A Study of Hybrid Transactional Memory

Fu'ad W. F. Al Tabba'

A thesis submitted in partial fulfillment of the requirements of
Doctor of Philosophy in Computer Science,
The University of Auckland, 2011.

Abstract

The rise of multicore processors is driving programmers towards parallel programming. Traditional lock-based parallel programming, however, breaks abstraction, hinders composition, and is further complicated by issues such as deadlock, priority inversion, and lack of scalability. Transactional memory, a promising programming model inspired by database transactions, is gaining popularity as a way to overcome the drawbacks of lock-based programming and make it easier to write parallel programs.

Different methods have been proposed for supporting transactional memory. Hardware proposals, which modify a processor's existing hardware to support transactions, are either too complex, or cannot handle all types of workload. Software proposals, which do not require any hardware support beyond what is already present for parallel programming, are too slow. This thesis argues that transactional memory should be supported by a hybrid combination of hardware and software. By combining the two, transactional memory can offer the best of both hardware and software. To support this argument, this thesis presents my work on transactional memory, which spans the areas of hardware, software, and hybrid support.

This thesis presents NZSTM, the first nonblocking, object-based, software transactional memory that does not require indirection to access data in the common case. It also presents NZTM, a hybrid transactional memory that uses NZSTM as its software component. The evaluation presented shows that nonblocking support introduces little overhead compared with blocking algorithms, and that NZTM is competitive with pure hardware transactional memory.

Furthermore, this thesis investigates how to improve the performance of hardware transactional memory by using data speculation in transactions, and how to reduce transactional conflicts by decoupling them from cache coherence conflicts. Most hardware proposals do not distinguish between transactional conflicts and coherence conflicts, leading to false transactional conflicts. This thesis explains how to mitigate the effects of coherence conflicts by using value prediction in transactions. It also shows that coherence decoupling and value prediction in transactions complement each other, because they both speculate on data in ways that are infeasible in the absence of hardware transactional memory support.

Dedication

To my parents, Samar and Wael, and to my partner in life, Anna,
I couldn't have done it without you!

Acknowledgements

They say that no man is an island, and good things come to those who wait.
— Jon Bon Jovi

The wait is almost over; to that I am indebted to many people, whom without this thesis would not have been possible. I was very lucky to have three great advisors, formidable collaborators, and a patient wife, all of whom have contributed directly to this thesis.

First and foremost, I thank my advisor, Jim Goodman, for all his guidance and support throughout the years. Jim contributed to all levels of the work presented in this thesis, whether it is the high level discussion sessions where we would develop an idea (*or shoot it down*), debugging a model (*Jim's ability to identify a bug in my code based only on my description of it is uncanny*), and helping me write this thesis (*I will never forget what the subjunctive mood is*).

I am also very grateful to Mark Moir, my unofficial co-advisor. Mark's contributions were instrumental to the ideas presented in this thesis, particularly to NZSTM and NZTM. Without Mark, NZTM would not have had its snazzy name! I learned a lot from Mark on how to write and publish a research paper. I am also grateful for Mark's comments on this thesis.

I thank Gill Dobbie, my co-advisor, for her help and advice, for the thought-provoking discussions we have had, and for her comments on this thesis.

James Wang, my first friend in the trenches, I thank him for showing me the ropes when I started out. James set up the original simulation environment, which I continued using throughout my work. The (*often heated*) discussions James and I have had were very helpful in developing many of the topics presented in this thesis.

To Andrew Hay, I am grateful for our collaboration, which resulted in the work on value prediction in transactions. I am not sure if either of us will miss those long hours of working on (*banging our heads against*) the simulator, or the waiting for and graphing all the results — only to discover that we need to redo the whole thing! I am also grateful to Andrew for his comments on this thesis.

I am very grateful to the awesome team at Sun Labs at Oracle for their valuable comments and for their help on the ATMTP simulator and the Rock machine: Kevin Moore, Dave Dice, Marek Olszewski, and especially to Dan Nussbaum for his help and patience in answering every one of my missives.

I thank Doug Burger, Guri Sohi, and Kevin Moore for their feedback on the work on value prediction. I especially thank Doug for his suggestions on how to extend that work to enable eager conflict detection.

I am grateful to Virtutech for the Simics simulator license, and to Sun Labs at Oracle for access to the Rock machine and for the Niagara machine donated to the University of Auckland. I am also very grateful to Education New Zealand and the University of Auckland for funding my research.

Last but not least, I thank my lovely wife, Anna Ogenblad, for actually reading and commenting on my whole thesis. Anna does not have a computer science background, so I can only imagine how boring it must have been — she would always read it right before sleeping... *Jag älskar dig Anna!*

Contents

List of Figures

List of Tables

Glossary

ATMTP: Adaptive Transactional Memory Test Platform, a simulator that models Rock-like hardware transactional memory (page 23)

Compare&Swap: An atomic processor instruction that reads a memory location, and if the value at that location matches the value the instruction expects to find, it replaces it with a new value, indicating whether the operation was successful (page 13)

CPS: Checkpoint Status register, a register in Rock that provides feedback on what caused a hardware transaction to abort (page 23)

DPTM: Decoupling and Prediction Transactional Memory, a best-effort hardware transactional memory proposal that mitigates the effects of false conflicts in transactions (page 118)

DSTM: Dynamic Software Transactional Memory, an early software transactional memory proposal (page 143)

GEMS: General Execution-driven Multiprocessor Simulator, an extension to the Simics simulator that allows detailed architectural timing simulation, as well as the modeling of different hardware transactional memory proposals (page 26)

GIL: Global Interpreter Lock, a mutual exclusion lock used in the de-facto standard Python interpreter implementation (page 101)

HTM: Hardware Transactional Memory (page 17)

NZSTM: Nonblocking Zero-indirection Software Transactional Memory (page 53)

NZTM: A hybrid Nonblocking Zero-indirection Transactional Memory (page 81)

Promela: A C-like language that describes models to check using Spin (page 73)

Rock: A multithreaded, multicore, SPARC processor developed at Sun Microsystems, and the first commercial processor designed with hardware transactional memory support (page 23)

SCSS: Single-Compare Single-Store, an atomic operation that modifies a location with a new given value only if another location's value matches an expected value (page 67)

Simics: A full-system functional simulator that allows for detailed timing simulation using extensions (page 26)

Spin: A mechanical verification system designed for the formal verification of high-level models for systems of concurrently executing processes (page 73)

STM: Software Transactional Memory (page 19)

TLB: Translation Lookaside Buffer, a cache for page table entries that speeds up virtual address translation (page 45)

Chapter 1

Introduction

> *The distant threat has come to pass. For 30 years or more, pundits have claimed that parallel computers are the inexorable next step in the evolution of computers and have warned us to learn to program these machines.*
>
> *— Larus and Rajwar [2007]*

The age of parallel computing is finally here. Although Moore's law is still driving the number of transistors higher, the actual improvement in processor design is slowing down. Processor manufacturers find themselves with the easier option of using these additional transistors to add more cores to a microprocessor, rather than build a bigger, more powerful, processor. This trend is pushing programmers towards an unfamiliar, parallel, programming paradigm.

Parallel programming is more complex than sequential programming. Parallel programming requires programmers to be able to manage all the available computing resources, and reason carefully about possible interactions of their programs when running concurrently. If programmers do not carefully consider these interactions, their programs' performance — and even correctness — could suffer.

Transactional memory [Herlihy and Moss, 1993] is one proposal that promises to make it easier to reason about parallel programs. Transactional memory is a programming model that provides a level of abstraction on top of critical sections by using transactions, a concept borrowed from the database community. Transactions promise to make it easier to reason about each critical section individually, without having to worry about the possible interactions between critical sections.

Support for transactional memory has been proposed in software, hardware, and as a hybrid combination of the two. Software support is more flexible than hardware support, but is significantly slower so far. Hardware support is fast; however, implementing a flexible hardware solution that can support the same workloads as software is expensive, or altogether infeasible.

This thesis argues that the hybrid approach is the most appropriate for the transactional model to succeed as a programming model. By combining both software and hardware support, the hybrid approach can overcome their individual weaknesses.

This chapter introduces the problems that led to the development of transactional memory as a programming model, and presents my contributions and the outline of this thesis.

1.1 Motivation: Transactions in the Multicore Age

> *To pull a bigger wagon, it is easier to add more oxen than to grow a gigantic ox.*
> *— Gropp, Lusk, and Skjellum [1999]*

In 1965, Gordon Moore published his article introducing what is now known as *Moore's law*. Moore's law predicted that the amount of complexity, or number of transistors, that can be used inexpensively in a chip will double every two years. Because this would double the number of transistors processor designers have at their disposal, Moore's law implies that processor performance would also improve at an exponential rate [Joy, 1995].

For the forty years following Moore's initial prediction, performance has indeed kept up with the increase in the number of transistors on a chip [Hennessy and Patterson, 2006]. More recently, however, improvements in performance seem to be slowing down [Sutter, 2005].

Figure 1.1 presents a history of the trends in Intel processors. It shows the trends in the number of transistors, the clock speed, power consumption, and performance as measured by the number of instruction running per clock cycle. The figure shows that, since 2003, the exponential growth in single-core performance has all but stopped; the same is true for the increase in clock speed — despite the number of transistor still growing exponentially.

Power consumption is one factor that accounts for the change in the performance trend: as the clock speed increases, power consumption also increases, making processors that operate at high frequencies difficult to cool down and more expensive to operate [Hennessy and Patterson, 2006]. Another problem is that as the field of processor design matures, processor designers find it increasingly difficult to use the additional transistors to improve the performance of a single processor. The economics of replication make it more appealing to use these additional transistors to create multiple processors, e.g., by adding more cores to the same chip. Assuming programmers use these additional cores efficiently, adding cores would be an easy solution that improves performance without increasing the processor clock speed.

Processor manufacturers have embraced the concept of adding multiple cores [Geer, 2005; Koch, 2005]. Today, virtually all desktop and laptop computers have at least two cores, gaming consoles have up to eight cores on each processor [Chen, Raghavan, Dale, and Iwata, 2007], and Intel is warning us that future processors would have thousands of cores on a single chip [Ghuloum, 2008]!

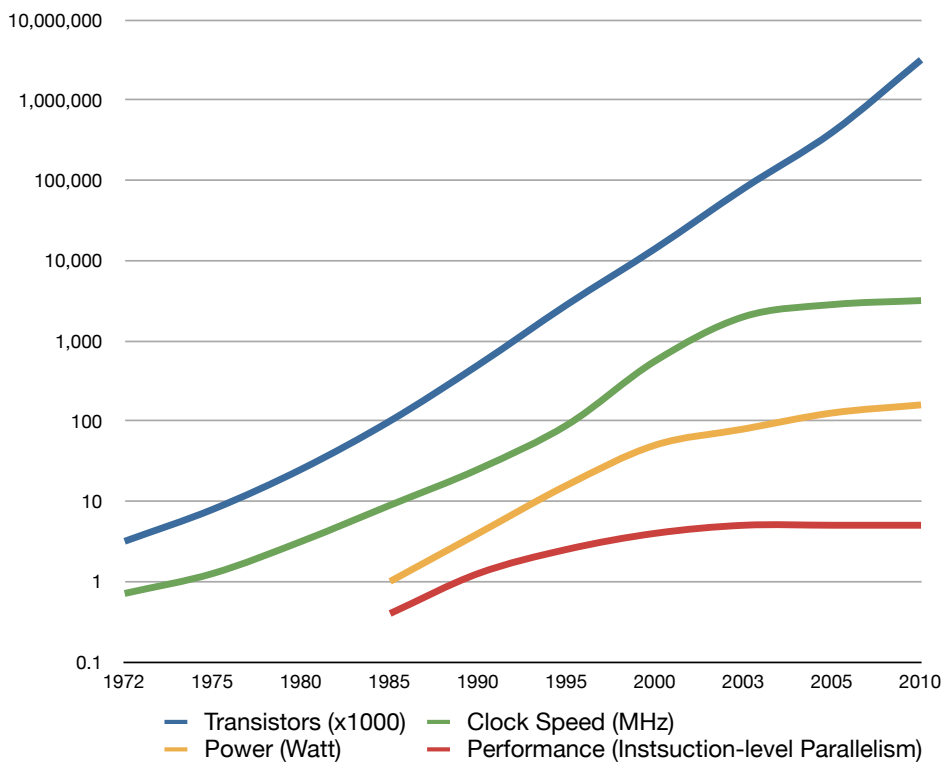

Figure 1.1: *Intel processor trends [adapted from Sutter, 2005]*

Using multiple cores, however, does not automatically improve a program's performance. Fred Brooks [1995], who managed the development of IBM's OS/360, noted that "the bearing of a child takes nine months, no matter how many women are assigned", similarly, certain programs are inherently incapable of running in parallel. Even for programs that can run in parallel, exploiting the underlying parallelism is not easy: there are issues with scheduling the program's tasks, removing the bottlenecks that slow down a parallel program, and protecting a program's critical sections and shared data.

