, 29, 49, 68]. While most HTM systems, which detect conflicts via coherence, are strongly atomic, most STM system proposals are not, because doing so has required instrumenting non-transactional code, the software overheads of which can be significant, even with aggressive whole program analysis [68].

In addition, the boundary of the transactional memory programming model is still a very active area of research, with many compelling opportunities to extend the transactional memory paradigm beyond the basic multi-word compare-and-swap. Two recent examples are transactional waiting and side-effecting transactions. Transactional waiting (introduced as the retry primitive by Harris et al. [31]) can eliminate lost wakeup bugs, but poses serious challenges for HTM systems. Side-effecting operations such as I/O are generally unsupported.
within hardware or software transactions, requiring critical sections that perform such operations to use locks. Transactions which support side-effecting operations [2, 7, 53] could permit transactions to be the single synchronization mechanism to be used for all isolated data accesses. As rich transactional programming models are still an active area of research, it is desirable that they are not precluded by a transactional memory system – as would be the case if a transactional memory system’s semantics were architected into hardware.

2.2.3 The Role of Hardware

Given the limitations of pure software transactional memory, most of the transactional memory community have accepted that some sort of hardware support is necessary to make transactional memory viable. To this end, many unbounded HTM architectures, which handle transactions completely in hardware, have been proposed [4, 16, 52, 60].

There are three problems with these approaches. First, much of the hardware complexity (relative to a bounded transactional memory proposal) is dedicated to handling cases that are expected to be relatively infrequent or not performance-critical. Second, these systems violate the precept that hardware should provide primitives and not solutions [73]; it seems particularly reckless to architect the transactional semantics into hardware when it is not yet clear what the desired semantics are and while potentially different language environments may require slightly different semantics. Third, these approaches introduce significant hardware additions that are specific to transactional programming, whose importance has yet to be quantified.

I believe that (at least for the foreseeable future) hardware support for transactional memory systems should avoid fixing transactional policy in hardware and minimize the amount of special-purpose transactional memory hardware. For this reason, this dissertation explores a primitives-based approach to transactional memory hardware, in which the bulk of the hardware required by the transactional memory system is also useful in contexts beyond transactional memory.
2.3 Hardware Primitives

In this section, I briefly introduce the two hardware components used in my proposed system, both of which are derived from previous work: a “best effort” hardware TM and fine-grained memory protection hardware.

2.3.1 BTM: a “Best-effort” Hardware TM

BTM is a hardware-based best-effort TM system that provides functionality similar to the original TM proposal by Herlihy and Moss [33]. It supports transactions that fit entirely in the transactional cache (L1 in this case), do not raise exceptions or receive interrupts (including timer interrupts), require only flattened nesting, and perform no I/O. These limitations notwithstanding, a significant majority of the dynamic transactions seen in the benchmarks I used are able to execute completely in BTM.3

BTM extends a write-back L1 cache to support speculatively committed loads and stores in much the same way that has been proposed for speculative multithreading [23], speculative lock elision [58], and many other hardware TM proposals (e.g., [4, 28]). BTM operates at cache-block granularity, extending every L1 cache block to include speculatively-read (SR) and speculatively-written (SW) bits. As usual, appropriate coherence permission must be acquired before completing transactional memory operations. When a transactional load commits, it sets its block’s SR bit. Before a transactional store commits, the cache makes sure that the to-be-written cache block is clean (writing a dirty block to the next lower level of the cache hierarchy) before completing the store and setting the block’s SW bit. If a block with an SR or SW bit set is evicted from the cache, the transaction is aborted; all transactionally-written lines are invalidated, and all SR/SW bits are flash-cleared. As discussed in Section 2.4.4, BTM uses an age-based contention management policy.

3This work was done in the context of the x86 ISA, which includes a hardware page-table walker to handle non-page-faulting TLB misses without exceptions. In other architectures, where TLB miss exceptions are relatively frequent, such faults may cause problems such as those described in [50].
This speculative execution hardware is exposed to software through a simple interface (Table 2.1) which permits high-performance implementations. Software specifies the beginning of a transaction with a `btm_begin` instruction, which specifies an abort PC. When a (non-nested) `btm_begin` is executed, a register checkpoint is taken; if the transaction is aborted, this checkpoint is restored and control is vectored to the abort PC. Software can specify that a transaction can be committed or aborted with the `btm_end` and `btm_abort` instructions. A (non-nested) transaction is committed by flash clearing all SR and SW bits and discarding the register checkpoint.

In addition, BTM provides access to status registers that record whether a transaction is executing, its current nesting depth and the reason for the last transaction abort. Abort reasons include cache set overflow, explicit abort, interrupt, illegal operation, conflict, exception, system call, uncacheable access, page fault, and hardware nesting depth overflow. When an address is associated with the event (such as the PC of an explicit abort or the page fault address) it is also recorded so it can be made available to software.

The hardware atomicity provided by BTM is useful not only for implementing TM systems. The same hardware can be used for implementing speculative lock elision (SLE) [58, 62], where lock-based critical sections are speculatively executed as hardware transactions. More recently, hardware atomicity has been proposed as a means to facilitate speculative software optimizations [56].

<table>
<thead>
<tr>
<th><code>btm_begin imm32</code></th>
<th>Begin a BTM transaction with abort address given in immediate &lt;imm32&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>btm_end</code></td>
<td>End a BTM transaction</td>
</tr>
<tr>
<td><code>btm_abort</code></td>
<td>Abort a BTM transaction</td>
</tr>
<tr>
<td><code>btm_mov reg, txr</code></td>
<td>Copy transactional status register &lt;txr&gt; to register &lt;reg&gt;</td>
</tr>
</tbody>
</table>

Table 2.1: The BTM ISA Extension
Fine-grained memory protection mechanisms refine the page-granularity protection supported by virtual memory to granularities smaller than a page [65, 72, 76]. For this work, I consider an iWatcher-style [76] mechanism that permits applications to install access permissions at a cache block granularity and achieves zero execution overhead checking of these access permissions in the common case when no faults occur. In this section, I describe an enhanced implementation of iWatcher, called UFO, which retains iWatcher’s modest hardware complexity, but also provides multiprocessor-safety and supports arbitrarily large regions, context switching, and swapping.

The implementation provides two User Fault-On (UFO) bits per cache line: a fault-on-read bit and a fault-on-write bit. The protection bits and the data with which they are associated move together throughout the whole memory hierarchy: caches, main memory, and the swap file. User-mode instructions are provided for setting and reading the UFO bits (Table 2.2), meaning that protection can be added/removed with low overhead but that this protection is not suitable for security applications. Setting UFO bits requires exclusive coherence permission to ensure that all copies of a block have consistent permissions.

When a processor performs an access not permitted by the block’s current UFO bit settings, a UFO fault is raised. The processor’s exception handling mechanism invokes a software handler registered by the application. As with page faults, the faulting address can be read from a special-purpose register, so that the UFO fault handler can take the
appropriate action. UFO provides the ability to disable UFO faults on a per-thread basis, in the same way that interrupts can be disabled; `enable_ufo`, `disable_ufo` respectively set and clear a *UFO enable* bit, which is part of the thread’s context.

While the implementation of UFO is not the focus of this chapter, understanding its implementation is potentially important to interpreting this chapter’s results. To this end, I provide a detailed description of the UFO implementation assumed in this work.

Fundamentally, the UFO implementation maintains the two bits of protection information described earlier for every cache block of data, extending all levels of the virtual memory hierarchy, as shown in Figure 2.6. My implementation of UFO augments the hardware in three places: the cache hierarchy, the memory controller, and the execution core.

The UFO bits travel with the data throughout the cache hierarchy. In the caches, each line is extended with one UFO fault-on-read and one UFO fault-on-write bit; even with SMT and shared caches, only a single copy of these bits is required per cache line. The existing cache coherence protocol is used to ensure that all threads or processors observe a consistent state of the UFO bits.

UFO bits are also present at a cache-line granularity in physical memory. These bits may be explicitly architected in DRAM chips or stored in a separate memory module, but this design re-purposes some ECC bits by encoding ECC at a larger granularity, as was done to provide storage for the Alpha 21364’s directory [27]. Given the increasing susceptibility to single-event upsets with decreasing feature size, it is likely that ECC will be pervasive throughout future systems. In this case, the only hardware change required to provide storage for UFO bits throughout physical memory is an amendment to the memory controller, permitting it to re-encode ECC at a coarser granularity and to store and retrieve UFO bits in the reclaimed bits.

As physical pages are swapped to and from disk, the operating system is responsible for saving and restoring the UFO bits. The Linux 2.6.23.9 kernel used was modified to allocate an array with one 16-byte element per swap-file location (much like the `swap_map`),
Shaded regions are added or altered to support UFO. In the cache, each line is augmented with Read and Write Protection bits. In the memory controller, ECC is encoded at a coarser granularity, and UFO bits are stored in ECC. In the virtual memory system, swapped-out pages are stored with (possibly compressed) UFO data.

to save the UFO bits when a page is swapped to disk, to restore the UFO bits when a page is swapped from disk, and to clear the bits when a physical page is freed. Using real machine experiments, the overhead of these changes was found to be negligible in workloads where swapping normally occurs (e.g., a parallel kernel build with 512MB memory). Minor overhead was observed with intensive page swapping (e.g., 8% additional overhead when the same kernel build is thrashing from only having 64MB memory); the source of this overhead is additional swapping induced by accesses to the UFO-bit storage arrays. Much of this overhead is eliminated by optimizing the case in which no UFO bits have been set (and thus do not need to be saved or restored) by maintaining an additional array with a single bit per page indicating that all of the UFO bits in a page are clear.

When an instruction accesses the cache, the UFO bit for that type of the access is consulted (as part of the tag check) and recorded in the instruction’s ROB entry, resulting in no additional overhead. Immediately prior to the instruction’s retirement, this bit is
checked; if it is set, then a UFO fault is raised. To support weak consistency models, stores can be speculatively retired into a store buffer before the cache block is available, using one of the many previously proposed techniques [13, 61, 71] to recover precise state if a UFO fault is required.

In addition to the debugging applications for which iWatcher was proposed [76], low-overhead fine-grained memory protection enables a broad array of applications including speculative value specialization optimizations [66], concurrent garbage collection [5], and efficiently supporting self-modifying code in binary translators/optimizers. Like BTM, UFO is a multi-purpose mechanism.

2.4 System Organization

In this section, I describe a novel hybrid transactional memory implementation that addresses the concerns raised in Section 2.2. First introduced in Baugh et al. [6], this hybrid transactional memory — comprised of the two general-purpose hardware mechanisms described in the previous section — executes most transactions directly in BTM with no instrumentation overhead, provides strong atomicity to yield clean transactional semantics, which can be extended (in software) to support a rich transactional programming model. The section is organized around my three key contributions: I show how fine-grained memory protection can be used to build a strongly-atomic STM (Section 2.4.1), I demonstrate how using such an STM enables a hybrid with zero-overhead HTM transactions (Section 2.4.3), and I highlight the policies necessary to achieve performance comparable to an unbounded HTM using such a hybrid (Section 2.4.4).

2.4.1 USTM: Building a Strongly-Atomic STM

I show how fine-grained memory protection can be used by an STM to provide strong atomicity at low overhead. As background, I first describe the design of the eager-versioning,
Table 2.3: The USTM API

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ustm_begin()</td>
<td>Begin a USTM transaction (checkpoint regs, clear log, get seq number, set transaction state)</td>
</tr>
<tr>
<td>ustm_end()</td>
<td>End a USTM transaction (release ownership, discard checkpoint)</td>
</tr>
<tr>
<td>ustm_abort()</td>
<td>Abort a USTM transaction (undo writes, release ownership, restore chkpt)</td>
</tr>
<tr>
<td>ustm_read_barrier(void *addr)</td>
<td>Acquire read permission for &lt;addr&gt;</td>
</tr>
<tr>
<td>ustm_write_barrier(void *addr)</td>
<td>Acquire write permission for &lt;addr&gt;</td>
</tr>
</tbody>
</table>

All functions return void.

The use of the term “barrier” derives from the garbage collection literature and not the instructions required by weak memory consistency models.

eager-conflict detection cache-block granularity STM that I use in this work. I then show how to extend it to be strongly atomic.

The UFO STM, or USTM, is an STM library for C/C++. It implements transactions using the API shown in Table 2.3. Of particular note, calls to ustm_read_barrier and ustm_write_barrier should be inserted (by a compiler) before read and write accesses to shared variables, allowing the STM to acquire permission and perform logging as necessary before the operation is performed: like BTM, USTM detects conflicts eagerly.

USTM relies internally on two data structures: an ownership table (otable) shared between all transactional threads and a per-thread transactional status structure (which includes a transaction log). The shared otable, as shown in Figure 2.7, contains a record for each cache line currently read or written by a USTM transaction (making USTM, like BTM, cache-line granularity.) The otable is logically organized as a chained hash table, with each entry containing: 1) a tag to identify which cache line is specified, 2) the permissions held for the line, and 3) the set of transactions that have access to it. When it cannot be updated with a single atomic operation, the head entry of a chain can be put into locked state to provide isolated access to the chain to handle races between threads updating the otable. In Figure 2.7, hash bins 0, 1, and 3 are unused, hash bin 4 is locked, and bins 2 and 5 have chains; realistic implementations generally have at least tens of thousands of entries to minimize aliasing.
Figure 2.7: The USTM Shared Ownership Table (otable)
The ownership table is organized as a chained hash. Entries contain a tag, owner list and a state. Two special entries exist: *null* entries, which are not in use, and *locked* entries, which can only be examined or altered by the thread that holds the lock.

The use of the *otable* is demonstrated by the *ustm_write_barrier* pseudocode given in Algorithm 2.8. This code optimizes for the common cases when chain length is zero or one. If the requester has an existing entry for the block with write permission, no action is necessary. Inserting an entry when none is present or upgrading an existing entry when the requester has sole read access can be done with a *compare&swap*. If one or more other transactions currently own this entry of the *otable*, conflict resolution is invoked as described below. If an entry is present, but its tag doesn’t match, the whole chain is locked while it is searched for a matching entry; if none is found, an entry is inserted at the head of the chain.

Once ownership has been obtained, the cache block’s address and current values are logged (eager versioning) and the write is performed to memory. The code for *ustm_read_barrier* is similar in structure, except multiple readers are permitted and only the block’s address is logged.
1: procedure USTM_WRITE_BARRIER(trans, addr)
2: index ← GET_INDEX(addr)
3: tag ← GET_TAG(addr)
4: entry_state ← MAKE_ENTRY_STATE(trans, addr, WRITE)  \[\triangleright\] We hope to put this in otable
5: loop \[\triangleright\] Loop to try to acquire ownership
6: o ← otable[index] \[\triangleright\] Read head of appropriate otable chain
7: if o = 0 then \[\triangleright\] Ownership table entry is unowned
8: \hspace{1em} C&S(otable[index].data, 0, entry_state) \[\triangleright\] Ownership acquired
9: return
10: else if LOCKED(o) then \[\triangleright\] Is the ownership table entry is locked?
11: \hspace{1em} BACKOFF( ) \[\triangleright\] Backoff and try again
12: else if TAG(o) \neq tag then \[\triangleright\] Is there a tag mismatch?
13: \hspace{1em} LOCK_ROW(index, o) \[\triangleright\]
14: cursor ← o.next \[\triangleright\] Iterate through the chain
15: while cursor \neq NULL do
16: \hspace{2em} if TAG(cursor) \neq tag then cursor ← cursor.next \[\triangleright\]
17: \hspace{2em} else if OWNERS(cursor) = trans then
18: \hspace{3em} if STATE(cursor) = WRITE then
19: \hspace{4em} UNLOCK_ROW(index) \[\triangleright\]
20: \hspace{3em} return \[\triangleright\] c&s is not required
21: \hspace{2em} else
22: \hspace{3em} cursor.data ← entry_state \[\triangleright\]
23: \hspace{3em} UNLOCK_ROW(index) \[\triangleright\]
24: \hspace{3em} return \[\triangleright\]
25: \hspace{2em} else
26: \hspace{3em} UNLOCK_ROW(index) \[\triangleright\]
27: \hspace{3em} RESOLVE_CONFLICT(index, trans) \[\triangleright\] stall, or kill someone
28: \hspace{3em} break
29: \hspace{2em} continue \[\triangleright\]
30: else if OWNERS(o) = trans then \[\triangleright\] trans is the sole owner of this line
31: \hspace{2em} if STATE(o) \neq WRITE then \[\triangleright\] If trans is already the only writer, do nothing
32: \hspace{3em} C&S(otable[index].data, o, entry_state) \[\triangleright\] If trans is sole reader, become sole writer
33: \hspace{2em} return \[\triangleright\]
34: else \[\triangleright\]
35: \hspace{2em} RESOLVE_CONFLICT(index, trans, o) \[\triangleright\] stall, or kill owners and wait until owners = 0
36: \hspace{3em} continue \[\triangleright\] then try again
37: \hspace{2em} continue

Figure 2.8: Code Example of a USTM Write Barrier

USTM_WRITE_BARRIER(trans, addr) checks the otable for conflicts, then, if none is found, acquires write ownership for trans on addr. If any C&S or LOCK_ROW() fails, the algorithm is repeated from line 5.
When a USTM transaction $T$ aborts or commits, it removes its entries from the otable. For each entry in $T$’s log, the corresponding entry in the otable is removed through a process similar to the algorithm in Figure 2.8: entries owned solely by $T$ are removed, and $T$ is removed from the owners sets of any shared read-only entries. If $T$ is aborting, it restores to memory the logged values of cache blocks to which it wrote.

The conflict resolution policy USTM uses is age-based. An STM transaction that conflicts with another STM transaction stalls if it is younger than one of the transactions it conflicts with, otherwise it aborts the conflicting transactions. My current STM implementation is blocking, relying on transactions to unwind themselves on an abort. As a result, after a transaction notifies a confictor that it should abort, it waits, monitoring the transaction status, until the abort process is complete before it continues. In this way, USTM avoids contention on the otable and possible livelock. Likewise, when a transaction is aborted, it waits until the transaction that aborted it has retired before reissuing, also to avoid otable contention and livelock.

2.4.2 Making USTM Strongly Atomic

USTM, as described in Section 2.4.1, is not strongly atomic. The key to making USTM strongly atomic is to install memory protection for transactionally-accessed cache blocks whenever otable entries are created or upgraded. Specifically, fault-on-write protection is installed by ustm_read_barriers, and both fault-on-read and fault-on-write protection is installed by ustm_write_barriers. To prevent USTM transactions from receiving protection faults for their transactional data, threads disable UFO faults at the beginning of USTM transactions and re-enable them at commit.\(^5\)

An example of the changes necessary to extend USTM to use UFO is given in Figure 2.9, which replaces lines 5 and 6 of the algorithm in Figure 2.8. Note that the atomicity of

\(^5\)This does not affect correct operation of the STM, as conflict detection between USTM transactions occurs as described in Section 2.4.1.
7: if \( o = 0 \) then \( \triangleright \) ownership table entry is unowned
8:   LOCK_ROW(index, 0) \( \triangleright \) lock this \texttt{otable} row
9:   SET_UFO_BITS(addr, \text{READ} | \text{WRITE}) \( \triangleright \) other threads should Fault-on Read and Fault-on Write
10:  \text{otable}[index] = entry_state \( \triangleright \) acquire ownership
11:  UNLOCK_ROW(index) \( \triangleright \) release ownership
12:  \text{return} \( \triangleright \) ownership acquired

Figure 2.9: \textbf{A USTM Write Barrier with Strong Isolation via UFO}

Fragment of \texttt{USTM_WRITE_BARRIER(trans, addr)}, corresponding to lines 7-9 in Figure 2.8, showing how strong atomicity is conferred on USTM writes. Appropriate UFO bits are set when \texttt{otable} entries are inserted.

inserting the \texttt{otable} entry and setting the UFO bits is ensured by locking the \texttt{otable} chain during the insertion. Doing so prevents races between threads inserting and removing an \texttt{otable} entry from leaving the UFO bits in a state inconsistent with the \texttt{otable}. Other places in the STM code that insert, upgrade (from read to write permission) or remove \texttt{otable} entries must set, upgrade, and clear UFO bits, respectively, in the same way.

The virtue of this approach to strong atomicity is that there is minimal additional execution overhead for the STM and no overhead for non-transactional code in the common case of no conflicts. When a conflict does occur, the faulting non-transactional thread vectors to a fault handler — registered by the STM before the first transaction executes — which can stall the non-transactional access or abort the conflicting transaction, based on a software-defined contention management policy. An example of an aborting UFO fault handler is provided in Figure 2.10.

2.4.3 Bringing It Together: The UFO Hybrid

The goal of a hybrid TM [18], which composes a bounded HTM with an STM, is to achieve HTM performance for most transactions, but to support large transactions without the hardware complexity of an unbounded HTM. Hybrid TMs can be implemented by organizing the transactional code as shown in Figure 2.11. The new component in this structure is the \textit{abort handler}, which decides after a transaction is tried and fails in the HTM whether to retry the transaction again in hardware or to failover in software. My abort handler is
1: `procedure` SIGUFO_HANDLER(sig_num, siginfo)
2: `fault_addr` ← GET_FAULT_ADDR(siginfo)
3: `fault_size` ← GET_FAULT_SIZE(siginfo)
4: `fault_type` ← GET_FAULT_TYPE(siginfo) "read or write"
5: `xact` ← FIND_CONFLICTING_XACT(fault_addr, fault_size, fault_type)
6: `while` xact \neq NULL `do`
7: ABORT_TRANSACTION(xact)
8: `xact` ← FIND_CONFLICTING_XACT(fault_addr, fault_size, fault_type)

Figure 2.10: A UFO Fault Handler

In this UFO fault handler, all conflicting transactions are aborted. Alternatively, the conflicting nontransactional operation could optionally be deferred until the conflicting transaction has retired. `FIND_CONFLICTING_XACT()` performs a lookup into the `otable` in the same way as `USTM_WRITE_BARRIER()` (shown in Figure 2.8)

described in Section 2.4.3.

The other additional requirement and the traditional challenge of hybrid TMs is to prevent HTM transactions from violating STM atomicity,\(^6\) which previous hybrid designs solve in ways that negatively impact hardware transactions (as discussed and demonstrated in Section 2.5). This challenge, however, is easily surmounted in a hybrid TM built on the strongly-atomic USTM. Because USTM provides strong atomicity via fine-grained memory protection, accesses by hardware transactions are prevented from violating STM atomicity the same way as non-transactional code is.\(^7\) Notably, this approach adds no execution overhead to hardware transactions — the common case — even when concurrently executing with STM transactions.

One undesirable interaction between UFO and BTM, resulting from their mutual reliance on the underlying coherence protocol, bears mentioning. As exclusive coherence permission is required to set UFO bits (in order to keep them coherent), actions by the STM can cause inflight BTM transactions to abort. As discussed in Section 2.4.4, it is not a problem when

\(^6\)Conflicts between HTM threads are detected by the HTM; those between STM threads are detected by the STM. The HTM’s isolation cannot be broken by the STM because the HTM is strongly atomic; if, for example, a STM transaction tries to write a variable read by the HTM, the request will stall (if `nacked`) or the HTM transaction will be aborted. Without a strongly atomic STM, however, nothing prevents an HTM transaction from observing the writes of, or writing to a variable previously read by, an uncommitted STM transaction.

\(^7\)Unlike STM transactions, HTM transactions do not disable UFO faults.
// A hybrid transaction which first attempts to execute in BTM, but can fail over to USTM.

xact:
  BTM_BEGIN( &&fail );
  // transaction body
  BTM_END();
  goto cont;

fail:
  // Resolve faults if possible. If faults seem resolved, goto xact; otherwise flow through
  BTM_ABORT_HANDLER( xact );
  USTM_BEGIN();
  // transaction body with ustm_reads and ustm_writes
  USTM_END();
  cont:;

Figure 2.11: Hybrid TM code.

If the transaction cannot succeed in BTM, even after BTM’s abort handler is invoked, then it fails over to USTM.

such an abort results from a true conflict, because it is reasonable to prioritize the STM transactions over the HTM transactions. Rather, the concern comes from the potential for false conflicts between two transactions reading the same line. USTM read barriers set the fault-on-write bit for the line in question, which will kill BTM transactions that have that block in their read set. The results given in Section 2.5 suggest that this is not a substantial problem. Were this interaction to lead to a significant loss of performance, it could be addressed by changing the coherence protocol to permit setting UFO bits in the owner state (as previously proposed [8, 11]) or by lazily clearing UFO bits for read-mostly data.