Transactional memory [Herlihy and Moss, 1993] is a programming model that promises to make it easier to reason about accesses to shared memory by creating an abstraction similar to database transactions. A *transaction* is defined as a "sequence of actions that appears indivisible and instantaneous to an outside observer" [Larus and Rajwar, 2007]. With transactional memory, programmers do not need to worry about how to synchronize accesses to shared data, they only need to protect their shared data and critical sections using transactions, leaving the details to the underlying implementation.

There are many different proposals for supporting transactional memory. These proposals can be broadly classified into three categories: hardware transactional memory [Herlihy and Moss, 1993], software transactional memory [Shavit and Touitou, 1995], and hybrid transactional memory, which combines both hardware and software support [Lie, 2004; Moir, 2005].

Hardware transactional memory has the advantage of speed, because hardware is particularly suited for performing tasks in parallel, as well as hiding the latency of operations using speculation. On the other hand, hardware support is expensive and more risky to implement: this new hardware must be designed with consideration to how it might interact with existing processor components, and transistors that could have been used for a different purpose, such as adding more cores, must now be dedicated to transactional memory. Moreover, if the transactional model is supported only by hardware, it means that transactional memory cannot be used on existing systems today, and all its benefits would be limited to those using the new hardware.

Software transactional memory resolves most of these issues, because software can be designed to work on existing systems without requiring special hardware support — and is therefore less expensive. However, software proposals, so far, are significantly slower than their hardware counterpart, slower by at least an integer factor or even by an order of magnitude.

Hybrid transactional memory aims to be the best of both: hybrid proposals use simple hardware support for speed, when available, while relying on software for cases that would complicate hardware design, or when hardware support is not available altogether. With the hybrid approach, transactional memory implementations can run on existing systems today, and improve incrementally as their underlying hardware or software components improve.

1.2 Thesis Organization and Summary of Contributions

This thesis argues for hybrid transactional memory by drawing on work by various researchers as well as my own experiences. These experiences span three different levels of support for transactional memory: hardware, software, and putting the two together in a hybrid system. In the course of working on this thesis, I have developed novel software and hybrid transactional memory proposals. I have experimented with using a prototype of Sun's Rock processor, one of the few existing hardware transactional memory implementations. I have also investigated possible enhancements to Rock's hardware support by modeling them on a simulator. This thesis reports on these experiences.

Chapter 2 presents some of the history and background of transactional memory. It also introduces most of the terms and concepts used throughout this thesis.

Chapter 3 makes a case for hybrid transactional memory by discussing some of the related work in this area. It starts by making the case for the transactional model in general, then it discusses the benefits and drawbacks of hardware and software support, and explains how hybrid support can leverage both hardware and software to make transactional memory feasible today.

Chapters 4 to 7 present a topdown discussion of my work in the area of transactional memory, starting with software, moving to hybrid, then the use of an actual hardware implementation, and finally, by suggesting modifications to improve the hardware implementation.

Chapter 4 introduces NZSTM, the first non-blocking, zero-indirection, object-based software transactional memory. It explains the design philosophy of NZSTM, discusses the algorithm, and shows how to exploit hardware support, if present, to greatly simplify the NZSTM algorithm. This chapter presents a correctness and performance evaluation of NZSTM, and shows that NZSTM's performance is competitive with algorithms designed to be blocking.

Chapter 5 introduces NZTM, a nonblocking hybrid transactional memory that can exploit hardware support, when available, and fall back gracefully on NZSTM when hardware support is not available or sufficient. This chapter presents a performance evaluation of NZTM using a simulator and using Sun's Rock processor, which offers limited hardware transactional memory support. The evaluation shows that NZTM performs significantly better than pure software proposals, and that it is competitive with pure hardware proposals.

Chapter 6 reports on my experiences of using Sun's Rock processor to parallelize Python using hardware transactional memory. This chapter explains how to make some workloads in Python, which by default is not concurrent, scale with additional cores. It also discusses the problems encountered in the process.

Chapter 7 investigates how to mitigate the effects of false conflicts in hardware transactional memory by using data speculation, focusing on false conflicts caused by false sharing at the cache line level. It explains why false conflicts are particularly detrimental in a transactional memory environment, and presents a solution that mitigates most of their effects.

Finally, Chapter 8 concludes the thesis.

Chapter 2

The Challenges of Parallelism and the Transactional Promise

This chapter presents some of the background relevant to this thesis. It discusses some of the challenges in the move towards concurrency, both at the hardware and software level. It also introduces the concept of transactional memory, and discusses some of the problems transactional memory addresses and the promises transactional memory makes.

2.1 The Rise of Multicores

Multiprocessing, and specifically *shared memory multiprocessing*, where two or more processors are connected to a single shared memory, has existed at least since the early 1960s, with the introduction of the Burroughs mainframe computers [McCullough, Speierman, and Zurcher, 1965]. The main motivator then was the economics of replication: to boost performance, it costs less to create multiple copies of the same processor and connect them than it costs to design a single fast processor. Multiprocessing works well for applications that lend themselves to being divided into subsections and having their workload distributed over multiple processors [Hennessy and Patterson, 2006], such as many scientific applications. For other types of applications, it is not obvious how their workloads can be divided to run in parallel on multiple processors.

For general purpose computing, particularly for desktop computers, there has not been a pressing need to exploit parallelism. Because of Moore's law and the ingenuity of processor designers, processor manufacturers had the designs and the transistors they needed to produce processors that are exponentially faster every year. Therefore, the advancements in single processor computers have, until recently, inhibited research and investment in multiprocessing.

Moore's law still drives processor manufacturing today; however, processor designers are having difficulties exploiting Moore's law to make a single processor faster. This is mainly

because of issues such as the increase in power consumption, and the lack of new ideas that could use additional transistors to speed up a single processor [Sutter, 2005; Hennessy and Patterson, 2006]. Therefore, processor manufacturers use these transistors to create multiple processors on a single chip, which are known as *multicores*. IBM started this trend with the release of the dual-core POWER4 processor in 2001 [Hennessy and Patterson, 2006]. In 2005, AMD and Intel both released their first dual core processor, an important milestone, because it is these companies' processors that dominate the ubiquitous personal computing market.

Today, Intel's Core i7, a processor meant for desktop and laptop computers, can have up to six cores [Smith, 2009]. The Sony Playstation 3, a gaming console, has an eight core STI Cell processor [Chen et al., 2007]. According to Intel, this trend will continue and programmers should start considering programming for many more cores in the near future [Ghuloum, 2008].

Even though parallel programming has been studied for over 40 years, writing parallel programs is still difficult. There are many factors that complicate writing parallel programs, such as finding algorithms that can be separated into parallel tasks, and balancing the workload among the available processors. The focus of my research, however, is on the difficulties involved in communicating and managing shared data between different processors.

When a program runs in parallel on different processors, the different threads running the same program need to communicate — even if it is just to inform each other that a certain job is completed. Threads can communicate by sending messages, this is known as the *message passing* model; or they can communicate by accessing and modifying the same shared memory, this is known as the *shared memory* model [Hennessy and Patterson, 2006]. The shared memory model has the advantage of using the same interface that programs running on single processors already use; therefore, programs require fewer modifications to use this model. The alternative, message passing, requires a different programming paradigm; programs need to be rewritten, and programmers must learn this new programming paradigm.

To complicate things further, shared memory is often supported, at the hardware level, by a cache coherence protocol that operates by sending messages, whereas the message passing model is sometimes implemented using shared memory. I do not make any judgements on which model is better; but because the shared memory model is the prevalent model today [Hennessy and Patterson, 2006], it is the focus of this thesis.

Maintaining the shared memory model poses certain challenges. At the hardware level, one such challenge is cache coherence, which is responsible for ensuring that each processor's local cache memory contains a faithful representation of what is in memory across all processors. At the software level, the challenge is that accesses to shared resources, and shared memory in particular, must be protected and synchronized. This is necessary to prevent concurrent accesses to these shared resources from interfering with each other, slowing each other down, or corrupting data altogether.

The next two sections discuss some of the challenges, at the hardware and the software levels, that are particularly relevant to this thesis.

2.2 The Cache Coherence Problem

Cache memory is a small, fast, memory that reduces the average memory access time and bandwidth requirements by taking advantage of locality in memory accesses. Cache memory works by storing a local copy of locations that the processor is likely to access soon — thereby saving the processor from having to access main memory directly. Accessing main memory can be a few orders of magnitude slower than accessing the local cache, and having to go to main memory for every access increases the bandwidth requirements on the interconnect between processors and memory [Goodman, 1983].

When there are multiple processors, each with its own local cache and sharing the same main memory, the problem is how does the system guarantee that each processor's local copy is consistent with copies in other processors as well as with main memory?

The solution to this problem is the domain of *cache coherence protocols*. Coherence protocols are responsible for ensuring that all processors see a consistent view of memory in a manner transparent to the higher levels of the system. For processors' local cache data to be consistent, coherence protocols must detect and resolve potentially conflicting accesses to the same *cache line*. A cache line is a block of memory, typically ranging in size from 64 to 256 bytes.

A conflict occurs when one processor tries to modify a cache line that is being read by one or more other processors. Coherence protocols typically allow more than one processor to read the same cache line, but only one processor can access a cache line at a time if it intends to modify it.

To illustrate how cache coherence works, the following describes a cache coherence protocol by example of a typical MESI coherence protocol [Goodman, 1983; Papamarcos and Patel, 1984; Sweazey and Smith, 1986] — named after the states that a cache line can be in. This type of protocol assigns a state to each cache line, which can be one of the following.

Modified (M): The cache line data has been modified and memory must (eventually) be updated with the new value. No other processor has the current data value of this line — or more accurately, no other processor has *permission* to access the data.

Exclusive (E): The cache line data is valid and has not been modified. The processor has permission to either modify it at any time, or to share it with any other processor. No other processor has permission to access the data value of this line.

Shared (S): The cache line data is valid and has not been modified. Other processors may also have a valid (*Shared*) copy of the same cache line, but no processor has permission to modify the line.

Invalid (I): The cache line data is no longer valid, i.e., it is *stale*. The processor must request the data (and implicitly, permission) before it can read or modify it.

Not Present (NP): This is not an explicit state, but is implied by the absence of the cache line address in any state. In practice, most cache coherence protocols treat this state the same way as an invalid state.