The BTM Abort Handler

When a BTM transaction fails, control is transferred to the abort PC provided to the btm_begin instruction. In my hybrid TM implementations, this address vectors to an abort handler that decides whether the transaction should be re-tried in BTM or should fail over to the STM. This abort handler (Figure 2.12) tries to complete as many transactions as possible in the HTM, while quickly failing over to software for transactions that will end up
completing there. It manages these conflicting goals by using the reason that the transaction aborted to categorize it into one of three classes: transactions likely to fail if tried again in hardware, transactions that should be retried in hardware, and transactions that can be retried in hardware after performing a software action.

Four conditions nearly guarantee that a transaction will abort again if it is re-tried in hardware: cache overflow, system call invocation, performing I/O, and incurring non-page fault exceptions. These transactions are immediately failed over to software — that is, re-tried as software transactions — as they represent the uncommon cases for which users rely upon the increased capabilities of the STM.

Some transactions are aborted due to conditions that are unlikely to repeat. For example, most transactions aborted by interrupts will complete when retried in hardware after the thread is re-scheduled. Similarly, when a transaction is aborted due to contention with another transaction, it is generally best to retry the transaction in hardware because the slower execution of software transactions will tend to aggravate the contention. To mitigate contention, I have implemented an exponential back-off scheme in the abort handler. As indicated in Figure 2.12, the BTM implementation that I used keeps track of the number of aborts by interrupts and conflicts (counting up to 7 of each) since the last committed transaction, so that the transaction can fail over to software and invoke backoff appropriately.

Finally, two abort conditions cause repeatable failure of hardware transactions, but can be addressed by the abort handler itself before restarting the transaction: missing register exceptions and page faults. The x86 ISA permits operating systems to lazily swap in the floating point registers by tracking the presence of the FP/SSE registers and raising an exception on their use when not present. If the transaction was aborted due to an SSE device not available exception (as is observed in the benchmark kmeans), the abort handler executes an SSE nop, provoking the fault, before retrying the transaction in hardware. In Figure 2.12, this is performed in fix_exception().

Similarly, when a transaction performs an access that generates a page fault, the trans-
1: procedure BTM_ABORT_HANDLER(xact_label)
2:      abort_reason ← GET_ABORT_REASON()
3:   if abort_reason = HW_CONFLICT then
4:      if MULTIPLE_CONFLICTS() then
5:         EXPONENTIAL_BACKOFF() ▷ optional software backoff from conflicts
6:      goto xact_label
7:   if abort_reason = INTERRUPT then
8:      if TOO_MANY_INTERRUPTS() then
9:         goto line 18 ▷ may be necessary for progress in long transactions
10:    else goto xact_label
11:   if abort_reason = SET_OVERFLOW then
12:      goto line 18 ▷ likely unrecoverable
13:   if abort_reason = PAGE_FAULT then
14:      FIX_PAGE_FAULT() ; goto xact_label ▷ force pagefault
15:   if abort_reason = EXCEPTION then
16:      if FIX_EXCEPTION() = EXCEPTION_FIXED then ▷ try to fix exception
17:         goto xact_label
18:   fallthrough:

Figure 2.12: The BTM Abort Handler

BTM_ABORT_HANDLER(xact_label), inlined at every transaction, attempts to resolve any problems preventing successful transaction completion in HTM. Only if it unlikely that the transaction will succeed will this routine fallthrough to execute the STM version of the transaction.

action is immediately aborted, leaving the fault unhandled. The faulting address and access type, however, are stored in a pair of transactional status registers. The abort handler can perform an appropriate access to the faulting address,\(^8\) forcing the fault to be handled outside of a transaction, before restarting the transaction in hardware.

2.4.4 Contention Management in a Hybrid TM

As demonstrated by previous work [12, 34], how a TM system responds to contention can have a first-order impact on how it performs. While those works studied contention management in pure HTM and pure STM contexts, respectively, my development of the UFO hybrid TM has led me to study contention management in the context of hybrid TMs. Following experimentation with several different policies, I have identified the following principles for

\(^8\)If a write, first the value is read, then the value read is written back using a compare-and-swap operation; this process repeats until the CAS succeeds.
handling conflicts and deciding when to retry a transaction in the HTM or the STM. I provide sensitivity results in Section 2.5.5 to support these assertions.

First, there appears to be no substitute for having a good contention management policy in hardware. As the need to implement contention management as part of an HTM introduces hardware complexity, I explored naïve hardware contention management policies which guaranteed forward progress by eventually failing over to the STM, where implementing contention management is straightforward. I found, however, that in high contention such an approach often performed worse than the STM by itself. In fact, I found that any significant simplification in the HTM contention management policy yielded a first-order drop in performance. The policy that I implemented, like LogTM [52], uses transaction age in contention management. Unlike LogTM, which implements “requester stalls” using transaction age to detect deadlock causing cycles, the UFO Hybrid performs age-ordered conflict resolution for every request in the HTM: if the requester is older than a block’s current owner, the block is taken and the current owner is aborted. If the current owner is older, the requester is nacked and requests again after 20 cycles.

Second, it is important to only execute a transaction in the STM if doing so is required. In particular, contention should not be a reason to fail over to software, because the STM’s overhead will increase the transaction’s duration, thereby holding contended variables longer, increasing contention. Policies that retry conflict-aborted HTM transactions as STM transactions are metastable; the slightest bit of contention can cause a chain reaction that throws all contending transactions into software.

Third, there is little potential benefit to dynamically prioritizing STM transactions with respect to HTM transactions. As the STM primarily runs long-running, large-footprint transactions (that have already failed to execute in the HTM), they are generally older than any hardware transactions they conflict with. In my experiments, the STM transaction is older in more than 99% of such conflicts. As a result, the UFO hybrid statically prioritizes STM transactions over HTM transactions, which is also the simplest policy to implement.
2.5 Performance Analysis

In this section, I characterize the performance of the UFO hybrid TM, demonstrating that it is a compelling alternative to previously proposed hybrid TMs and that it achieves performance comparable to pure (unbounded) HTM systems, which must guarantee the forward progress of all transactions in hardware. Specifically, my experiments compare the UFO hybrid to an unbounded HTM, three STMs, and two previously proposed hybrid TMs which I describe below: HyTM and Phased TM (PhTM). To facilitate comparison between these TM schemes, wherever possible I use the same building blocks: USTM (Section 2.4.1) and BTM (Section 2.3.1).

I give results from three STMs: USTM without UFO-based strong atomicity, USTM with UFO-based strong atomicity to show the overhead of using memory protection for strong atomicity, and TL2, to link my performance with previously published results [19, 49]. For all but the unbounded HTM configurations, hardware transactions are limited to those that can fit in the L1 data cache.

For the unbounded HTM system that I model, I use the BTM model,\(^9\) except the memory footprint of a transaction is not limited. In this way, my unbounded HTM is idealized with respect to actual pure HTM proposals (e.g., it can flash clear on an abort, where LogTM uses a software rollback mechanism), meaning the results given for the unbounded case may be optimistic with respect to what is implementable.

HyTM [18] addresses the challenge of detecting HTM/STM conflicts by burdening hardware transactions with the responsibility of checking the STM metadata to ensure that they are not violating the atomicity of STM transactions. The code of HyTM’s hardware transactions are instrumented with read and write barriers, much like its software transactions, but the code is substantially simpler. The barriers perform the same `etable` lookup, but

\(^9\)While the code for the unbounded HTM experiments does not include STM-targeted versions of the transactions, it does include a simplified abort handler necessary to make forward progress in the presence of page faults and `SSE device not available` exceptions (described in Section 2.4.3).
merely inspect whether a conflicting record is present. If a conflicting record is present, the
transaction explicitly aborts and retries again in hardware. The primary drawback of adding
these checks is the execution overhead they introduce into the hardware transactions. In
addition, my implementation, like the original [18], reads otable entries transactionally, cre-
ating the potential for false conflicts when unrelated STM accesses alias to the same otable
rows previously read by HTM transactions.\footnote{Such false conflicts could be eliminated by extending BTM to support nontransactional loads for use by the barrier code.} Furthermore, these transactional reads of the
otable inflate HyTM’s transactional footprint, sometimes yielding extra cache set overflows.

PhTM [39] avoids instrumenting hardware transactions by precluding HTM and STM
transactions from executing concurrently. The system maintains a counter of the number
of STM transactions currently executing, which is read at the beginning of each HTM
transaction. If the counter is non-zero when read or is updated during the HTM transaction’s
execution, the HTM transaction aborts. The major drawback of this approach is that if one
hardware transaction has to fail over to software, it takes the rest of the concurrent hardware
transactions – even those which could have completed in hardware – with it. To prevent an
STM phase from lasting perpetually, PhTM maintains a second counter which tracks the
number of running transactions that failed over to software due to a condition the HTM does
not support (e.g., cache overflow, exception). As long as this second counter is non-zero, any
new transaction will commence in the STM. When this second counter reaches zero, PhTM
starts the shift back to an HTM phase by stalling transactions rather than starting them in
the STM. When the first counter reaches zero, the last STM transaction has completed, and
the waiting transactions can commence as HTM transactions.

2.5.1 Experimental Method

In these experiments, the hardware is modeled in an x86 full-system, timing-first [47],
evaluation-driven simulator, built using Virtutech Simics [42] and incorporating the x86 in-
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor frequency</td>
<td>4.0 GHz</td>
</tr>
<tr>
<td>Fetch/Decode width</td>
<td>3 x86-instructions</td>
</tr>
<tr>
<td>Rename/Issue/Retire width</td>
<td>4/4/4</td>
</tr>
<tr>
<td>Instruction window size</td>
<td>128</td>
</tr>
<tr>
<td>Branch predictor</td>
<td>4K gshare, 8-bit history</td>
</tr>
<tr>
<td>Indirect target predictor</td>
<td>256-entry BTB, 32-entry RAS</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64 KB, 8-way, 3 cycle hit</td>
</tr>
<tr>
<td>L2 Unified Cache</td>
<td>512 KB, 8-way, 11 cycle hit</td>
</tr>
<tr>
<td>L1 Cache Line size</td>
<td>64-bytes</td>
</tr>
<tr>
<td>L2 Cache Line size</td>
<td>64-bytes</td>
</tr>
<tr>
<td>Physical memory</td>
<td>512 MB, 100 cycle latency</td>
</tr>
<tr>
<td>Coherence protocol</td>
<td>MOESI directory</td>
</tr>
<tr>
<td>Operating system</td>
<td>Red Hat Enterprise Linux 4</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>Modified 2.6.23.9 kernel</td>
</tr>
<tr>
<td>USTM table size</td>
<td>65,536 entries</td>
</tr>
</tbody>
</table>

Table 2.4: Simulation parameters

struction decoder from PTLsim [75] and the Ruby MOESI-directory memory system [45]. This simulator is distributed publicly as FeS2, and is available at http://fes2.cs.uiuc.edu.

The simulated system includes a modified Linux kernel that provides support for saving and restoring UFO bits when physical memory pages are swapped to and from disk. The details of the simulated system are provided in Table 2.4.

I used programs from the STAMP benchmark suite [48, 49], which consists of several transactional programs that exhibit a diversity of transaction construction and interaction. kmeans implements a clustering algorithm and consists largely of small transactions. ssca2 is a set of graph-theory kernels. vacation is a reservation-scheduling system that includes large, long-running transactions that sometimes overflow the cache. genome is a gene-sequencing application whose transactions periodically overflow the cache and exhibit significant contention at a central bottleneck. labyrinth is a maze-routing algorithm. As in previous work [48, 49], I show data for both high- and low-contention configurations of kmeans and vacation.
Figure 2.13: **STAMP Performance of Several TM Schemes**

Performance is normalized to that of sequential execution. USTM (SA) is USTM with strong atomicity via UFO enabled.