In such a protocol, processors, at least conceptually, share a common bus to access main memory, and issue their requests by broadcasting them on the bus. The protocol maintains the correct state by snooping the bus to intercept any requests made by other processors, hence why this is known as a *snooping* protocol [Goodman, 1983; Frank and Inselberg, 1984; Thacker and Stewart, 1987]. Figure 2.1 shows an example of this protocol in action.

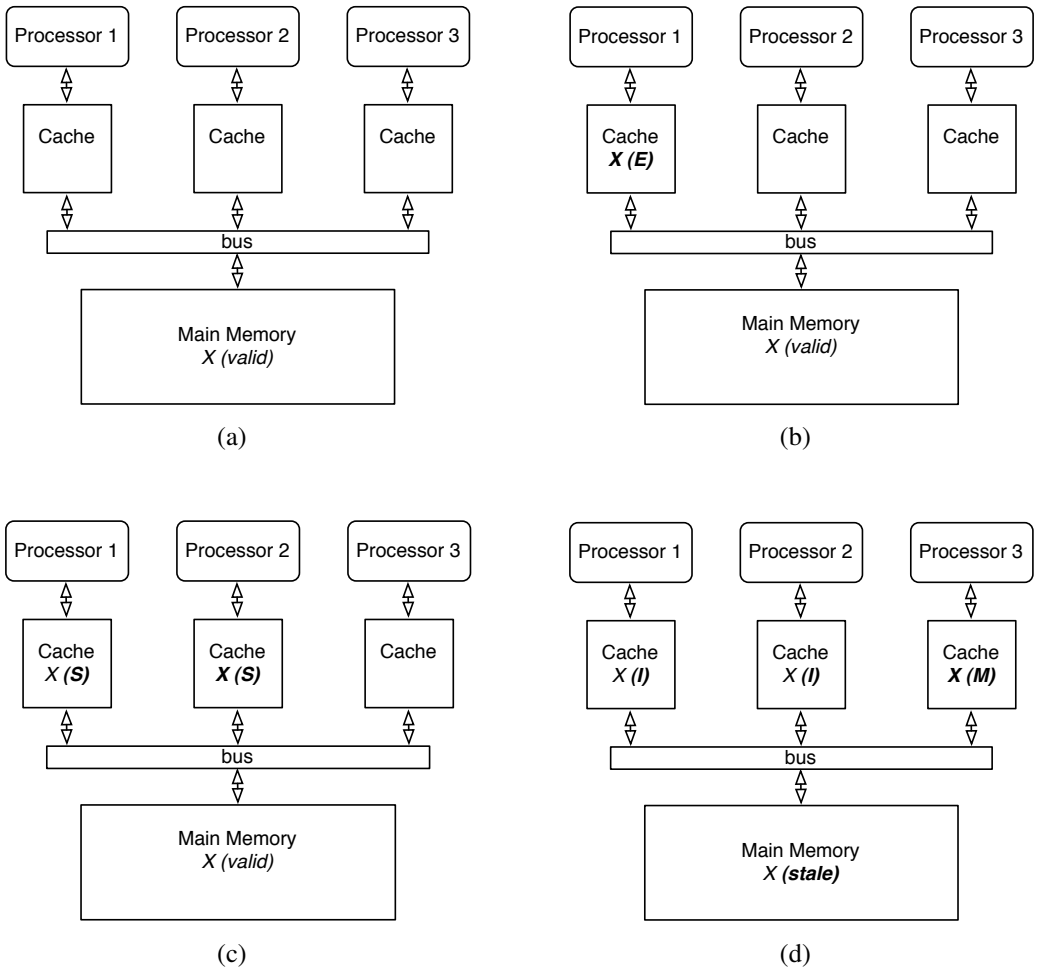

Figure 2.1: *An example of cache coherence at work*

In the example in Figure 2.1, initially (a), none of the processors have the line associated with cache line *X* in their cache; therefore, it is in the implicit *Not Present* state. Processor 1 requests to read cache line *X*, and acquires the cache line in an *Exclusive* state, because no other processor has a copy of the line (b). Next, processor 2 requests a copy for reading, and

obtains it in a *Shared* state; processor 1, seeing processor 2's request for reading, downgrades the state of its cache line to *Shared* (c). Processor 3 decides to write to X; it requests to modify the line. Processors 1 and 2 detect the conflict with processor 3, and invalidate their own copies of cache line X upon seeing processor 3's request. When processor 3 receives its exclusive copy, it applies its modifications, and therefore has the cache line in a *Modified* state (d).

Cache coherence protocols do not necessarily require a shared common bus, as in the example above. Coherence protocols can also be implemented as *directory-based* protocols [Tang, 1976; Censier and Feautrier, 1978], where the sharing state of a cache line is maintained at a common location, the *directory*. Processors request permission to access cache lines from the directory; in turn, the directory is responsible for informing processors about potential conflicts [Hennessy and Patterson, 2006].

Although cache coherence is not the only challenge at the hardware level in designing multiprocessors, it is particularly important because cache coherence is directly responsible for maintaining the shared memory abstraction. Any compromises in cache coherence would lead to leaks in the shared memory model abstraction, and could corrupt the data in memory.

Cache coherence is essentially about managing a shared resource at the hardware level, the resource being memory. With the shared memory abstraction in place, it is the operating system's and programmers' responsibility to manage the sharing of objects at the software level.

2.3 Synchronization, Mutual Exclusion, and Locks

When using the shared memory model, programmers are responsible for ensuring that, when their programs share data, concurrent accesses to the shared data are correct. Correctness depends on the program, but typically means that a shared data structure, or a group of related shared structures, must not be accessed in a *conflicting* manner by more than one thread at a time: multiple threads can simultaneously read the same data, but only one thread at a time can access the data to modify it.

The part of the program that accesses shared resources where conflicts might occur is known as a *critical section*. To ensure that the shared resources in critical sections are protected from conflicts, programmers often rely on *mutual exclusion* mechanisms [Dijkstra, 1965; Hoare, 1974; Lamport, 1974; Peterson, 1981]. Mutual exclusion mechanisms protect against conflicts by ensuring that no more than one thread can simultaneously enter the same critical section and access the same shared data.

One common method of ensuring mutual exclusion is by using *locks*. A lock is a data structure typically associated with a certain critical section or a specific shared data structure. A thread must acquire a lock before it can execute the critical section, or before it can access the shared data the lock protects. Each lock may not be owned by more than one thread at

a time; if a thread wants to acquire a lock owned by another thread, it must *block*, i.e., wait, until the lock is released.

Locks, and methods of protecting critical sections by mutual exclusion in general, have existed for over four decades. In a sense, they are tried-and-true methods. However, there are many problems associated with using locks that could compromise correctness, performance, and reliability.

One of the problems with locks, in terms of correctness, is the possibility of *deadlock*. Deadlock occurs when different threads try to acquire the same locks (or resources in general) in a different order, so each thread is waiting for another thread to release its lock before any one can proceed [Coffman, Elphick, and Shoshani, 1971]. This results in none of the threads making any progress, and often leads to the whole program — or even the whole system if the operating system is involved — coming to a halt.

Programmers can avoid deadlock by ensuring that different threads always acquire locks in the same order. This, however, adds complexity to a program, and requires programmers to be aware of instances of where locks might be used and the correct order of acquiring them — all of which makes parallel programming even more difficult. Alternatively, programmers can try to detect a deadlock, and try to recover from a deadlock once detected, which also adds complexity.

Programmers can reduce the likelihood of deadlock by using fewer locks with *coarser* granularity, where each lock is responsible for protecting bigger or more critical sections. At one extreme, programmers could use a *single global lock* that protects *all* critical sections; this would ensure that deadlock cannot occur and simplifies reasoning about the program. However, because locks are mutually exclusive, critical sections protected by the same lock cannot run in parallel, which could negate the benefits of multiprocessing.

Another problem with locks is that they are *blocking*: once a thread acquires a lock, it cannot be forced to release the lock, and there is no guarantee that the thread will release the lock within a bounded period of time. This leads to two more problems: *priority inversion* and lack of *fault tolerance*.

Priority inversion is when a low priority thread acquires a lock, and then a higher priority thread needs to acquire the same lock, but is unable to do so. If the low priority thread releases the lock as soon as it is aware of the higher priority thread, the problem is not severe. If the low priority thread is oblivious to the higher priority thread, and delays the release of the lock, the problem could manifest in the form of the system becoming unresponsive.

A blocking system is also less fault-tolerant. If a thread acquires a lock, modifies data, and crashes before it completes its modifications, then there is no reliable way of releasing the lock and ensuring that the modifications are consistent, unless the system tracks the modifications every thread makes in a critical section. In practice, what often happens is that the faulty thread corrupts critical data and causes the whole program to crash, instead of just crashing silently and dying alone.

Moreover, locks are typically implemented as *advisory* locks, where threads must cooperate to obey the locking protocol. A buggy thread can ignore locks altogether and corrupt the lock-protected data. This makes systems that use locks less tolerant to such bugs.

Because of these problems, programming with locks goes against some of the principles of software engineering. Two of the principles that software engineers rely on are *abstraction* and *composition* [Larus and Rajwar, 2007], principles that help reason about large and complex programs.

Abstraction enables software engineers to manage the complexity of their designs by reducing them to modular components; whereas composition enables them to combine these components into a bigger, more complex, application. These principles allow software engineers to reason about each component separately, without worrying about any adverse interactions these components might have when composed together. Engineers in other fields handle complexity in a similar way. For example, automobile designers do not need to know every single detail about car design, such as how tires are made or the inner workings of a particular engine model. Automobile engineers handle different components individually, abstracting away the details of every component, and then put them together, or compose them, into a car.

Programming with locks breaks abstraction because software engineers need to be aware whether a particular module they are using acquires any locks. If it does acquire a lock, they must ensure that by using it, their program still observes the locking order that prevents deadlock. Otherwise the engineer, by composing two individually correct modules together, could inadvertently cause deadlock, thus breaking composition.

In addition to the software engineering concerns, blocking makes locks unacceptable for use in certain tasks, such as interrupt handlers in an operating system [Ramadan, Rossbach, Porter, Hofmann, Bhandari, and Witchel, 2007]. Interrupt handlers *must* not be blocked by the thread they have interrupted, otherwise the whole system would deadlock. This requirement significantly complicates the design of interrupt handlers in an operating system.

Lock-free Programming

It is possible, in theory at least, to write correct parallel code without using locks or any other method of mutual exclusion — this is the area of nonblocking synchronization [Herlihy and Shavit, 2008]. Nonblocking synchronization promises that, with a minimum amount of hardware support, it is possible to write parallel algorithms that are guaranteed to complete within a bounded period of time.

The minimum hardware support for nonblocking synchronization is satisfied by a *Compare&Swap* instruction [Herlihy, 1991]. *Compare&Swap* is an atomic instruction that reads a memory location, and if the value at that location matches the value the instruction expects to find, it replaces it with a new value, indicating whether the operation was

successful. *Compare&Swap*, or similar instructions, are available on most modern parallel processors, such as x86 [int, 2010b] and SPARC [Weaver and Germond, 2000]. The code listing below demonstrates the effects of a typical implementation of a *Compare&Swap* instruction.

```
bool CompareAndSwap(int *location, int expected, int update)
{
    atomic {
        /* appears to the programmer as a single instruction */
        if (*location == expected) {
            *location = update;
            return true;
        } else {
            return false;
        }
    }
}
```

Nonblocking algorithms resolve most of the problems inherent in locks: nonblocking algorithms do not deadlock and are always guaranteed, by definition, to make forward progress. Nonblocking algorithms are fault-tolerant: if a thread executing a non-blocking algorithm crashes or hangs, only that thread is affected, and the rest of the system can continue running. Nonblocking algorithms do not break abstraction or composition; programmers can call any nonblocking algorithms within their critical sections, without having to reason about possible interactions with other parts of their programs. Moreover, nonblocking algorithms are suitable for use in interrupt handlers, because interrupt handlers in a nonblocking system will not be blocked by other threads, including threads they have interrupted.