Figure 2.14: **Hardware Abort Reasons**

Explicit aborts only occur in PhTM and HyTM, and UFO bit sets only occur in the UFO hybrid, so they are shown with the same bar. Recoverable aborts include interrupts, page faults, and **SSE device not available** exceptions. NonTransaction conflicts result from STM writes to data held by hardware transactions. **kmeans** high contention (not shown) is qualitatively similar to **kmeans** low contention.
2.5.2 Results

The performance results are shown in Figure 2.13, plotted as speedups relative to sequential execution. Even when some transactions overflow to software, the UFO hybrid can achieve performance close to that of an unbounded HTM solution while retaining the ability to fall back to an STM for cases that hardware designers choose not to handle. Additionally, the UFO hybrid consistently outperforms (or performs equally well as) both the HyTM and PhTM hybrids. Figure 2.14 shows the reasons that hardware transactions aborted in the benchmarks.\footnote{STAMP benchmarks feature no explicit aborts, so any aborts seen in Figure 2.14 are due to the operation of PhTM and HyTM.} Finally, it can be seen that making USTM strongly atomic (via adding UFO bit operations) adds little overhead to the baseline USTM, which performs similarly to TL2 [19] in all but \textit{kmeans}.

As noted previously, \textit{kmeans} gives hybrids few reasons for transactions to fail over to software. Almost all of the aborts in kmeans are due to contention or other recoverable reasons. As a result, performance of all of the hybrids closely parallels that of unbounded hardware in both the high- and low-contention runs because almost all transactions commit in hardware. Specifically, there is less than a 1\% difference in performance between unbounded HTM, the UFO hybrid, and PhTM. The barriers in HyTM cause its performance to lag the rest by 10-20\%; in \textit{kmeans} the barrier overhead is small because of its low density of STM barriers.

\textit{ssca2} has similar performance to \textit{kmeans}. Like \textit{kmeans}, its transactions are generally small-footprint, and they also tend to be quite short in execution time, such that virtually all hardware aborts are due to conflicts.

In contrast, \textit{vacation} presents a significant challenge for hybrid TMs, since it consists of long-running, large-memory-footprint transactions. The hybrid TMs actually perform better in the high-contention case because the low-contention version has more transactions that overflow the cache.\footnote{The benchmark configuration dictates what fraction of a fixed-sized database is accessed, with low contention accessing more of the database, leading to larger working sets.} The effects of a non-trivial number of large transactions failing
over to software can be seen in Figure 2.14c: the UFO hybrid incurs more transactions killed by *UFO bit sets* (re-tried in hardware); HyTM receives more *nonT conflicts* on previously-read *otable* entries, as described earlier in this section; and PhTM generates more *explicit aborts* (due to trying to start hardware transactions while software transactions are in-flight) and *nonT conflicts* (on the software-transactions-in-flight counter, due to software transactions starting while hardware transactions are in-flight). This propensity to cache overflow, combined with the long-running nature of *vacation*’s transactions (which run still longer in software) particularly affects PhTM, whose performance actually begins to degrade as the number of threads (and the chance that one of them is running a software transaction) increases, as shown in Figure 2.13.

The performance variation seen among the hybrids in *vacation* low contention is largely due to the set overflows; when the transactional cache is made sufficiently large to hold all *vacation* low contention’s transactions, the hybrids perform almost exactly as they do for *vacation* high contention. However, only the UFO hybrid is capable of simultaneously deploying no-overhead hardware transactions while permitting only those transactions that really need to fail over to software to do so. This ability gives it a marked advantage.

The larger working set of *vacation* also reveals another weakness of HyTM: in Figure 2.14c, it can be seen that HyTM suffers a notably greater number of cache *set overflows* than the other hybrids. This effect is due to accessed *otable* entries competing with the transaction’s data for space in the transactional cache. These *otable* accesses do not only result in additional cache overflows, but also drastically increase the number of *nonT conflicts* that abort hardware transactions, both of which negatively impact performance (as seen in Figure 2.13.) When HyTM’s accesses to the *otable* are performed nontransactionally, the number of set overflows experienced under HyTM in *vacation* drops by over two-thirds. Furthermore, the longer-running HyTM hardware transactions tend to run into timer interrupts more often, yielding a greater number of recoverable aborts.

*genome* exhibits a high-contention initialization phase, in which elements are inserted
into a shared linked list in sorted order — a data structure not well suited for concurrent writes by transactions. While the code could be modified to alleviate this contention, it serves as a challenging test case for TM implementations. When a transaction writes the list, it kills any transactions (which are necessarily younger because the UFO Hybrid resolves conflicts with age ordering) that have read the written part of the list. It is this type of code that requires that a TM has robust contention management so that forward progress and scalability are ensured, as discussed in Section 2.5.5. As relatively few transactions fail over to software, PhTM and the UFO hybrid are able to give performance comparable to the unbounded HTM.

The unbounded hardware TM system which we used as a control provided an upper bound on performance scaling for all the benchmarks but one – labyrinth. Its behavior is worth describing in some detail, because in labyrinth, the ideal, unbounded hardware TM system underperforms every other TM system, with essentially serial performance regardless of the number of processors used. Starting with a work-queue containing pairs of three-dimensional coordinates – each work-queue element contains a source coordinate and a destination coordinate – labyrinth attempts to route as many paths as possible to connect sources with destinations, but without overlapping any paths. There are three static transactions in this program: the first to atomically obtain a worklist element, the second to attempt to create a route between the source and destination points obtained from the worklist, and the third to record this route in a shared list. The first and third static transactions are like those seen in kmeans or ssca2: short and prone to good concurrent execution with hardware aborts largely due to conflicts. The second static transaction, however, is unlike the others. Rather than simply attempting to provide isolation on shared structures, this transaction attempts to perform a complex action – random generation of a non-overlapping path between a source point and a destination point– atomically.

Figure 2.15 shows pseudocode of this transaction. It begins by making a local working copy of the current state of the maze – an action which must be performed transactionally.
Figure 2.15: A Problematic Transaction in labyrinth

The GRID_COPY() operation, without early transactional release, assures that all threads in the hardware-only TM will serialize this work.

Note that for correctness, this working copy need not be the most up-to-date; should anything have changed since its creation, this will be explicitly detected and handled later on. Next, it determines whether any path exists at all between the source and the destination in the local copy of the grid. If such a path does exist, it is inserted, one point at a time, into the global, shared, maze structure. If, while inserting a point, it is found that another route has already passed through that point, the transaction is aborted, discarding all work done on the current coordinate pair; if no points on the path were found already to be in use, the transaction commits, effectively instantaneously adding the entire new path to the global maze structure.

The challenge for a best-effort hardware TM system lies in the GRID_COPY() operation, which brings the entire shared grid structure into transactionally-read state. This has two troublesome effects. First, depending on the size of the grid, it can demand a great deal of space in the transactional cache. Even for a very modest grid size, the unbounded TM system required on the order of sixteen times more hardware transactional storage than was allowed to the hybrids; incidentally, this yielded a guaranteed set-overflow for all the hybrids which always forced this transaction to execute in software. More troubling, however, was the fact that in every dynamic hardware instance of this transaction, the entire grid is read, and a part of it is written, which ensures that every dynamic hardware instance of this
Figure 2.16: Range of Speculative Cache Footprints

Results are shown for the three hybrid TM systems. HyTM tends to exaggerate large transactional footprints, such as those in \textit{genome} and \textit{vacation}.

transaction will conflict with every other dynamic hardware instance of this transaction – effectively serializing this work across all processors.\textsuperscript{13} Should this circumstance prove common, it might motivate the support of some form of early transactional release, such as open nesting \textsuperscript{[54]}, in hardware.

2.5.3 Speculative Footprint

As discussed in Section 2.5.2, a primary cause of software failover is a set overflow in the transactional cache. HyTM, which performs an \texttt{otable} lookup for every transactional read or write, is particularly prone to set overflows. Just before the commit or abort of a hardware transaction, the percentage of the transactional cache currently occupied by speculative data is recorded; the ranges of these results for the three hybrid TMs on several transactional benchmarks are shown in Figure 2.16. This figure makes it clear why, in Figures 2.13 and 2.14, HyTM suffered such poor performance under \textit{genome} and \textit{vacation}: its transactional footprint is simply much larger than those of UFO or PhTM, due to transactionally reading the STM’s \texttt{otable}.

Figure 2.14 shows that HyTM is more prone to set overflows than the other hybrid

\textsuperscript{13}Again, my primary concern here is not whether this code is implemented optimally, but rather I purpose to demonstrate robust performance across a wide variety of code – even inefficently-written code.
approaches, but some benchmarks – notably vacation and labyrinth – produce set overflows in all hybrids.\textsuperscript{14} It is of course essential that the cache replacement policy does not evict transactional data when nontransactional data exists in the cache set, but sometimes even this policy is not enough, and set overflows ensue. A simple technique to avoid this is to incorporate a transactional victim cache [35], a small fully-associative cache to which transactional values evicted from their set may be relocated to avoid leaving the transactional cache entirely. Only when a transactional value is evicted from this victim cache is a transactional set overflow abort generated.

In Figure 2.17, I show the effects of varying-sized victim caches in two frequently set-overflowing benchmarks: vacation, low contention (a); and labyrinth (b). In vacation, increasing the victim cache capacity gradually reduces set overflows. To demonstrate this effect, I have varied the cache geometry from 16 ways down to 2 ways (all with 64 sets), but I especially note that in the 8-way cache configuration used for all results in this chapter, a 9-entry transactional victim cache would avert all set overflows and, correspondingly, significantly improve UFO Hybrid and PhTM Hybrid performance.

labyrinth, on the other hand, does not benefit from any small victim cache. As mentioned in Section 2.5.2, labyrinth features a static transaction which is certain to yield set overflows for the hybrid TMs in every dynamic instance; this is simply a matter of the cache being much too small for the transactional working set. Figure 2.17 (b) shows the effect of this transaction: until the victim cache is large enough to compensate for the too-small cache, none of this labyrinth’s set overflows are averted.

\textbf{2.5.4 Software Failover}

The performance of the hybrid TMs is a complex interaction involving contention and the fraction of transactions that must execute in software. In real programs, these variables

\textsuperscript{14}N.b.: these set overflows occur not because the cache is full, but because one particular set overflowed. This is not an uncommon event, and a similar effect troubles STMs, as shown by Zilles and Rajwar in [79].
Figure 2.17: Victim Caches Can Avert Set Overflows

Vacation, low contention (a) can particularly benefit from such victim caches, but workloads whose transactional working set is simply too large for the cache (such as labyrinth, (b)) will not benefit. All results for 16p.

are related, as having a transaction fail over to software typically increases the amount of contention, as software transactions tend to hold permissions for cache lines longer. In an attempt to isolate the impact of software failover rate, I constructed a microbenchmark where the transactions result in no conflicts, but randomly fail over to software at a prescribed rate. I show the results of this microbenchmark in Figure 2.18 as a function of the failover rate, compared to pure HTM and pure STM approaches.

Clearly, for every hybrid, an increasing software failover rate results in decreasing performance – or, more precisely, as more transactions are forced to software, performance becomes more like a pure STM. However, the rate at which performance decreases differs between the hybrid proposals. UFO Hybrid and HyTM vary almost linearly between pure HTM and pure STM performance, executing only as many transactions in software as were randomly failed over, but PhTM must execute not only those transactions in software, but also any other concurrent transactions in software, even if they could have completed in hardware. This behavior worsens with increased processor count, with increased likelihood that at least one transaction needs to execute in the STM, as was seen in vacation in Figure 2.13.

While all of the hybrids approach pure HTM performance at low failover rates, Fig-
Figure 2.18: **Software Failover Limit**

(a) compares the hybrid proposals listed in Section 2.5. PhTM, whose performance varies with the number of processors, has five curves from 1p to 16p; for HyTM and the UFO hybrid only the 8p runs are shown, as their behavior doesn’t vary with processor count. Parallel runtime (y-axis) is normalized to HTM-only execution as software failover rate (x-axis) is varied from 0-100%. STM data are provided for reference. (b) zooms into (a) at low software failover rates.

Figure 2.18b shows that there are performance differences. At 0% failover, the UFO hybrid performs equivalently with the pure HTM; the observed 6% overhead is due to additional code in all of the hybrid TMs that force the software failovers. PhTM incurs an additional 2% overhead from checking the counter of STM transactions. HyTM’s overhead is even higher, due to the _etable_ lookups it must perform. In this way, the UFO hybrid exhibits the _pay-per-use_ principle, where the overhead of supporting STM transactions is only paid by those transactions that execute in the STM. The UFO hybrid, however, introduces overhead (not present in HyTM and PhTM) into its software transactions to set and clear the UFO bits. As a result the UFO hybrid’s curve has a greater slope than HyTM, making it underperform HyTM for software failover rates exceeding 45%, but it is unclear whether the performance of workloads with such high failover rates is important.