Unfortunately, nonblocking algorithms that perform well are very difficult to write — more so than writing parallel algorithms with locks [Herlihy and Shavit, 2008]. Writing a nonblocking version of even simple blocking algorithms is often considered to be a publishable result [Larus and Rajwar, 2007].

It is this research into nonblocking algorithms that was the prime motivator for transactional memory. Herlihy and Moss [1993], who coined the term "transactional memory", originally proposed the transactional model specifically to make it easier to write efficient nonblocking algorithms.

2.4 What is Transactional Memory?

Transactional memory is a programming model that provides an abstraction on top of critical sections for managing accesses to shared data, making it easier to reason about critical sections and allowing them to be composable. Transactional memory uses the concept of a *transaction* [Eswaran, Gray, Lorie, and Traiger, 1976; Gray, 1981], borrowed from the database community [Codd, 1970]. A transaction is defined as "a sequence of actions that appears indivisible and instantaneous to an outside observer" [Larus and Rajwar, 2007].

Using transactional memory, the programmer designates certain critical section code as being *transactional*, and it is the underlying transactional memory implementation's responsibility to ensure that the code runs correctly and efficiently. The programmer, ideally, does not need to consider the ordering of transactions, the size of the transactions, or how different transactions might interact with each other. Instead, programmers merely need to ensure that their code is correct, assuming the transactional component truly appears indivisible and instantaneous.

Towards that end, transactional memory provides *atomicity*, *consistency*, and *isolation* for its component transactions — concepts also borrowed from the database community [Haerder and Reuter, 1983].

Atomicity means that a transaction is indivisible — it is all or nothing. If a transaction succeeds in running to completion, it makes its changes visible only once it *commits*. If for some reason the transaction cannot commit, it *aborts*, and it appears as if nothing has happened, i.e., as if the transaction was never executed in the first place.

Consistency ensures that a transaction must leave the system in a consistent state; it must obey all legal protocols, invariants, and constraints set by the system.

Isolation ensures that the changes made by a transaction are observable only once a transaction commits. It is isolation that is responsible for transactions appearing to be instantaneous; intermediate changes performed by the transaction cannot be observed by any thread outside of the transaction, regardless of whether the other threads are transactional or not. If other threads could view partial modifications, then they might observe an inconsistent state of the system.

Below is a simple example of how the transactional model can be used. To protect a critical section that increments a variable `x` using locks, a programmer would write the following code.

```
lock_acquire(x_lock);
x = x + 1;
lock_release(x_lock);
```

Using the transactional model, a programmer would write the following code.

```
transaction {
    x = x + 1;
}
```

In other words, with transactional memory, programmers specify *what* should be protected when running in parallel, and the underlying system deals with the *how*.

A transactional memory system should, for performance reasons, allow multiple transactions to run concurrently as long as they do not *conflict*. A conflict occurs when concurrent transactions access shared data and at least one of them modifies the data [Larus and Rajwar, 2007]. When transactions conflict, the transactional memory system arbitrates and serializes access to the conflicting location, either by stalling or aborting one or more of the conflicting

transactions. Aborting a transaction reverts any changes the transaction has made, and it appears as if the transaction never happened. Once a transaction aborts, then depending on the implementation, the system would either try it again — which is the more common approach, or inform the program that the transaction aborted, e.g., by throwing an exception.

2.5 Transactional Memory Design Space

The design space of transactional memory covers various areas such as the type of language support necessary for its programming interface, to the different ways of supporting transactions through hardware or software. This section presents some of the points in the design space that are relevant to this thesis.

At the highest level, there is the issue of how to expose the transactional memory abstraction to the programmer. Some researchers have proposed additions and extensions to existing languages to support transactional memory, such as adding the `transaction` keyword to the C++ programming language [Adl-Tabatabai, Lewis, Menon, Murphy, Saha, and Shpeisman, 2006a]. Others argue that, because programmers are familiar with the locking model, systems should retain that model at the high level and elide those locks using transactional support; a technique known as *speculative lock elision* [Rajwar and Goodman, 2001]. The work presented in this thesis is mainly concerned with the underlying implementation of a transactional system, and is independent of its programming interface.

Another aspect of transactional design is related to conflict detection between transactions. First, there is the issue of the level of *granularity* of conflict detection. This can range from being a single word, a cache line, or a whole (high-level) object. Granularity is typically governed by the type of support the system is using. Hardware proposals typically have a granularity level of a cache line, because, as will be explained later, hardware proposals often leverage the underlying cache coherence protocol for conflict detection. Software transactional memory proposals typically detect conflicts at a granularity level of a word or an object. Bigger granularity makes it easier to amortize some of a transaction's overhead, but increases the chance of *false conflicts* due to different transactions accessing disjoint parts of the same granularity unit.

Another issue with conflict detection is when to *detect* and when to try to *resolve* a conflict between transactions. Some systems *eagerly* detect conflicts as soon as two or more transactions appear to access the same data in a conflicting manner, whereas other systems *lazily* delay conflict detection until one of the transactions is ready to commit.[1] As with many design alternatives, the better design choice is not obvious. For example, a transaction using eager conflict detection could abort a conflicting transaction, only to discover later that it is itself *doomed* and cannot commit because of conflicts with other transactions. On the other

[1] Eager conflict detection is analogous to pessimistic concurrency control in database management systems, whereas lazy conflict detection is analogous to optimistic concurrency control [Larus and Rajwar, 2007].

hand, lazy conflict detection could allow a doomed transaction, which is going to eventually have to abort because of other conflicts, to continue running wasting work and resources, when eager conflict detection would already have aborted it.

Once a transactional system detects a conflict, there is the issue of conflict management and resolution. When transactions conflict with each other, there is a winner, who gains access to the data. There is also a loser, who must wait for the winner to either commit or abort. The loser waits either by stalling or aborting. Conflict management deals with the heuristics that determine a transaction's priority to decide which transaction wins when there is a conflict.

Versioning is another issue in transactional memory system design, and is concerned with maintaining information so the system can undo any modifications an aborted transaction has made. One option is for a transaction to update objects in-place, and keep a backup copy of the old data. An alternative is for a transaction to buffer its updates, applying them only when it knows that it will commit successfully.

One of the most important design issues, an issue that could directly affect other design decisions, is the type of underlying support for a transactional system. Following is a discussion of the different proposed methods to support transactional memory: hardware, software, and a hybrid combination of the two.

Hardware Transactional Memory

Hardware transactional memory proposals are classified as *best-effort* [Moir, 2005], *bounded*, and *unbounded* [Larus and Rajwar, 2007].

Best-effort hardware transactional memory [e.g., Herlihy and Moss, 1993; Rajwar and Goodman, 2001; Chaudhry, Cypher, Ekman, Karlsson, Landin, Yip, Zeffer, and Tremblay, 2009a] does not guarantee that all transactions will eventually commit successfully using hardware support alone. One of the main limitations of best-effort proposals is the size and associativity of a processor's local cache memory.

Best-effort hardware proposals rely on the underlying cache memory and cache coherence protocol. Cache memory can, with little or no adjustment, perform automatic versioning of data modified in a transaction. Write-back cache memory keeps the modified copy of a cache line in a processor's local cache, until the processor writes it back to memory or shares it with other processors; therefore, the original copy is still intact either in main memory, or in another processor's cache [Larus and Rajwar, 2007]. This simplifies aborting a transaction: to undo a transaction's modifications, a processor discards all cache lines modified during the transaction.

Cache coherence also facilitates conflict detection between transactions. Coherence protocols, by design, notify a processor of conflicting accesses to memory locations in its cache by other processors. To support conflict detection in transactional memory, each processor tracks which cache lines it has accessed as part of a transaction, and infers transactional conflicts from coherence conflicts for cache lines accessed as part of the transaction.

Best-effort proposals typically add a bit to each cache line in a processor's local cache to track the lines accessed as part of a transaction. Best-effort proposals maintain atomicity and isolation by ensuring that modifications to those cache lines are not exposed, through the coherence protocol, until the transaction commits. If a transaction aborts, the processor discards all lines modified inside the transaction — implicitly rolling back these modifications. If a transaction commits, the processor ensures that all modified lines are instantly visible to others by immediately exposing those lines through the cache coherence protocol.

Because best-effort hardware transactional memory relies on the processor's local cache to maintain its transactional state, any event that invalidates or evicts a cache line that has been accessed as part of the current transaction also results in aborting the transaction. Without additional hardware support, a transaction cannot track the state of an evicted cache line, and therefore cannot observe if other transactions access the evicted line in a conflicting manner.

Bounded hardware transactional memory proposals [e.g., Hammond, Wong, Chen, Carlstrom, Davis, Hertzberg, Prabhu, Wijaya, Kozyrakis, and Olukotun, 2004; Ananian, Asanovic, Kuszmaul, Leiserson, and Lie, 2005; Moore, Bobba, Moravan, Hill, and Wood, 2006] guarantee that certain transactions will eventually commit successfully using hardware support alone. Bounded proposals typically behave in the same way as best-effort ones, as long as cache memory resources are sufficient. When the resources are not sufficient, e.g., a transaction encounters an event that invalidates or evicts a transactional cache line, bounded proposals rely on additional hardware mechanisms and data structures that reside in main memory to continue tracking the transactional state.

Bounded proposals are not restricted by the size and associativity of cache memory; however, they cannot commit transactions that encounter events that are too complicated to handle in hardware — events that best-effort proposals cannot handle either. Whether an event is too complicated to handle in hardware depends on the particular proposal. Examples of such events include context switches, thread migration, virtual memory paging, I/O, exceptions, and interrupts [Hofmann, Porter, Rossbach, Ramadan, and Witchel, 2007]. When a transaction encounters such an event, it aborts.

Unbounded hardware transactional memory proposals [e.g., Ananian et al., 2005; Rajwar, Herlihy, and Lai, 2005] guarantee that all transactions would eventually commit, regardless of their size or the events they may encounter. Therefore, unbounded proposals must be able to handle any event without indefinitely aborting the same transaction. Having to handle all types of events adds more complexity to the hardware than in bounded proposals.

Software Transactional Memory

Software transactional memory does not use any hardware support beyond what is already available for concurrent programming. Software transactional memory proposals, therefore, must rely on software structures for versioning and conflict management.

A simple software transactional memory could be implemented using a single global lock that serializes all transactions. Such an implementation requires no additional data structures for versioning, because every transaction, once it has acquired the single lock, is guaranteed to succeed. Without additional hardware support, using a single global lock does not scale; however, such a system has low overhead, which might be acceptable if there are not many transactions.

Practical software proposals usually allow concurrency between transactions. Such proposals use more elaborate methods for conflict detection and resolution than a single global lock, and rely on a variety of versioning schemes to rollback aborted transactions.

For versioning, a software system could maintain a write set, where it stores all tentative writes of a particular transaction, committing those writes to memory once it knows the transaction is guaranteed to commit. Another option would be to maintain an undo log, or a backup copy, of the modified locations. If a transaction aborts, then the aborted transaction, or other transactions, can access the undo logs to restore the original data.

For conflict detection, software proposals often create and associate *ownership records* with each location, or group of locations, that could be accessed inside a transaction. An ownership record can refer to a single word in memory and reside in a hash table, or it can refer to a whole object and reside in its header. Therefore, before a transaction can commit, it must check the ownership record of every location it has accessed and resolve any conflicts. Moreover, a transaction must acquire a location's ownership record before it can commit its modifications of that location's data to ensure that only one transaction can modify the data at the same time.

If a transaction discovers that a location is acquired by another transaction, there is a conflict. A transaction can wait until the competing transaction has either committed or aborted, and has thereby relinquished ownership of the location. Otherwise, the transaction can abort the competing transaction and forcibly acquire ownership of the location.

The decision on whether to wait for or to abort the competing transaction is typically not hardwired into the software transactional memory algorithm, but determined by consulting a *contention manager* module [Herlihy, Luchangco, Moir, and Scherer, 2003b], which may apply different conflict resolution policies. The contention manager is often implemented as an *out-of-band* module [Herlihy et al., 2003b], where its policy and logic are independent from the underlying algorithm. Therefore, an out-of-band contention manager can implement various policies and conflict resolution schemes without affecting the correctness of the software transactional memory.

Transactions that only read a location do not necessarily need to acquire ownership of that location. Using *invisible reads*, a transaction can detect conflicts by *validating*, before a transaction commits, that a location has not been modified since the transaction has read it.

A transaction can validate by checking that the location's ownership record has not changed since it last read it, because for another transaction to modify the data it must also have modified the location's ownership record. A transaction can also validate by comparing the current data value of the location with the value it has used previously. If the value has not changed, then as far as the transaction is concerned, no other transactions have modified this location in the interim, and no conflicts have occurred. This is known as *value-based* conflict detection [Ding, Shen, Kelsey, Tice, Huang, and Zhang, 2007; Olszewski, Cutler, and Steffan, 2007].

As an alternative to invisible reads, transactions could use read ownership records, or *visible reads*, for conflict detection. A transaction that reads a location adds itself to a readers list associated with that location. Any transaction that wants to modify the location must check this readers list and resolve all conflicts before it can commit.

Software transactional memory algorithms can be either blocking or nonblocking. Nonblocking software transactional memory, like nonblocking algorithms in general, are more complex and challenging to write. Some have even argued that nonblocking software transactional memory algorithms are inherently slow and should not even be considered [Ennals, 2006]. Chapter 4 presents my response to this argument.

Because software transactional memory proposals must perform these additional tasks in software, compared with hardware transactional memory, the overhead of software transactional memory is quite high, as explained in the next chapter. Software proposals can be an order of magnitude slower than hardware for some workloads. Software transactional memory can be designed with low overhead; the simple system based on a single global lock described earlier is an example of that. However, such systems do not scale without additional hardware support.

Hybrid Transactional Memory

Hybrid transactional memory uses both hardware and software for its underlying implementation. The goal of hybrid transactional memory is the flexibility of software and the performance of hardware, while requiring hardware support that is considered to be realistic and feasible to implement.

Hybrid transactional memory design can begin with a software transactional memory, and use hardware support to optimize its bottlenecks. Hybrid design could also begin with a hardware transactional memory, and use software to handle the corner cases and complicated aspects that hardware cannot feasibly handle. A special case of this approach, highlighted because of its relevance to this thesis, is when a hybrid system attempts a transaction using

best-effort hardware transactional memory, and falls back on a software transaction when the hardware transaction (repeatedly) aborts [Lie, 2004; Moir, 2005].

For performance, it is important that a hybrid system run as many transactions as possible using the fast hardware path, and avoid using the slower software path. For correctness, a hybrid system must detect and correctly handle conflicts between transactions that could be running using hardware or software support.

In a hybrid system that uses best-effort hardware and falls back on software, hardware transactions automatically detect conflicts with any software accesses that follow hardware accesses. This is a direct result of hardware transactions relying on the cache coherence protocol for conflict detection. However, software transactions do not automatically detect conflicts with hardware transactions when a hardware transaction accesses an object *after* the software transaction does, because hardware transactions, by design, are a low level system abstraction whose effects are isolated from software until the hardware transaction commits. Therefore, the burden is on hardware transactions to ensure that the data they access is not being accessed by a concurrent software transaction in a conflicting way. This checking could be instrumented as part of the hardware transaction's code, but adds overhead to hardware transactions in this type of hybrid system.

2.6 Evaluating Transactional Memory

One of the challenges in transactional memory pertains not to the design of a transactional system, but to evaluating how well it performs. The problem with evaluating transactional systems is that the transactional model is new, so there are few existing applications that use this model.

Many of the first transactional systems evaluated their proposals using a set of synthetic and microbenchmarks created specifically for the purpose of testing transactional memory proposals [Herlihy et al., 2003b; Marathe, Spear, Heriot, Acharya, Eisenstat, Scherer, and Scott, 2006]. These benchmarks, as with any synthetic benchmark, do not represent realistic workloads.

Later transactional proposals used some of the older and well-established benchmarks, such as the SPLASH-2 benchmark suit [Woo, Ohara, Torrie, Singh, and Gupta, 1995], for their evaluation, by converting the benchmarks' critical sections to use transactions instead of locks [Moore et al., 2006]. The problem with this approach is that critical sections written for the mutually-exclusive locking model are optimized by experts to be small and have as little shared data as possible. This type of workload is not necessarily representative of how transactions might be used by the average programmer.

A few research groups proposed new benchmarks to fill the gap. One of the most comprehensive benchmark suite designed for evaluating transactional memory is the STAMP benchmark suite [Minh, Chung, Kozyrakis, and Olukotun, 2008]. The STAMP benchmarks

use different types of algorithms and exhibit different transactional characteristics in terms of transaction length, conflict rates, and size of the read and write sets. Because of STAMP's broad coverage, it is the most widely used benchmark suit for evaluating transactional systems. STAMP, however, is written by experts on parallel programming. Although some of its workloads represent realistic applications, the programming style is not necessarily representative of how the average programmer might use the transactional model.

To cover a wider variety of workloads, the evaluations presented in this thesis use a combination of synthetic benchmarks I have developed, microbenchmarks, SPLASH-2 benchmarks, and STAMP benchmarks. Benchmarks are seldom comprehensive or conclusive, but using a variety of benchmarks helps develop better intuition of the performance of the evaluated systems.

2.7 The Current State of Transactional Memory

This section explores the current state of transactional memory and the progress it has made towards becoming a mainstream model. It looks at transactional memory in production systems and in systems that are being tested for production use.

For hardware transactional memory, as far as I am aware, the only two attempts of supporting hardware transactional memory in a commercial processor are by Sun Microsystems and Azul Systems.

In 2007, Sun Microsystems announced a new processor, codenamed *Rock*, which as of yet has not been released to the public [Chaudhry et al., 2009a].[2] Rock's design was a departure from Sun's Niagara family of processor. The Niagara family of processors comprises a group of simple cores designed for high throughput applications such as web servers. Rock was designed as a high-performing floating-point intensive processor. Each Rock processor has 16 cores, each core capable of running two threads. One of the new features in Rock is that it performs aggressive speculation, which Sun leveraged to support best-effort hardware transactional memory. I used a Rock prototype for part of the evaluation presented in this thesis. Rock's best-effort hardware, and the particulars of the prototype used, are discussed in the next section.

Azul Systems is a privately-held company that manufactures specialized *computer appliances* that run Java-based applications [Click, 2009c]. Their Vega 3 series computer has 864 cores, and is designed for high throughput and scalability. Azul's processor implements a best-effort hardware transactional memory, which they use to elide locks in the Java Virtual Machine [Click, 2009a,b]. Because Azul's hardware is best-effort, when a hardware transaction aborts a certain number of times, the system falls back on acquiring the Java lock it had attempted to elide.

[2] It is likely that Rock has been cancelled [Vance, 2009], although Sun has neither confirmed nor denied this yet.

Azul's hardware transactional memory uses the first level (L1) cache as its transactional cache, adding two bits per cache line to track the lines that have been read or modified as part of a transaction. Therefore, hardware transactions are limited by the size and associativity of the L1 cache. A transaction aborts if a cache line it has accessed speculatively is invalidated or evicted from the cache, either because of a conflict with another processor, or if the L1 cache is not large enough to hold all the transactional data.

Even though hardware transactional memory was first proposed in 1993 [Herlihy and Moss, 1993], only two actual implementations exist today, Sun's and Azul's. Neither are on a commercially-available general purpose computer, and both are best-effort. This shows that implementing hardware support for transaction — and unbounded support in particular — is difficult, at least in the sense that processor manufacturers believe that the cost/benefit tradeoff does not yet justify any hardware implementation that goes beyond best-effort.

When it comes to software transactional memory, the only production-quality compiler that supports it is the Glasgow Haskell Compiler, the main compiler for the Haskell programming language [Perfumo, Sönmez, Stipic, Unsal, Cristal, Harris, and Valero, 2008; ghc, 2010]. Haskell is a pure functional programming language, which by default does not allow mutable variables. Therefore, it can more easily accommodate transactional memory: the language is defined so mutable variables can be accessed only inside transactions [Harris, Marlow, Peyton-Jones, and Herlihy, 2005].

As for more mainstream programming languages, there are still no production-quality compilers or runtime environments, that I am aware of, with transactional memory support. However, Microsoft has released STM.NET, an experimental version of the .NET framework that supports transactional memory [stm, 2009]. Intel has also an experimental C++ compiler with transactional memory support [int, 2010a]. Sun has recently developed a C++ compiler that supports software and hybrid transactional memory, and takes advantage of Rock's best-effort hardware support [Lev, Luchangco, Marathe, and Moir, 2009]. The GNU Compiler Collection (GCC) project [Stallman, 2004] is working on incorporating software transactional memory support into the collection [gcc, 2010], but has not released it yet.

2.8 The Rock Processor and the ATMTP Simulator

Sun Microsystems developed a new processor, codenamed Rock, which supports best-effort hardware transactional memory [Chaudhry et al., 2009a]. Rock is not commercially available; however, Sun granted me access to a prototype. Sun has also released a simulator, the *Adaptive Transactional Memory Test Platform* (ATMTP) [Moir, Moore, and Nussbaum, 2008]. ATMTP is binary-compatible with Rock, and is a first-order approximation of the behavior of transactions on Rock. Both Rock and ATMTP are used for the evaluation in later chapters. This section presents a background of both, focusing on the aspects relevant to transactional memory.

Rock is a 64-bit multicore SPARC microprocessor. The prototype I had access to ran a single Rock processor, with 16 cores, each core configured to run at 1.5 GHz, and to support only one thread (i.e., *SSE* mode).[3] Rock's 16 cores are arranged in clusters of four: each cluster shares a 32 KiB L1 instruction cache and a 64 entry instruction *translation lookaside buffer* (TLB), and each two cores within a cluster share a 32 KiB L1 data cache and a 32 entry data TLB. Both the L1 instruction and the L1 data caches are 4-way set-associative. All cores share a 2 MiB unified L2 cache, which consists of four 8-way set-associative banks. Rock uses a directory-based MESI cache coherence protocol. Figure 2.2 shows the organization of a single Rock chip.