A curious feature of Figure 2.18a is the ‘bump’ in PhTM’s curves, particularly notable in the 8p curve around 25% failover and in the 16p curve around 15% failover. In these curves, PhTM’s performance at relatively low failover rates, in the 15-40% range, is actually worse than its performance at 100% failover, with the problem being exacerbated with
Figure 2.19: **Failover Limit Detail for PhTM as the Amount of Work Done in Each Transaction Is Varied**

As transactions perform more work, the penalty for aborting them partway through increases.

Increasing processor count. At some point along each of these curves, the great majority of PhTM transactions will ultimately commit in software. Clearly, this is the case at 100% failover, but as the number of processors increases, the likelihood at any given time that a software transaction is in flight also increases, simply because more transactions are in flight concurrently, so that in 16p, under 10% of transactions retire in hardware at a forced failover rate of only 20%, and in 8p, under 10% of transactions retire in hardware at a forced failover rate of only 35%. If the forced failover rate is relatively low, but the ultimate software commit rate is relatively high, then many transactions begin in hardware, then do some work as hardware transactions, then fail over to software, thus wasting all the work done in hardware, and producing the ‘bump’ in Figure 2.18a.

Figure 2.19 further explores this phenomenon by varying the work done in PhTM’s transactions. As the amount of work per transaction increases, the opportunity for hardware transactions to waste work increases, and the bump becomes more pronounced. Accordingly, I expect PhTM to exhibit pathological (e.g. trending to worse than pure-STM) performance in multithreaded workloads with larger transactions when the software failover rate is not negligible. This matches the behavior seen in vacation 16p in Figure 2.13.
2.5.5 Sensitivity Analysis

In the development of the policies described in Section 2.4.4, I ran a number of experiments exploring alternative policies, which I attempt to summarize here. My most important result is that performance fell sharply whenever I used a low quality hardware contention management policy. For example, using a simple “requester-wins” policy for hardware contention management results in a nearly-twofold slowdown overall – the result of STM-like performance in high-contention regions. (first bar in Figure 2.20) That these regions exhibited STM-like performance in high-contention regions is not surprising: the “requester-wins” policy was highly sensitive to livelock, and avoided it by failing over to the STM after a number (e.g., 5) of contention-induced aborts.

Second, presuming a strong hardware contention management policy, I found that performance was better if conflicts never caused a fail-over to software, as opposed to failing over on the nth abort (second bar). I found this effect could be partially mitigated by preventing hardware transactions from aborting unless absolutely necessary (third bar) — for example, I tried having BTM transactions stall instead of abort on UFO faults resulting from conflicts with STM transactions — but this is not necessary when contention never causes a fail-over. Finally, I include results for a limit study where UFO bit sets only abort BTM transactions when they represent a true conflict that shows that there is little lost performance due to
false conflicts (fourth bar).

2.5.6 Using HTM to Support an STM System

Like most STM systems, USTM uses conventional atomicity techniques like compare-and-swap to guarantee exclusive access to its ownership table, as described in Section 2.4.1. When a transactional read or write is performed, the relevant ownership table entry must be atomically read, modified, and written back. In the algorithm in Figure 2.8, there are two ways in which this can happen. If the relevant ownership table entry is empty, then a simple compare-and-swap operation can be used to gain ownership of a line atomically – as long as an otable entry fits within a compare-and-swap word. However, if it is not empty, a list-traversal may be required, which requires more atomic action than can be accomplished in a single compare-and-swap; in this case the otable is locked. Since this locked region never accesses more than one otable row, deadlock does not threaten the easy (and effective) performance optimization of permitting otable locking on a finer granularity, such as per-row.

The assumption that an ownership table entry fits within a compare-and-swap word, however, may not be a safe one. In USTM, a tag, a bitmap of owners, and a transactional state must all fit within an ownership table entry. The size of the tag is dictated by the size of the otable, and the number of possible states is fixed, but the size of the bitmap of owners sets a clear constraint on the number of concurrent transactional threads. In my USTM implementation, up to sixteen transactional threads may be uniquely identified by software transactions, but this may be unnecessarily limiting for many applications.

If, however, USTM is being used in the context of a UFO Hybrid, then the means for a multi-word compare-and-swap are already in place – the best-effort hardware TM system. Doing so has two beneficial effects: it can permit a much larger amount of state \(^{15}\) to

\(^{15}\)The amount of state which can be atomically updated depends, of course, on the transactional footprint supported by the HTM. A 64k cache can, however, support a bitmap holding over 250,000 possible transactional owners, so this limit is probably theoretical
be atomically updated in the otable without a significant loss of performance, and it can improve concurrency when separate transactional threads are operating on different transactional data which nonetheless aliases to the same otable row. In this work, I implemented a version of USTM which uses hardware transactions to perform atomic updates to its otable; an example of the resulting code is given in Figure 2.21. Note that this approach approximates speculative lock elision [59] using TM hardware, and is very similar to the Cxspinlock construct introduced to support TxLinux. [62]

In this algorithm, operations on the otable are first attempted in BTM transactions. Should a BTM transaction fail, due to conflicts\textsuperscript{16}, page faults, or other events, an abort handler is invoked, like the one described in Figure 2.4.3. To ensure forward progress, if the fault is deemed unrecoverable, USTM retries the otable operation using conventional locking – a circumstance which never arose in my testing. When the operation is complete – whether through BTM or through conventional locking – the barrier code proceeds in the normal way, handling any conflicts and returning.

Figures 2.22 and 2.23 characterize the BTM transactions executed by this version of USTM (in full STM mode, not hybrid). In Figure 2.22, the abort rates of these BTM transactions is given. The great majority of workloads exhibit BTM abort rates well under 1%; only labyrinth-16p shows significant abort rates. This is because labyrinth executes relatively few transactions, so a few operations on very high-contention structures – a work queue and an output queue – experienced many conflicts. Figure 2.23 shows that most of the BTM transactions which did abort did so due to conflicts; no BTM transactions aborted for an unrecoverable reason.

\textsuperscript{16}It is vital to stress that these are conflicts on the ownership table, not on the transactional data being accessed by the library.
procedure PERFORM_WRITE_BARRIER(trans, addr)

1: tag ← GET_TAG(addr)
2: otatable_row ← GET_OTABLE_ROW(addr)
3: if GET_STATE(otatable_row) == UNOWNED then
4: MAKE_WRITE_ENTRY(otatable_row, trans)
5: else
6: Travers a (possibly singleton) list
7: link ← otatable_row
8: prev ← NULL
9: while link ≠ NULL do
10: if GET_TAG(link) == tag then
11: if GET_OWNERS(link, trans) == trans then
12: if GET_STATE(link) == WRITE_OWNERSHIP then
13: RETURN(NULL) ▷ We own it in write; no action needed
14: else
15: SET_STATE(link, WRITE_OWNERSHIP)
16: RETURN(NULL) ▷ acquire in write
17: else
18: conFLicer ← GET_OWNERS(link, trans)[0] ▷ Return first owner
19: RETURN(conFLicer)
20: else ▷ Look in the next list element...
21: prev ← link
22: link ← link→next
23: MAKE_WRITE_ENTRY(prev, trans) ▷ Add a new chain link

1: procedure WRITE_BARRIER(trans, addr)
2: conFLicer ← NULL ▷ On BTM xact failure, goto line 8
3: BTM_BEGIN(8) ▷ Spin-wait
4: while GET_LOCK(addr) ≠ UNLOCKED do nothing
5: conFLicer ← PERFORM_WRITE_BARRIER(trans, addr)
6: BTM_END( ) ▷ Transaction completed successfully
7: GOTO(11)
8: BTM_ABORT_HANDLER(3) ▷ Goto line 3 if fault was handled
9: while ACQUIRE_LOCK(addr) ≠ UNLOCKED do ▷ Spin-wait
10: conFLicer ← PERFORM_WRITE_BARRIER(trans, addr) RELEASE_LOCK(addr)
11: if conFLicer ≠ NULL then
12: index ← GET_INDEX(addr)
13: tag ← GET_TAG(addr)
14: RESOLVE_CONFLICT(index, tag)

Figure 2.21: Code Example of a USTM Write Barrier using BTM for Atomic otatable Updates
The write barrier work, done in PERFORM_WRITE_BARRIER(), is first attempted in a BTM transaction. If this fails, a conventional lock is acquired. Note that the conventional lock state is read transactionally, and that the structure of the atomic region in WRITE_BARRIER() is similar to the hybrid transaction shown in Figure 2.11.
Considering this modification less quantitatively, there are two compelling reasons borne from my experience to adopt it. First, as mentioned earlier, it efficiently supports many more concurrent transactions by permitting a much larger bitmap of owners to be atomically modified. This is not, however, the greatest advantage which this technique gave me. Correctly implementing an efficient STM was a difficult and exacting task, rife with subtle race conditions. Under the HTM-based STM implementation described in this section, however, operations so rarely failed over to the lock-based implementation, fine-tuning the performance of that implementation was not essential. The result for me, as STM developer, was a much simpler and cleaner STM implementation requiring significantly less debugging.

2.6 Concluding Remarks

I have described an implementation of transactional memory which I believe is compelling. This proposed system, the UFO hybrid TM, is comprised of two hardware primitives of modest complexity which have broad applicability beyond TM, provides a clean TM programming model with strong atomicity, and achieves performance rivaling an unbounded HTM across a broad set of workloads, without the challenges of having to guarantee completion of all transactions in hardware. In addition, since the semantics of the TM are not architected in hardware, this approach is viable even in the presence of the current TM bootstrapping problem: without significant transactional application development, it is difficult to know what features a useful TM programming model requires, but few – if any – programmers will be willing to do significant development until transactional memory offers acceptable performance – that is, better performance than STMs currently give. The hybrid approach permits evolution of the TM semantics via the STM’s extensibility while preserving near-hardware performance.
Figure 2.22: Abort Rates of BTM Transactions in USTM Execution
BTM abort rates are under 1% for most workloads and concurrencies. labyrinth is a notable exception, in which a large proportion of transactional accesses are to a highly shared work-queue and output-queue.

Figure 2.23: Reasons for BTM Aborts in USTM Execution
Most BTM-based USTM operations abort due to conflict, but recoverable exceptions like pagefaults and interrupts are significant in some workloads.
2.6.1 Hardware TM Policies

One of the most attractive qualities of hybrid transactional memory systems is their ability to employ best-effort hardware transactional memory support; the potential simplicity of such HTMs presumably makes them more likely to be implemented. The flexibility of the hybrid TM to use whatever hardware support is made available should not, however, be taken as a blank check to use extremely primitive hardware designs. When considering the performance results given in Section 3.5, I found that good hardware design choices were essential for good performance in several benchmarks. Section 2.5.5 shows, for example, that when the hardware conflict resolution policy is “requester wins” – perhaps the most straightforward policy to implement, as it requires no extension to the existing cache coherence policy – performance suffers significantly. That a hardware conflict resolution policy which is unaware of the context of its dynamic transactions should be detrimental to hybrid TM performance should not be surprising; Bobba et al. came to the same conclusions in their work on HTM pathologies [12], and Rajwar et al. included transactional priorities in their work on transactional lock removal [57]. Providing a more flexible policy, such as age-ordering (“oldest wins”), is likely to increase hardware complexity. However, I do not believe this is insurmountable; to support this belief, I will sketch out how an approximate-age-ordering system might be implemented.

Age-ordering in HTM seems to have two engineering problems: assigning ages fairly to hardware transactions and providing age information when conflicts arise so that the correct resolution can be applied. To simplify this somewhat, I note that true age-ordering isn’t actually required – some absolute ordering on transactions is what is really needed to resolve conflicts in a way which permits forward progress, and the closer this ordering is to age-ordering, the more fair the resulting conflict-resolution policy will be.

The problem of assigning ages fairly to hardware transactions could be solved through the use of logical clocking [37]. While the best choice of logical clock implementation may vary
between specific architectures, whatever scheme is chosen should be low-overhead, should increment its clocks at a reasonably fine granularity (to keep CPUs with many transactions and those with few transactions nearly-synchronized) and should not require expensive interconnect broadcasts (to permit its use in directory-based coherence systems). An example of such a logical clock is the one used by Sorin et al. in SafetyNet. This proposal employs clocks with the same frequency in all CPUs, ensuring that the skew between them is sufficiently small [70]. For the purpose of assigning ages to transactions, a larger skew may be acceptable, for it affects only fairness, not correctness.