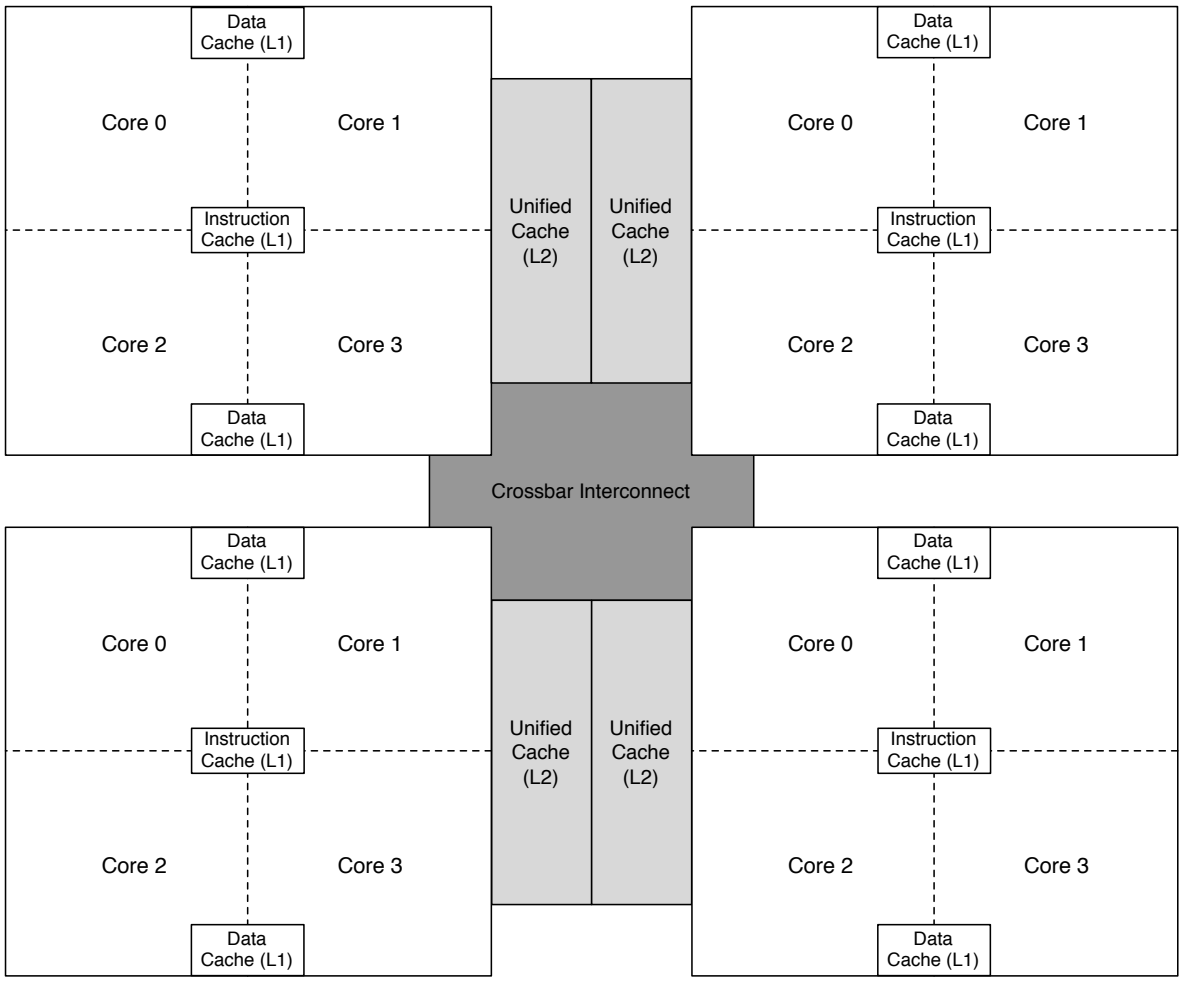

Figure 2.2: *The Rock processor organization [based on the description in Chaudhry et al., 2009a]*

In addition to best-effort hardware transactional memory, Rock supports other forms of aggressive speculation, such as the use of a hardware scout when a thread is stalled because of a cache miss [Chaudhry, Cypher, Ekman, Karlsson, Landin, Yip, Zeffer, and Tremblay,

[3] It is the same Rock prototype that Dice, Lev, Moir, and Nussbaum [2009a] refer to as "R2" in their work.

2009b]. A hardware scout is a speculative hardware thread that runs ahead of the main thread it is supporting, prefetching cache lines and speculatively retiring instructions out-of-order.

To support speculation in general, Rock tracks speculatively accessed cache lines at the L1 data cache. Rock uses a 32 entry write buffer, per core, to hold speculative stores, instead of storing them in the L1 cache. Rock introduces two new instructions specifically for transactional memory support: `chkpt` and `commit`. The `chkpt` instruction begins a transaction, and specifies a *fail address*, which it branches to if the hardware transaction aborts. The `commit` instruction attempts to commit the current hardware transaction. Rock does not have an instruction explicitly designed to abort a hardware transaction. However, there are many instructions that Rock does not support inside a transaction; such instructions can be used to explicitly abort a transaction.

When a transaction aborts, Rock provides feedback on the cause of the abort using the *Checkpoint Status* (CPS) register. The contents of the CPS register is typically read by the logic at the fail address of the `chkpt` instruction. Depending on the cause of the abort, the program could either attempt the transaction again in hardware, or use a software fallback mechanism, such as a software transaction or a lock.

Rock's hardware support is best-effort, and many events cause transactions in Rock to abort. Some of the events that could abort a Rock transaction are the following [Dice, Lev, Moir, Nussbaum, and Olszewski, 2009b].

- Running out of resources in the L1 data cache, caused by too many reads or too many writes.
- Running out of space in the write buffer, caused by too many writes.
- Cache line invalidation, caused by a coherence conflict with another thread. Whenever there is a conflict over a cache line, the requester of the line always wins, which invalidates all other copies of the line, in turn aborting any active transactions that have accessed that line.
- Unsupported instructions. Some instructions, such as the divide instruction, always abort transactions.
- Function calls inside a transaction. This is a subset of *unsupported instructions*, because Rock aborts transactions that execute a `save` instruction and a subsequent `restore` instruction. The `save`/`restore` combination is a SPARC assembly idiom that allocates and deallocates register sets, and is typically used in function calls [Weaver and Germond, 2000].
- Mis-speculation. As mentioned, Rock uses aggressive speculation to improve performance. If Rock speculates inside a transaction, and the speculations turns out to have taken the wrong path, the transaction aborts.

- Exceptions, TLB misses, page faults, and interrupts.

Because Rock does not guarantee that any transaction will eventually succeed, the transaction fail address should always provide a software mechanism for the transaction to fall back on, even for simple transactions.

To test some design alternatives that model a Rock-like processor, I used and extended Sun's ATMTP simulator [Moir et al., 2008]. ATMTP is part of the University of Wisconsin's *General Execution-driven Multiprocessor Simulator* (GEMS) [Martin, Sorin, Beckmann, Marty, Xu, Alameldeen, Moore, Hill, and Wood, 2005], which in turn is a *Simics* extension module [Magnusson, Christensson, Eskilson, Forsgren, Hallberg, Hogberg, Larsson, Moestedt, and Werner, 2002].

Simics is a full-system simulator capable of simulating a complete SPARC system running Solaris with complete binary compatibility. Simics, however, does not perform any low-level hardware or memory system modeling, and, by default, assumes that every instruction, including memory instructions, takes one clock cycle to complete. Simics provides hooks that can adjust the timing and control flow of instructions.

The Wisconsin GEMS toolset uses the hooks Simics provides to model different memory systems, cache coherence protocols, and transactional memory proposals. GEMS relies on Simics to perform the functional aspect of the simulation, and uses its own timing models. Therefore, Simics determines *what* the result of executing a certain instruction is, whereas GEMS determines *when* that instruction should execute, and how long it should take. GEMS comprises two main modules, *Ruby* and *Opal* (Figure 2.3).

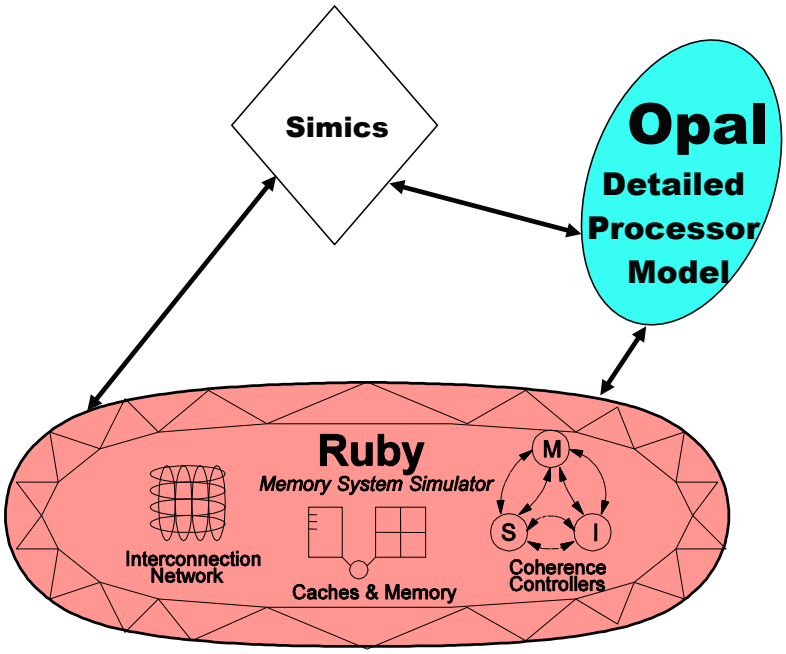

Figure 2.3: *Overview of the architecture of Wisconsin GEMS [reproduced from Martin et al., 2005, with modifications]*

Ruby models a multiprocessor's memory system in detail, including caches, cache controllers, interconnects, and memory banks. Ruby is written in C++, and communicates with Simics using hooks that Simics triggers when memory operations are invoked. Simics also provides hooks for altering the contents of memory, and the outcome of load and store instructions. Ruby uses these hooks to model transactional memory proposals.

Opal models a dynamically scheduled superscalar multiprocessor in detail. Opal, however, is not supported by the ATMTP simulator, because of the complexity involved in modifying its detailed architectural model to support transactional memory. With each additional module in Simics, the total simulation time increases by an order of magnitude [Marty, Beckmann, Yen, Alameldeen, Xu, and Moore, 2005]. The sets of experiments I run typically take over a week to complete, running on a dedicated cluster; therefore, increasing their running time by an order of magnitude is not feasible. For these reasons, Opal is not used.

Not using Opal, and not modeling the processor in detail, means that the simulation results may be less accurate. Most of the evaluation in this thesis is concerned with the memory system, which Ruby models in detail. Although the simulation results might be less accurate, I believe they are suited for evaluating transactional memory proposals, so do others who have used the same simulation environment [Moore et al., 2006; Yen, Bobba, Marty, Moore, Volos, Hill, Swift, and Wood, 2007; Bobba, Goyal, Hill, Swift, and Wood, 2008; Moir et al., 2008; Shriraman, Dwarkadas, and Scott, 2008]. Comparing the experiments on the simulator with the ones on Rock corroborates this (Sections 4.3 and 5.2.4).

ATMTP is itself a Ruby component. The system ATMTP models is quite different from Rock. Because ATMTP uses only Ruby and not Opal, it models a single-issue, in-order, processor. ATMTP does not accurately model instruction-level execution; each instruction, apart from those that access memory, takes one clock cycle to complete. Although ATMTP is binary-compatible with Rock, some of its transactions might commit successfully when they would have otherwise aborted on Rock, and vice-versa. Sun has not released any detailed documentation describing the details of these scenarios.

ATMTP's processor organization differs significantly from Rock's organization. ATMTP models a multicore chip, where each core has its own L1 instruction and data caches, and all cores share a unified L2 cache, which consists of sixteen banks. The size and associativity of the caches are adjustable parameters. ATMTP models a directory-based MESI cache coherence protocol; however, it is not known how similar the details of ATMTP's coherence protocol are to Rock's. Figure 2.4 shows the organization of the system ATMTP models.

ATMTP models Rock's transactional instructions, `chkpt` and `commit`. ATMTP can also be configured to allow some of Rock's unsupported instructions inside transactions, and by extension, to allow function calls in transactions. ATMTP can apply different policies for handling conflicts between hardware transactions, such as Rock's requester-wins [Moir et al., 2008], and priority-based timestamp policies [Bobba, Moore, Volos, Yen, Hill, Swift, and

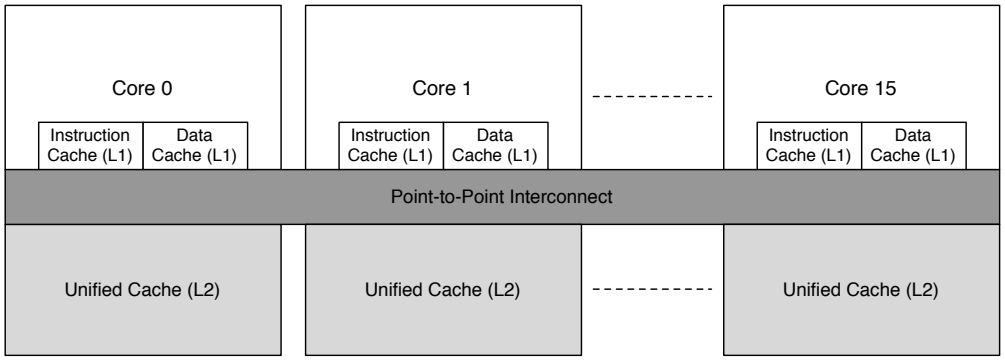

Figure 2.4: *ATMTP processor organization model*

Wood, 2007]. However, exceptions and interrupts are still difficult to handle in ATMTP and cause its transactions to abort.

2.9 Other Challenges in Parallel Programming

Even though the main focus of this thesis is on transactional memory, I emphasize that transactional memory is *not* a panacea. Transactional memory does not solve all problems related to parallel programming and does not promise to do so. Transactional memory provides a level of abstraction that makes it easier to reason about accessing shared data between different threads. This abstraction does not make an incorrect algorithm, or a buggy program, correct. It does, however, reduce the chances of concurrency bugs by making it easier to reason about the program.

Transactional memory makes no promises regarding performance either. Although many transactional memory implementations try to achieve good performance, it is possible to write efficient code by hand using traditional methods. As with many other abstractions, such as using a high level language instead of a low level language, performance is a cost/benefit tradeoff between the time programmers spend writing and debugging their programs and the time it takes to run those programs.

Below is an overview of some of the challenges that transactional memory, as a model, does not address.

Amdahl's Law

Gene Amdahl [1967] formulated what is now known as *Amdahl's law*, which implies that there was a fundamental limit to the speedup possible from parallelism. Assuming a program has a serial component and a parallel component, and assuming that all the available processors can be used perfectly without any overhead, the best speedup possible from using additional processors is represented by the following equation.

$$speedup = (s + p)/(s + p/N)$$

where s is a program's sequential component, p is its parallel component, and N is the total number of processors running the program.

For example, assume that a program starts by loading data from a drive and initializing the data, an operation which takes 10 seconds. The program then performs computations that can run concurrently, which take 90 seconds to run on a single processor. Amdahl's law shows that, depending on the workload, there is less gain from adding more processors. With infinite processors, the program in this example cannot run in less than 10 seconds — the time it takes to run the serial component. Therefore, the best possible speedup is a factor of 10 in this example.

The implication of Amdahl's law is that programmers cannot expect to take any program and automatically get improvement with additional cores, even if the program is written for a parallel system. To benefit from the additional cores, programmers must increase the parallel portion of the program, either by using workloads with a bigger parallel component, or by using a different algorithm that has a smaller sequential component. Otherwise, even a small serial component becomes a bottleneck as the number of processors increases.

Managing and Coordinating Multiple Threads

Another difficulty that programmers face is how to manage all the threads in an application.

The first problem is deciding on how many threads a program should spawn to run its workload. Spawning too many threads, more than the underlying hardware can handle, could result in contention among the threads, and waste resources by having the hardware swap between the different threads. Even if there is no contention between the threads, the act of creating a new thread can incur significant overhead. For example, in Apple's OS X, each thread has a space overhead of 512 KiB and takes several hundred instructions to create [Siracusa, 2009].

Programmers might be tempted to create as many threads as the hardware allows. This might be a good solution if that program is the only one running on the system; but if multiple programs are running, this would create contention over the available resources between the different programs. This becomes more complicated if the workload of individual programs is variable. This problem is tangential to transactional memory, and is the domain of solutions such as Cilk [Blumofe, Joerg, Kuszmaul, Leiserson, Randall, and Zhou, 1995], OpenMP [ope, 2008], and Grand Central Dispatch [gcd, 2010], all of which abstract away the process of creating and managing multiple threads.

Leaky Abstractions and the False Sharing Problem

The notion of abstraction is one of the cornerstones of software engineering, and is particularly important on systems as complex as today's processors. However, abstractions are not

always perfect and often *leak* [Spolsky, 2004]. That is, an abstraction can break, or become partially exposed, when it is used in certain workloads that the abstraction designers might not have anticipated.

For example, virtual memory abstracts away the limitations of physical memory, and, among other things, provides the illusion of infinite memory. If users were to load programs that use more than the physical memory available on a computer, the operating system resorts to paging to the hard drive, users notice that the hard drive is spinning and thrashing, and the computer slows down significantly. The system would technically still work, but it might be too slow to do anything useful with.

Another example of leaky abstractions is the problem of *false sharing*, at the cache line level, in cache coherent multiprocessors [Goodman and Woest, 1988]. Cache memory allows each processor to keep a local copy of memory locations it is accessing, allowing it to read and modify its own cached copy for better performance, while coherence protocols guarantee that, despite the memory system potentially having multiple copies of the same memory location, each processor has a consistent view of memory. As long as processors access different memory locations, there are no conflicts, and coherence should ideally allow these accesses to proceed without impacting performance.

In practice, processors do not maintain coherence at the granularity level of a single address byte, or even a single word. For simplicity, processors maintain coherence at the granularity level of a cache line, which typically ranges from 64 bytes to 256 bytes. Therefore, if two processors access different parts of the same cache line, coherence protocols consider that as a coherence conflict, and serialize accesses to that cache line.

For example, consider the code segment below, where two threads concurrently increment two logically distinct values.

```
/*
 * Two global variables
 */
int x = 0;
int y = 0;

/*
 * called by thread 1
 */
while (x < 1000000) x++;

/*
 * called by thread 2 running in parallel with thread 1
 */
while (y < 1000000) y++;
```

If thread 1 and thread 2 run concurrently, then because the threads are accessing different data, programmers would expect this program to run twice as fast on a dual core system as it would on a system using a single processor.

However, a typical compiler might allocate the variables `x` and `y` consecutively, to addresses that fall on the same cache line. Every time thread 1 increments `x`, it exclusively acquires the line containing `x`. Because that line is shared with `y`, by acquiring the line con-

taining `x` exclusively, thread 1 also acquires `y` exclusively even though it is not interested in it. The performance of running a program that suffers from this type of false sharing could be equivalent to running the threads sequentially rather than in parallel. It could even result in worse performance than sequential execution because of the thrashing of the cache lines between the two processors.

For this particular example, one possible solution is to pad the variables to ensure they fall on different cache lines. This could be done by adding a dummy variable, as big as the cache line size, between `x` and `y`, as the code segment below demonstrates.

```
#define CACHE_LINE_SIZE 64

int x = 0;
char padding[CACHE_LINE_SIZE];
int y = 0;
```

Padding increases the memory footprint of the program and reduces locality. What if, in a different part of the program, the same thread needs to access both `x` and `y`? Instead of having to request only one cache line, the processor now needs to request two cache lines.

Padding is also difficult when programming in a high level environment, such as the Java Virtual Machine or the Microsoft .NET Framework. High-level environments provide yet another level of abstraction and make it difficult for programmers to specify the data layout at the lower levels. The Sun Java Virtual Machine, for example, does not lay out the variables of a class in the order they are declared, as C typically does for its structs (structured records). Instead, it orders them by type, and adds headers to each object [Neto, 2008]. In this case, padding might have the effect of separating an object's data from its header, impacting performance even more.

In this example, identifying the variables that suffer from false sharing is simple; however, this can be significantly more challenging in more complex programs. Furthermore, the programmer needs to know an architectural detail, namely the cache line size, and possess a basic understanding of cache coherence to be able to resolve this problem. All of these architectural details should ideally be abstracted away.

Chapter 3

A Case for Hybrid Transactional Memory

This chapter makes a case for hybrid transactional memory. It argues that, if transactional memory as a programming model is going to be successful, and go beyond being a niche tool, it would be with the help of hybrid transactional memory. This argument is based on the observations of many experts in the field, and on my own experiences, which the coming chapters elaborate on.

The argument in this chapter can be summarized as follows.

With the rise of multicores, parallel programming is becoming a necessity if programmers want to take advantage of the additional processing power; however, parallel programming is difficult. To make parallel programming feasible, programmers need a new programming model, or abstraction, that helps them reason about parallel programs. In my opinion, transactional memory is one of the most promising models that could make parallel programming manageable.

Best-effort hardware transactional memory has been implemented on at least two processors; however, it is not capable of handling all transactions, and therefore cannot provide a complete solution. Bounded transactional memory is more complex than best-effort, yet is also not capable of handling all transactions, and therefore does not provide a complete solution either. Unbounded hardware transactional memory is fast and promises to handle all transactions in hardware; however, it is impractical to implement.

Software transactional memory can be — and has been — implemented to run on current systems at no additional hardware cost; however, software transactional memory adds too much overhead for it to become a viable alternative to locks. Therefore, hybrid transactional memory is the best — and currently the only — practical approach that addresses the shortcomings of both hardware and software proposals.

This chapter starts by making a case for transactional memory as a model in general, and then focuses on hybrid transactional memory in particular.

3.1 A Case for Transactional Memory

To make a case for hybrid transactional memory, it should be established that transactional memory itself has a compelling case. Although a detailed argument for transactional memory is outside the scope of this thesis, an argument for hybrid transactional memory would be incomplete without a brief discussion.

First, I reemphasize that transactional memory is *not* a panacea, it will not solve all problems related to parallel programming, and it does not promise to do so. The main promise of transactional memory is that it provides a level of abstraction over critical sections that helps make code that uses transactions modular and composable, which is difficult to achieve with a lock-based model.

By promising abstraction, transactional memory implicitly promises to make it easier to write code that performs well. Performance, in and of itself, is not the main goal of transactional memory. Transactional memory promises to make the programmer's time versus the program's performance tradeoff in favor of the programmer, as many other abstractions do [Spolsky, 2004]. For the same amount of time a programmer spends writing, optimizing, and maintaining code, transactional code would be faster than traditional lock-based code.

The question becomes, is there evidence that the transactional model could deliver on its promise to make it easier to write and reason about parallel programs?

Below are some of the arguments and evidence in the literature that make the case for the transactional model. Some arguments show that transactional memory simplifies writing concurrent algorithms. Other arguments show that it can improve the performance of complex real-world applications without adding additional complexity to the code. Furthermore, there are empirical user studies of non-expert programmers using transactional memory, which demonstrate how non-expert programmers might benefit from using the transactional model in practice.

3.1.1 Simplifying Concurrent Algorithms

The first argument is that transactional memory simplifies reasoning about concurrent algorithms. An intuitive example of this argument, observed by Professor Maurice Herlihy, is writing a concurrent first-in first-out (FIFO) queue [Larus and Rajwar, 2007]. A sequential FIFO queue is a simple data structure typically taught in introductory computer science courses. An example of writing such a queue, just with the enqueue operation, in (simplified) C would be as follows.

```
struct Node {
   int value;
   Node *next;
};
```

```
struct Queue {
    Node *head;
    Node *tail;
};


void enqueue(Queue *queue, int value)
{
    Node *newNode;

    newNode = malloc(sizeof(Node));
    newNode->value = value;
    newNode->next = NULL;
    queue->tail->next = newNode;
    queue->tail = newNode;
}
```

However, writing a correct and concurrent queue that performs well is a difficult task. As a testimony of its difficulty, Michael and Scott published a solution to the concurrent queue problem in the 1996 Symposium on Principles of Distributed Computing, a leading conference in the field of concurrent algorithms.