Such a mechanism for transactional age ordering provides unique and approximately-fair serial numbers for each transaction, but sending such serial numbers along with every transactional coherence message clearly introduces considerable overhead; that information should only be sent when a real transactional conflict may exist. To accomplish this, when a CPU with an active hardware transaction receives a request for some data it holds transactionally, it could reply with a \texttt{NACK} message which includes its transaction’s serial number. The requesting CPU may then decide what to do: if it’s running nontransactional code, or if it is running a transaction with a lower serial number than the requestee’s, it may send another message confirming the request and demanding the contended line; otherwise it can stall or abort its transaction. In this way, we avoid burdening the interconnect with unnecessary (and large, to avoid frequent counter overflows) transaction serial numbers.

The results for \texttt{labyrinth} given in Section 2.5.2 reveal another weakness to which best-effort HTMs may be prone. The lack of a hardware mechanism for early transactional release causes the bulk of \texttt{labyrinth}’s transactional execution to serialize in the HTM, resulting in no performance improvement as concurrency increases. The hybrids still perform reasonably, but only because they all fail over to USTM, which can perform nontransactional reads, and thus doesn’t need early release. In general, the performance of a hybrid system will not surpass the performance of both of its components; since the purpose of the best-effort HTM in a hybrid is to provide good common-case performance, compromises in its design will
yield compromises in overall hybrid performance. However, no amount of hardware design can compensate for a poorly-written transactional program: I believe that high-performance transactional programming requires the same care and mindfulness of the underlying systems as does high-performance lock programming.

2.6.2 Advanced Transactional Semantics

In the next chapter, I discuss I/O and other side-effecting operations in the context of transactional memory. In that chapter, I find that such operations can be supported within transactions by transactional memory systems by the use of advanced transactional semantics like compensation code, side-effect deferral, open nesting, and “going nonspeculative”. Such advanced semantics are very unlikely to be supported by best-effort hardware transactional memory systems, but could easily be deployed in a software transactional memory system; if this were done in the context of a hybrid transactional memory system like the UFO hybrid, performance would only be affected proportionally to the use of the new semantics.

Chapter 3 gives some concrete examples of this. As part of the development of the UFO hybrid, I have been considering the extension of its programming model to support system calls, I/O, and transactional waiting. Idempotent system calls (e.g., sbrk, gettimeofday) are already trivially supported by failing over to the STM. I use this support to handle the sbrks in malloc called within transactions; a feat achieved much less gracefully in the unbounded HTM by introducing complexity into its abort handler. By further adding support for deferring, “going non-speculative”, and compensation code the vast majority of side-effecting operations exhibited in real code can be supported. Such extensions are more straightforward because they only require modification of the STM.

A less concrete, but very compelling, example is supporting the transactional waiting primitive retry [31]. A transaction which determines, based on transactionally-read data, that conditions prevent its forward progress may issue the retry command. This
action undoes the transaction’s speculative writes, converts all its held otable entries to transactionally-read, and changes its state to retrying. The transaction may then deschedule itself. When a later transaction, while committing, updates a value that the retrying transaction had read, it awakens the retrying transaction, which releases its remaining otable entries and restarts as if after an abort.

In the context of a hybrid TM, such a retrying transaction amounts to a very long-running software transaction, with all the performance implications that this implies. Under PhTM, for example, no hardware transactions can complete as long as a retrying transaction is in flight. The UFO hybrid, however, could handle this quite naturally and with good performance. When the compiler generates the HTM-targeted version of the code, it could translate retry into an explicit abort, causing hardware transactions reaching that point to fail over to software. Upon reaching a retry, a software transaction would perform an action essentially similar to an abort, including undoing its speculative writes, but leave a stub in the otable for all held data items. It would also ensure that all this data is protected by a UFO fault-on-write bit, before descheduling itself. Software transactions which write its read data would then encounter the stub in the otable, and take the appropriate action. Hardware transactions, however, wake retrying transactions differently. A hardware transaction would detect a conflict with a retrying transaction when a UFO fault is raised; through inspecting the otable in the user-mode UFO fault handler (executed while in BTM), the ID of the retrying transaction could be recorded so that it can be awakened after the BTM commit. The transaction could then clear the UFO bit, relying on the fact that this update will not be visible until its commit and will be discarded if it aborts.
Chapter 3
I/O in Critical Sections

One aspect of transactional memory that remains unclear is how side-effecting operations – that is, those which cannot be transparently undone by a TM system – should be handled. This uncertainty poses a significant barrier to the general applicability and acceptance of TM. Further, the absence of transactional workloads makes it difficult to study this aspect.

In this section, I characterize the usage of I/O, and in particular system calls (which I refer to in this dissertation as *syscalls*), within critical sections in two large applications, exploring both the actions performed and the characteristics of the critical sections in which they are performed. Shared memory programs employing critical sections are the closest approximations available to non-trivial transactional workloads, so using this characterization, I attempt to reason about how the behavior I observed relates to the previous proposals for handling side-effecting operations within transactions. I find 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. I also find that while side-effecting critical sections are rare, they tend to be quite long-lasting, and that many of these critical sections perform their first syscall (and thus become side-effecting) relatively early in their execution. Finally, I show that while these long-lived, side-effecting critical sections tend to execute concurrently with many critical sections on other threads, I observe little concurrency between side-effecting critical sections.
3.1 Introduction

Transactional memory only provides clean semantics for speculatively accessing cacheable shared-memory storage locations, not side-effects such as I/O. Side-effects, simply defined as effects on logical system state which the TM system provides neither conflict detection nor rollback, are difficult to handle speculatively because they can violate isolation and atomicity. For example, when a transaction which has written to a file is aborted, the file has still been written to.

Interesting programs necessarily have side-effects, such as I/O and system calls. The interaction of these side-effects and TM must be considered if a code region containing side-effects requires isolated access to program state. In my analysis of two large multi-threaded programs, I have found that it is not uncommon for side-effecting actions to be contained in (lock-based) critical sections, and, at least for the bulk of the critical sections I analyzed by hand, the lock was providing isolated access to data, not just ensuring mutual exclusion.

To provide isolated access to a data structure in the presence of side-effecting operations, one has two choices: 1) to retain the use of locks to protect any data that are accessed by a critical section containing a side-effect; these locks would need to be acquired by any code that accessed those data, even strongly-atomic transactional code, or 2) to enable TM atomic regions to include side-effects, at least for the commonly occurring cases. The second approach has the potential to provide an overall simpler way to ensure isolated data access in the presence of side-effects.

While previous work has proposed a number of mechanisms for handling I/O [2, 10, 28, 31, 53, 54, 77], the paucity of significant TM workloads prevents me from evaluating these proposals in any meaningful way. The goal of this chapter is to try to understand how TM programmers will use side-effecting operations and, therefore, how support for them should be architected in TM systems. To shed light on this subject – in the absence of compelling TM workloads – I characterize the use of side-effecting operations, specifically syscalls, in two
existing, conventionally synchronized, multithreaded workloads. In particular, I characterize
and classify the kinds of side-effecting operations seen in my workloads, what minimum
protection they require, and the characteristics of the critical sections that contain them.

I first provide a view of transactional actions drawn from transactional databases (Section 3.2), and present a survey of proposed techniques for handling side-effects in transactions (Section 3.3). Then, on the supposition that contemporary critical sections will provide a lower bound (in size and composition) to the transactions of the future, I examine two large multithreaded workloads – MySQL and Firefox – to characterize the syscalls occurring in their critical sections. I find that most of the syscalls can be encapsulated so that programmers need not be aware of them. Finally, I characterize the critical sections invoking these syscalls themselves, describing some of their structure and examining the impact of syscalling transactions upon the whole program, with a view towards the amount of concurrency that may be lost by forcing syscalling critical sections to execute serially (Section 3.5).

3.2 Types of Actions

Transactional memory has strong roots in the earlier research in transactional databases, which share many properties with TM. When considering how TM might handle side-effecting actions in atomic regions, it is useful to consider a taxonomy for transactional actions presented by Gray and Reuter [25]:

Protected actions are those which can be completely compensated for by the transactional memory system. Guaranteeing both failure atomicity and isolation, these are operations which affect only CPU state and memory.

Unprotected actions are those for which the TM system cannot compensate, but for which the programmer may provide compensation code. With correct compensation code, these actions provide atomicity and may, though do not necessarily, provide isolation. For example, filesystem operations may be considered unprotected. It is important to note that
the selection of adequate compensation code, and indeed what constitutes such code, is left to the discretion of the programmer.

Real actions are those for which there is no adequate compensation. Gray and Reuter’s canonical example is “launch missile,” but more prosaically, printing a document, deleting a file, or even sending a message across a network may be considered real actions. Typically, real actions may only be executed when a transaction is known to be nonspeculative. If, however, the programmer is willing to ignore a spuriously printed document, or creates a transaction-safe network protocol, even these actions may be considered merely unprotected. In a real sense, what makes an action real or unprotected is merely what the programmer is willing to tolerate.

Drawing from this classification, I will use the term protection to refer to something enabling real or unprotected actions to be executed from within transactions: a transition to nonspeculative execution, a deferral of real or unprotected actions until the transaction is known to be nonspeculative, or the compensation block registered by the programmer on some piece of transactional code to be executed if the transaction aborts. I call any code which consists entirely of protected actions and unprotected or protected real actions transaction-safe.

3.3 Proposals for Handling Side-Effects in Transactions

I summarize several proposals which address unprotected or real actions in transactions:

- **Outlaw:** The most restrictive approach, used in STMs [31], as well as best-effort HTM systems like BTM 2.3.1, simply forbids any non-protected actions from occurring transactionally. While this approach offers simplicity by sidestepping the issue entirely, the limitations on programmability and composition that it places are probably
unacceptable for general use.

- **Defer:** In some cases, it may be possible to defer *unprotected* or *real* actions until the transaction is certain to commit [28, 30] or placing them in completion actions [77]. This may be done explicitly by the programmer, as in two-phase commit [2] or may be performed implicitly by the compiler or TM system. While this is suitable for some write-only actions or flushes, it is not a general solution because it prohibits dependences upon the return values of *unprotected* or *real* actions within a transaction, such as checking for an error status. Moreover, programmers expect their code to execute in program order; an automatic reordering could lead to unexpected effects.

- **“Go Nonspeculative”:** Another approach is simply to force transactions about to perform *unprotected* or *real* actions to “go nonspeculative” by acquiring a global commit token [10, 28]. While this technique can accommodate even the most irreversible of *real* actions handily, it does have some drawbacks. Under this regime, an *unprotected* or *real* operation inside of a transaction amounts to an implicit promise, by the programmer, that the transaction will complete. The TM system can be made to guarantee that it will not abort this transaction, but this can limit concurrency by serializing side-effecting transactions, negatively affecting performance. Furthermore, the transaction must not have an explicit abort operation along any execution path after the first *unprotected* or *real* action. Because the promise to complete was implicit, and the *unprotected* or *real* action may have been called deep within library code, programmers may be left uncertain of where explicit abort operations may be used. This in turn affects the application of language-level techniques like *retry* and *orElse* [31].

- **Compensate:** The last, and most complex, approach is to permit the programmer to protect *unprotected* code by associating a compensation block with the *unprotected* code [2, 30, 53, 54, 77], not unlike the use of *catch* blocks to guard possibly-excepting code. This approach permits programmers to identify the appropriate compensation
for their unprotected actions, allows transactions with unprotected actions to execute concurrently, and also permits explicit aborts. However, it also introduces a new source for potential bugs, and does not implicitly provide isolation or conflict detection on unprotected code.

As I will show in Section 3.4, no one of these techniques clearly subsumes the others with respect to supporting all side-effects in all possible transaction code, or even in the workloads I studied. Outlawing side-effects in transactions is problematic, as discussed in Section 3.1. If it is not feasible to prohibit all side-effects in atomic regions, then atomic side-effects must be performed in lock-based critical sections. If, however, there is any intersection between the data accessed in transactions and the data accessed in critical sections, then all such transactions will also have to acquire the relevant locks – which can be fraught with difficulty [69]. Deferral can only be used when no return values from side-effecting operations are computed upon later in the transaction. “Going nonspeculative” precludes explicit aborts (and raises difficulties with combining locks and transactions.) Compensation cannot be applied whenever the programmer cannot provide an adequate compensation block. Indeed, all are necessarily incomplete solutions, for some transactional code is simply untenable – consider the case of a transaction which performs a real action and later issues an explicit abort.

In this section, I am concerned with the relative suitability of these approaches. I raise the following questions:

- How common are side-effecting operations in critical sections? In other words, how much code would simply outlawing side-effects in transactions affect?

- What kinds of side-effecting operations are seen in critical sections? How much of it must be considered real actions, and how much unprotected?

- How often are side-effecting operations performed at the end of critical sections? Side-effecting operations performed at the end are perhaps less likely to return a value that
the transaction operates on, so they are more likely to be deferrable.

- If “going nonspeculative” is required, how early the transaction must do so affects how much concurrency the transaction will permit. How are side-effecting operations distributed throughout the lifetime of side-effecting critical sections, and how long are these critical sections? To what degree do side-effecting critical sections overlap with other critical sections?

- What kinds of compensation do the side-effecting operations in my workloads’ critical sections require?

3.4 Side-Effects in Critical Sections

For this analysis, I sought out large, complex, multithreaded programs in which I/O and other side-effects is reasonably expected. After exploring a number of programs, I selected two workloads that had non-trivial amounts of side-effecting operations called from within critical sections: MySQL and Firefox. MySQL is a multithreaded SQL database server; my test installation of MySQL uses the InnoDB storage engine. MySQL has a main connection thread and creates a new thread for each client connection it receives, and InnoDB was configured to permit up to 50 concurrent I/O threads. My profiled runs used SysBench [1] first to prepare (create and populate) a small database, and then to access it. Firefox is a popular web browser. My profiled runs started Firefox up in a pre-set profile, loaded a series of web pages, then shut it down.

I/Os and Syscalls

Having selected my workloads, I must specify what side-effects I sought. In Section 3.1, I defined side-effects as operations which affect system state in a way for which the TM system cannot automatically extend isolation and atomicity. TM systems guarantee isolation and
atomicity to CPU state and memory, so in non-transactional code like my workloads side-effecting operations are device I/O.¹

In x86 code, there are three ways in which I/O may be performed:

- The `in` and `out` instructions permit communication directly with a port, as configured by `ioperm()` or `iopl()`.
- Some system calls can delegate to the kernel to perform I/O work.
- Memory-mapped I/O permits the mapping of memory spaces to devices or files.

The `in` and `out` instructions in application code are quite uncommon in both MySQL and Firefox, and are not seen at all inside of critical sections. Memory-mapped I/O generally happens in kernel code, which (for reasons I shall address shortly) should not be transactionally executed in user-level transactions. One exception to this is X11, which among my workloads only applies to Firefox. However, as my Firefox delegates all its X11 rendering to a single thread, I do not consider memory-mapped I/O. System calls, on the other hand, are plentiful across all threads, as well as within critical sections – but not all actually perform I/O. All, however, execute kernel code.

Current software transactional memory systems (STMs) cannot execute kernel code transactionally, and so in STMs all syscalls are side-effecting, whether or not they actually generate I/O. Nor is it clear that this inability is a weakness of STMs. Zilles and Flint have argued against transactional execution of system calls, reasoning that conflicts on kernel data structures will not only reduce concurrency in the transactional application, but will compromise performance isolation, which could negatively impact other running applications [78].

Zilles and Baugh [77] and Moravan et al. [53], observing that it is untenable to execute syscalls transactionally, have proposed wrapping them in nontransactional regions. These

¹Several techniques have been proposed which could make side-effects of changes to CPU state or memory, by enclosing these changes in paused regions or open nests [53, 54, 77].
regions, called \textit{paused regions} in the former paper and \textit{escape actions} in the latter, are blocks of code inside transactions that are not executed transactionally. Their memory footprints do not contribute to their enclosing transaction’s, and no automatic compensation is provided for them\footnote{However – as in \textit{open nesting} \cite{54} – it is possible to register blocks of abort (compensation) or commit code, for executing on the abort or on the commit of the parent transaction.}. I assume that all syscalls in transactions will be executed in such regions, and that consequently all syscalls will be side-effecting.

\subsection*{3.4.1 Experimental Method}

I profiled the workloads using the Pin binary instrumentation tool \cite{41} on an Intel Core 2 Duo dual-core processor. Because these programs are pthread-based, my Pin module tracked \texttt{pthread_mutex} acquires and releases and recorded I/O behavior, particularly syscalls, within these critical sections. For the purposes of my analysis, I assume that all of the critical sections I observed would be implemented as transactions, had the application been written for TM. I believe this is a fair assumption because, while locks can be used for mutual exclusion as well as isolated data access, my source inspection led me to believe that isolated access to data was a motivation for the subset of critical sections that I inspected in detail.

I measured durations using wall-clock time, using \texttt{gettimeofday()} with a granularity of one microsecond. When measuring time in Pin-instrumented code, it is necessary to consider the overhead incurred by the instrumentation itself. When tracking durations, I discounted time spent in instrumentation code by calculating the difference between the entrance to an instrumentation block and its exit, then subtracting that from the recorded duration of the instrumented critical section.

Critical sections, like transactions, may nest. For the purposes of this analysis, I only consider toplevel critical sections, which I abbreviate to TCSs. I abbreviate those TCSs which perform syscalls as \textit{syscalling-TCSs}.

In my analysis, I found it useful to determine the degree of concurrency, or \textit{overlap},
Figure 3.1: A Syscalling-TCS Overlapping with TCSs

Right-pointing triangles are lock acquires; left-pointing triangles are lock releases. The star marks the first syscall in its syscalling-TCS. c is the global TCS retire counter.

experienced by syscalling-TCSs. This metric, which reflects the number of other TCSs that retired between a syscalling-TCS’s first syscall and its retirement, is gathered by a mechanism shown in Figure 3.1. I employ a global toplevel critical section retire counter, called c, which every TCS increments upon retiring. In the Figure, the syscalling-TCS on thread t₀ sees four TCSs retire in its lifetime. On its first syscall, it reads c, finding it to be 4. When the critical sections on threads t₁ and t₂ retire, they each increment c by one; the nested critical section on t₂ is not toplevel and so does not increment c. When the syscalling-TCS in t₀ closes, and before it increments c, it subtracts c’s current value from the value read at its first syscall. Correspondingly, I say that the overlap of the syscalling critical section on thread t₀ is 2. My dual-core processor may artificially limit the overlap seen, so to maximize the amount of overlap I observed I inserted nanosleeps before every I/O operation (thus prompting the CPU to switch threads). When tracking TCS and syscalling-TCS durations, I discounted time spent in nanosleeps.
### 3.5 IO Results

I examine the syscalls made from within critical sections from two perspectives. First, I examine the syscalls themselves to understand what is being called and with what frequency and to discover what techniques can be employed by kernel or library developers to render these syscalls transaction-safe. Next, I examine the context of the syscalls to discover the higher-level behavior of critical sections that contain syscalls.

#### 3.5.1 Syscall Sites

Previous work characterized syscalls inside critical sections for a selection of workloads, dividing them into read and write operations and determining the frequency of critical sections performing them [17]. I delve deeper into the kinds of syscalls seen in my workloads, determining what calls are being made, what they are doing, and what minimum protection they require.

In Table 3.1, I list the syscalls dynamically detected in critical sections in my workloads, divided into six categories: filesystem, process memory, process maintenance, system info, communications, and system info.
Figure 3.2: Distribution of Syscalls across Syscalling-TCSs
Syscalls are distributed throughout syscalling-TCSs, and are more frequent towards the end of the critical sections. Aggregated results for all syscalling-TCSs in (a) Firefox, (b) MySQL. Bars are per-bin with axis on left; line is cumulative with axis on right.

Syscalls are distributed throughout syscalling-TCSs, and are more frequent towards the end of the critical sections. Aggregated results for all syscalling-TCSs in (a) Firefox, (b) MySQL. Bars are per-bin with axis on left; line is cumulative with axis on right.

Syscalls are distributed throughout syscalling-TCSs, and are more frequent towards the end of the critical sections. Aggregated results for all syscalling-TCSs in (a) Firefox, (b) MySQL. Bars are per-bin with axis on left; line is cumulative with axis on right.

time, and communication. Some of the syscalls listed belong to more than one category – for example, read may be applied to a file handle as well as to a socket. Such syscalls are marked with an asterisk. At the right side of the table, I show the relative dynamic frequency of each category of syscall in each workload. It is notable, in light of the observations I make in the next section, that very few communication syscalls are seen in either workload.

3.5.2 Syscall Protection – The Advantage of Compensation Code

As I have suggested, not all of the syscalls that I observed require the same treatment to become transaction-safe. I found four “protection classes” among the syscalls I observed:

- **Null compensation**: Some syscalls require no protection, as their speculative execution does not logically change system state. Time syscalls like gettimeofday fall in this class, as may syscalls like pread, which reads from a file without altering the file pointer. If side-effects are handled by forcing transactions to become nonspeculative, this class of syscalls will not require that. If side-effects are handled by compensation code, a null compensation block will be sufficient. Over 70% of the dynamic syscalls
in Firefox critical sections fall into this category; under 10% of MySQL’s do.

- **Memory-fixup**: Many syscalls’ only side effect is a change of kernel state. For example, `lseek` does not affect a file directly, but instead adjusts a file offset pointer within a file handle data structure. Since kernel code is not executed transactionally, these actions must be considered as side-effects. If the TM system provides a mechanism for registering compensating code, then this work may be easily done speculatively; otherwise this side effect will necessitate either that the transaction go nonspeculative, or that the call be deferred until commit.

- **Full compensation**: Many observed syscalls perform unprotected I/O actions and will require “going nonspeculative” or compensation code. For example, a transaction with an `open` call which creates a file will register a corresponding `unlink`, an `append` call might register a corresponding `truncate`\(^3\). If the filesystem has transactional support, then the compensation blocks might simply force a filesystem transaction abort.

- **Real actions**: A small minority of the syscalls I saw cannot be adequately compensated at the scope of the syscall. Some process maintenance syscalls like `tgkill`, and communications syscalls like `socket`, `pipe`, and `read` or `write` to sockets or pipes cannot be executed speculatively without knowledge of the context of the call. These syscalls are the most difficult to encapsulate so that programmers at higher levels of abstraction may use them speculatively, but some possibilities exist. Buffering support in the manner of ReVive I/O may be employed [55], but unlike that work, the amount that may need buffering is unbounded. Alternatively, if it is possible, compen-

\(^3\)While these compensations do not provide isolation from the rest of the system, it is not clear that this matters in most cases. For example, a critical section in MySQL responsible for database creation first created a directory for the database, then attempted to create an options file inside that directory. If the file creation failed, the directory was deleted again, violating system-wide isolation – but the programmers were willing to accept this result. A transactional filesystem can provide true isolation for filesystem side-effects, if needed.
sation may be provided at a higher level of abstraction by the application or library developer. For example, a network program may use a transaction-safe protocol, in which messages may be tentatively issued and revoked later – or it might not, and this information would not be available at the level of the system call. In these cases, the only choice is to wait until the transaction is non-speculative before executing the syscalls – or permitting compensation code to be registered at a higher scope. This class corresponds to the Communication category described in Table 3.1, as well as some process management syscalls like `clone` and `tgkill`. It comprises about 7% of the syscalls in MySQL’s critical sections and a minuscule component of Firefox’s.

Importantly, all these categories except for the last may be compensated for at the level of the syscall – and the last category represents very few of the dynamic syscalls I encountered. It is clear that with adequate compensation the bulk of dynamic syscalls can be rendered transaction-safe and, thus, speculatively executable. For the workloads I examined, written in C/C++, this could be accomplished simply by providing compensation code in the system library (e.g. libc); if this were done, my workloads could be *transactified* with nearly all syscalls in transactions handled transparently to the application developers.

![Distribution of TCS Durations](image.png)

**Figure 3.3: Distribution of Syscalling-TCS Durations (in µsec)**

*TCSs with syscalls trend much larger than those without. Black bars are TCSs without syscalls; grey bars are syscalling-TCSs. Results for (a) Firefox, (b) MySQL*
Figure 3.4: Distribution of First Syscalls across Critical Sections

A significant fraction of first syscalls occur well before the end of the critical section. Aggregated results for all critical sections in (a) Firefox, (b) MySQL. Bars are per-bin with axis on left; line is cumulative with axis on right.

3.5.3 Syscall Context

I have examined the types of syscalls executed from critical sections in my workloads; now I examine their context, exploring the structure of critical sections which perform syscalls. I will attempt to characterize side-effecting transactions by their frequency, by the distribution of syscalls within them, by their (temporal) length, and by the degree of concurrency they expose.

Previous research on other applications has shown that syscalling critical sections are dynamically very rare in multithreaded workloads [17]. I find the same: in Firefox, only 0.71% of dynamic critical sections issue syscalls; in MySQL, the proportion is even smaller: only 0.02%. However, further examination of these critical sections shows that it may not be wise to dismiss them as too rare to matter.

Figure 3.3 shows the approximate durations, in μseconds, of toplevel critical sections – both those which execute syscalls (the syscalling-TCSs) and those which do not – in my workloads. In both workloads, I find that critical sections performing syscalls tend to be much longer than those which do not. To what degree this reflects an intrinsic quality of
atomic regions which perform syscall, and to what degree it reflects the cost of switching into kernel mode, matters less than the fact that critical sections performing syscalls tend to be quite long. The longer transactions last, the more chance they have to affect the performance of other transactions in the same application.

3.5.4 The Applicability of Deferral

In Figure 3.2, I show where, in the progress of syscalling-TCSs in my workloads, syscalls are executed. In these graphs, every syscall executed within a syscalling-TCS (including syscalls indirectly called by nested children of the syscalling-TCS) increments the bar corresponding to that syscall’s position in the lifetime of its syscalling-TCS. These graphs show that while syscalls do tend frequently to be positioned near the end of their syscalling-TCSs, they exist in significant numbers throughout the lives of syscalling-TCSs. This has several ramifications for the TM proposals reviewed in Section 3.3. For techniques which defer syscalls until their transaction is validated, the earlier a syscall happens, the longer it must be deferred, and the more significant its implicit reordering becomes. Furthermore, as an operation may not be moved after any of its consumers, any syscall in a TCS which produces values that are consumed later in the TCS cannot be deferred. I analyzed syscalling-TCSs responsible for 90% of the dynamic syscalling-TCS instances in my benchmarks, and found that over 96% of those in MySQL, and 100% of those in Firefox, consumed the result of the first syscall in the TCS – suggesting that deferral will not apply in the preponderance of cases. For the technique of forcing transactions to “go nonspeculative” prior to executing any syscall, a more serious problem awaits.

4That is, known to be safe to commit and correspondingly nonspeculative
Figure 3.5: Degree of Overlap of Toplevel Syscalling Critical Sections (syscalling-TCSs) and Toplevel Critical Sections (TCSs)

All syscalling-TCSs in (a) Firefox, (b) MySQL. This cumulative plot shows (black line) how many syscalling-TCSs (y-axis) overlap (x-axis) or fewer TCS; (grey line) how many syscalling-TCSs (y-axis) overlap (x-axis) or fewer other syscalling-TCSs

3.5.5 The Overhead of “Going Nonspeculative”

In Figure 3.4, I show the distribution of the positions of first syscalls within the durations of their syscalling-TCSs. In these graphs, the first syscall executed within a syscalling-TCS (including syscalls indirectly called by nested children of the syscalling-TCS) increments the bar corresponding to that syscall’s position in the lifetime of its syscalling-TCS. The figure shows that while a significant number of syscalling-TCSs have their first syscall near the end of their lives – over 50% in the last 10% of Firefox syscalling-TCSs – many execute their first syscall relatively early. If transactions attempting syscalls must first “go nonspeculative”, then from that point – the point of the first syscall – until their commit, no other transaction may become nonspeculative. This Figure shows that at least one third of syscalling-TCSs in both workloads see their first syscall before they are halfway finished. As the barrier to concurrency is a problem proportional to the length of syscalling-TCSs, which tend large (as shown in Figure 3.3), this represents considerable potential for even a relatively small number of toplevel side-effecting transactions to impact concurrency throughout applications.

These large transactions translate to a significant degree of overlap (described in Sec-
tion 3.4.1.) The overlaps of syscalling-TCSs in my workloads are shown in the black plots in Figure 3.5. While a majority of syscalling-TCSs overlap with no other TCS – two-thirds in Firefox and half in MySQL – a significant minority have substantial overlap. 15% of the syscalling-TCSs in Firefox overlap with 360 or more TCSs; if the presence of a nonspeculative transaction precludes any other transactions from committing (e.g., by holding a “commit” token), this betokens a significant loss of concurrency throughout both these applications.

Blundell et al. suggest that the nonspeculative transaction (the unrestricted transaction in their parlance) need not block speculative (restricted) transactions from retiring – as long as there is only one nonspeculative transaction active at a time, and it may never be aborted [10]. Their restricted transactions are not only constrained from “going nonspeculative”, but are also bounded in time and memory footprint. However, the memory and time bounds are not strictly required; there is no reason that speculative transactions of any size or duration might not retire even while another transaction is nonspeculative, so long as the speculative transactions do not conflict with the nonspeculative. In this case, it is instructive to consider, as a lower bound to syscalling-TCS overlap, the overlap that syscalling-TCSs have with other syscalling-TCSs. This metric, measured in the grey plots in Figure 3.5, is similar to that shown in Figure 3.1 (Section 3.4.1), except that not every retiring TCS increments the global TCS retire counter – only retiring syscalling-TCSs may. As only one syscalling-TCS is allowed to be nonspeculative at a time, the overlapped critical sections in the grey plots in Figure 3.5 cannot be executed concurrently under a regime in which syscalling-TCSs “go nonspeculative” – but the loss of concurrency is much less than that seen in the black plots in Figure 3.5: over 93% of Firefox syscalling-TCSs overlap with 10 or fewer other syscalling-TCSs, and only 2% of MySQL syscalling-TCSs overlap with any other syscalling-TCSs at all, none with more than 4 others.

I expect these results to be a lower bound of the actual concurrency available in transactional versions of these workloads. Atomic regions which, in lock-based code, would be guarded by different locks, will, of course, when transactional, not conflict (for if they did,
the lock-based code would have race conditions). However, some atomic regions which in lock-based code would be guarded by the same lock might as transactions not conflict.

3.6 Concluding Remarks

While transactional memory is a promising technique for achieving concurrency in synchronized code, two of its oft-cited advantages – composability and programmability – are compromised by the difficulty of speculatively executing side-effecting code. In this chapter, I have examined the side-effecting operations, particularly the syscalls, performed in critical sections in two large, multithreaded workloads. I have classified the kinds of syscalls thus performed, noting that the presence of correct compensation code and a transactional filesystem permit nearly all syscalls to be executed speculatively. Transaction-safe system libraries, linked to transactional filesystems, could enable application programmers to speculatively invoke, directly or through composition, nearly every syscall.

I have also examined the contexts of syscalls in critical sections in my workloads, observing that those critical sections which do perform syscalls do so throughout their lifetimes, and that there is a strong tendency for results of syscalls in critical sections to be used within those critical sections. This tendency may limit the usefulness of deferring I/O until the end of its enclosing transaction. I have also observed that the lifetimes of syscalling critical sections tend to be quite long compared to those of other critical sections. Considering the effect that long syscalling transactions might have on overall concurrency available in applications, I examined the amount of overlap syscalling critical sections have with other critical sections. I observe that the technique of “going nonspeculative” results in substantial loss of concurrency when not applied carefully, but that this loss can be dramatically reduced if nontransactional transactions only preclude the retirement of other conflicting transactions or other transactions which perform syscalls.
Chapter 4
Conclusion

In the years leading up to this dissertation, transactional memory has garnered a great deal of research attention. Research on transactional memory has increasingly appeared in prominent conferences on many topics, from object-oriented programming to high-performance computing; major architecture conferences have begun to include sessions on transactional memory; and at least one new workshop, the ACM SIGPLAN Workshop on Transactional Computing (TRANSACT), has arisen to facilitate the dialogue on transactional memory. It is a fertile research area, with proposals and findings in the areas of automatic parallelization, hardware and software TM system design, advanced transactional semantics which extend the programming model available to the user and enable entirely new ways of programming, language support for transactions, dynamic speculative optimization, and the theory of transactions and speculative execution. Hardware transactional memory has become a reality, with the 16-core Rock CMP from Sun offering HTM support. Notably absent, however, in this otherwise teeming field of study are parallel applications which use transactions to control their concurrency.

I believe that there are two reasons for this absence:

• **Performance:** Programmers who write parallel programs do so in the expectation of improving performance over serial execution; at the moment, the performance of software TM systems doesn’t generally approach that of tuned conventional locks, as attested by Dice and Shavit [20] and Keir Fraser [24] among others. Transactional memory performance will need to improve significantly before the programming model tempts parallel programmers, but hardware support seems necessary for real perfor-
• **Programmability:** Even if programmers were willing to accept reduced parallel performance, an agreed semantic for transactional memory – both plausible to implement in hardware or software and useful for transactional programmers – does not appear to have emerged. An agreed semantic is absent not only at the frontiers of transactional memory – where proposed advanced transactional semantics like compensation code, `retry` and `orElse`, or open nesting may or may not ultimately be of use – but is also absent at the very heart of transactional programming: operations like data privatization and syscalls which are straightforward and commonplace in lock-based code have unspecified or undesirable behaviors in most transactional memory systems. Until parallel programmers can depend upon a useful transactional semantic, I believe they are unlikely to risk developing transactional code.

When considering the current state of transactional memory, it seems to me that the TM community faces a bootstrap problem. Lacking good-quality transactional workloads, there is little incentive to provide significant dedicated hardware support for transactional memory – and few good ways to really gauge the performance of any transactional memory system. Without an active community of transactional programmers, it’s difficult to know what the best transactional semantics are. But without transactional performance and programmability, good transactional workloads and engaged transactional programmers will not appear.

In this dissertation, I have striven to narrow the gap between transactional memory as a research topic and transactional memory as a programming tool. I have proposed a new hybrid transactional memory system, the UFO Hybrid, which performs similarly to an idealized, unbounded hardware TM system, and outperforms the other current hybrid TM proposals. The UFO Hybrid is also strongly atomic, allowing conventional locking techniques like privatization – a feature not offered by other hybrid systems, and only by a
few software TM systems. The UFO Hybrid supports transactional programmers by offering HTM-like performance for common-case transactions while supporting whatever advanced transactional semantics are provided by the STM – which can be modified or extended far more easily than can hardware. It supports transactional memory system developers by placing few demands on its constituent HTM – only that it be able to run most transactions with high performance. If successful in these goals, it would also support TM researchers by encouraging active transactional workload development.

I have also tried to develop transactional memory as a viable concurrent programming model by addressing the issue of side-effects within transactions. As speculatively-executed units of code, any work done in transactions may need to be revoked later. While this is not problematic for transactional accesses to memory or CPU state, side-effects can be quite difficult to accommodate transactionally. Yet composability, a frequently-lauded benefit of transactional memory, implies that transactional programmers may feel free to compose any work at all into a transaction, without having to know anything about the work. Transactional side-effects pose a serious threat to transactional composability. In this dissertation, I have attempted to measure transactional side-effects, classifying them into protection classes and characterizing the atomic regions which contain them. With these data, I have then drawn conclusions about what transactional memory systems will need to do in order to effectively support side-effects, and preserve transactional composability.
References


[38] Stefan M. Larson, Christopher D. Snow, Michael R. Shirts, and Vijay S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. Computational Genomics, 2002.


Author’s Biography

Lee Baugh was born in Texas in 1977, and grew up in Texas and North Carolina. He attended the Texas Academy of Mathematics and Science from 1993 to 1995; upon graduating with a high-school diploma, he continued his education at the University of Texas at Dallas, earning the Bachelor of Science (1997) and the Master of Science (2000). He commenced the Ph.D in August of 2001 at the University of Illinois, where he was awarded the Illiac and Carver fellowships. During the course of his Ph.D, he interned at Microsoft and Intel. Parts of the work presented in this dissertation were presented at the Second ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT-07), the 2008 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS-08), and the 35th International Symposium on Computer Architecture (ISCA 2008).