By contrast, with transactional memory, the solution of this problem is as simple as the code that follows.

```
void enqueue(Queue *queue, int value)
{
   Node *newNode;

   transaction {
       newNode = malloc(sizeof(Node));
       newNode->value = value;
       newNode->next = NULL;
       queue->tail->next = newNode;
       queue->tail = newNode;
   }
}
```

On a more practical level, Dice, Lev, Marathe, Moir, Olszewski, and Nussbaum [2010] explored the use of Rock's best-effort hardware transactional memory to simplify writing concurrent algorithms. They experimented with a range of complex concurrent algorithms, such as double-ended queues, work-stealing queues, and memory allocators — all more complex than a FIFO queue. They reported that, not only was it easier to make these algorithms concurrent using transactions, but also that these algorithms' throughput was almost as good, if not better than other concurrent versions of the same algorithms written by experts.

3.1.2 Improving the Performance of Real-World Applications

Another argument for transactional memory is based on the experiences of experts applying transactional memory to real-world applications, with the result being simpler code, improved performance, and better reliability.

One such application is the Linux kernel, the most popular open source operating system kernel, which has been the focus of two studies. Operating system kernels are attractive targets for transactional memory because the whole computer system relies on them; therefore,

any improvement in the kernel could improve the user experience as a whole. Kernels are also complex programs that must handle multiprocessing before any user application running on top of the kernel could.

Kernel concurrency is complicated because of the nature of interrupts and their handlers. An interrupt handler could be invoked in the middle of executing a variety of different applications. This complicates the handlers' design and makes it more difficult to use locks because an interrupt handler cannot share any locks with the application its interrupting, otherwise it might deadlock with that application [Ramadan et al., 2007].

Hofmann, Rossbach, and Witchel [2009a; 2009b] evaluated using transactions in Linux 2.4 and compared the result with Linux 2.6. Linux 2.4, released early 2001, uses about 8,000 locks for its synchronization needs. It also relies heavily on the *Big Kernel Lock*, a coarse-grained lock it uses to protect different, possibly unrelated, kernel operations. Linux 2.6, released late 2003, was better optimized for multiprocessing: it uses at least 640,000 locks and restricts the use of the Big Kernel Lock more so than Linux 2.4. This finer locking granularity results in a three-fold speedup for Linux 2.6 in some cases. This improvement comes at the cost of having to reason about two-orders of magnitude more locks.

Hofmann et al. modeled a best-effort hardware transactional memory to elide locks in Linux 2.4, falling back on locks when transactions repeatedly abort. They then compared the results with an unmodified Linux 2.6. For the benchmarks they used, transactional Linux's performance closed half the performance gap between Linux 2.4 and 2.6, and in some cases, it closed almost 70% of the gap — without adding much programming complexity to Linux 2.4.

Another example of using transactional memory in Linux is the work of Ramadan, Rossbach, Porter, Hofmann, Bhandari, and Witchel [2007] on using transactions in Linux 2.6. As mentioned, Linux 2.6 is already optimized for multiprocessing, and therefore has less potential for performance improvement — at least compared with Linux 2.4.

Instead of using best-effort hardware transactional memory, Ramadan et al. modeled *unbounded* hardware transactional memory, converting lock-protected critical sections in Linux to transactions, without requiring a fallback mechanism. In addition to the other advantages of using transactions instead of locks, such as better fault tolerance, deadlock prevention, and no priority inversion, in one benchmark, they obtained an improvement in execution time of 80% compared with the regular Linux 2.6.

Other applications that could also benefit from transactional memory are runtime environments and byte-code interpreters. In particular, researchers have investigated the use of transactional memory to improve concurrency in the Python byte-code interpreter.

Python is a high level programming language introduced in 1991 [van Rossum, 2009a]. There are many different implementations of Python, but the de facto standard is CPython, a byte-code interpreter written in C. CPython was not initially concerned with performance on multiprocessors, a reasonable approach given the systems Python ran on and the workloads

it was expected to handle at the time. To protect its critical sections when running multiple threads, CPython uses a single global lock, known as the *Global Interpreter Lock* (GIL). This lock serializes *all* accesses to CPython's internal structures; therefore, it cannot fully take advantage of multiple processors except when performing certain tasks not protected by the GIL, such as I/O and running external modules.

CPython developers tried to decompose the coarse-grained GIL into multiple fine-grained locks; however, the resulting overhead in the single-threaded case — the common case for most Python programs — was a slowdown by a factor of two; so they abandoned the attempt [Stein, 2001].

Riley and Zilles [2006] proposed substituting GIL-protected critical sections with with transactions, and present an evaluation of the viability of this approach. Blundell, Raghavan, and Martin [2010] evaluated using transactional memory to improve Python's scalability using a simulator. I also investigated a similar approach using Sun's Rock machine.

Chapter 6 presents a more detailed discussion of using transactions in Python. In summary, with a certain level hardware support for transactions, Python interpreters can go from not scaling to scaling almost linearly. Even with limited best-effort hardware support, such as Rock's, some workloads benefit from transactions and scale with the number of threads.

3.1.3 Empirical User Studies

One of transactional memory's main appeal is that it should make it easier even for *non-experts* to reason about concurrent programs. Recent empirical studies investigated whether it is, in fact, easier for non-experts to program using transactional memory compared with traditional synchronization mechanisms. Those studies came to the same broad conclusions: transactional memory does make it easier to program.

The first study, by Rossbach, Hofmann, and Witchel [2009, 2010], involved 237 undergraduate students taking an operating systems course at the University of Texas at Austin. Each student was required to implement the same concurrent program, in Java, using coarse-grained locks, fine-grained locks, and software transactional memory. The program the students implemented simulates and displays a shooting gallery, where different randomized *shooter* threads shoot red or blue paint balls — so the lane changes color when a shooter shoots. Shooters are allowed to shoot a lane only if it is white, and only one shooter may shoot the same lane at the same time. There are also *cleaner* threads that clean the gallery only once all the lanes have been shot. The program graphically displays the simulated shooting range, which enables the students to visually identify some of the possible race conditions and bugs (Figure 3.1).

In terms of development time, the study reports that programming with transactional memory took more time than using coarse-grained locks, but less time than using fine-grained locks. The authors attribute this to the familiarity students already have with locks and the

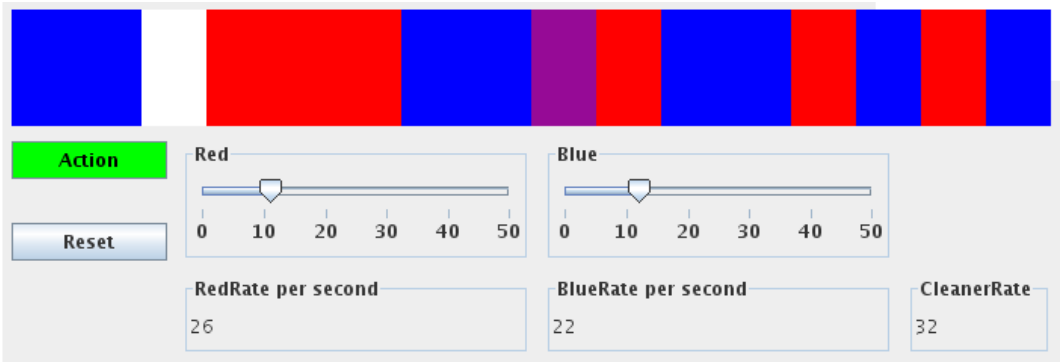

Figure 3.1: *A screenshot of the program the students were asked to implement. In the figure, the colored boxes represent 16 shooting lanes in a gallery populated by shooters. A red or blue box represents a box in which a shooter has shot either a red or blue paint ball. A white box represents a box in which no shooting has yet taken place. A purple box indicates a line in which both a red and blue shot have occurred, indicating a race condition in the program. Sliders control the rate at which shooting and cleaning threads perform their work. [figure and caption reproduced from Rossbach et al., 2010, with modifications]*

novelty of the transactional model. These results also correlate with the students' own reported experiences: the students thought that coarse-grained locks were easier to reason about than transactions, and that transactions were easier to reason about than fine-grained locks.

Perhaps the most significant observation was that over 70% of the students made programming errors when using fine-grained locks, whereas fewer than 10% made errors when using transactions.

The second study, by Pankratius, Adl-Tabatabai, and Otto [2009] at the University of Karlsruhe in Germany, involved 12 graduate students working in teams of two. The teams competed against each other to develop the fastest desktop search engine, according to predefined criteria, using either C or C++. Three teams were randomly assigned to use locks for their search engine, and three to use software transactional memory.

The first team to complete the assignment was one of the three transactional memory teams; moreover, that same team was the winning team according to the competition's criteria. On average, the transactional memory teams spent less time working on their projects, and less time debugging their code.

This study is limited in size, with only 12 students, and its findings are not statistically significant. Despite the differences between this study and the Texas study, its results are consistent with the Texas study, and strengthens the argument that transactional memory makes parallel programs easier to write and to debug.

Finally, there is a study by Lu, Park, Seo, and Zhou [2008], in which the authors examine 105 randomly selected concurrency bugs from four server and client applications: MySQL (database server), Apache (web server), Mozilla (web browser), and OpenOffice (office productivity suit).

Lu et al. concluded that using transactions instead of locks could have avoided about one third (41/105) of the examined concurrency bugs, specifically bugs related to atomicity violation of critical sections and deadlocks. The authors also believe that transactional memory could help avoid over a third *more* bugs (for a total of 85/105) if certain concerns are addressed by the underlying transactional implementation. These concerns include the ability to handle operations such as I/O inside transactions, and the ability to handle long-running transactions. The authors also found that the remaining bugs (20/105) *cannot* benefit from transactions, because these bugs violate programmer intentions, which the transactional model is not aware of — confirming that transactional memory is not a panacea.

I do not suggest that the evidence presented is conclusive — such an endeavor is beyond the scope of this thesis. I do believe, however, that this evidence makes a strong case for transactional memory. Therefore, the rest of the chapter argues that, for transactional memory to become a successful programming model, the best approach for implementing transactional memory is the hybrid approach.

3.2 Making a Case for Hybrid Transactional Memory

This section attempts to make a case for hybrid transactional memory; specifically, it argues that if transactional memory is to become successful, then it should adopt the hybrid approach. This argument can be summarized as follows.

Neither best-effort nor bounded hardware transactional memory, even when the bounds are big, are a complete transactional memory solution. Best-effort and bounded proposals do not guarantee that all transactions will eventually commit. Therefore, programmers using this type of hardware support must understand the limitations of this support, and write their programs around these limitations — this breaks the abstraction and goes against the transactional memory ideal. The alternative would be to use a software fallback mechanism when transactions cannot commit in hardware, which would result in a *hybrid* system.

Unbounded hardware transactional memory has two problems. It goes against the computer architecture design principle of providing primitives, and not solutions [Wulf, 1981], by requiring the complete design and policy of the transactional memory system to be hard-coded in the hardware. As described in current proposals, unbounded hardware transactional memory is also expensive and too costly for processor manufacturers to consider in practice, at least for the time being.

Software transactional memory is flexible and can run on existing systems today; however, its main problem is performance. All software transactional memory proposals I have investigated incur, at best, a 100% overhead compared with sequential execution, and for some workloads can be an order of magnitude slower than sequential execution. In the future, a software transactional memory with very low overhead might emerge, but that future does not appear imminent.

Hybrid transactional memory, which combines hardware and software, is in my opinion the best solution given the tools available. Hybrid transactional memory can run as fast as the underlying hardware in the common case, and is as flexible as its software component. Some hybrid proposals, such as NZTM (Chapter 5), can also run on existing systems today, benefiting from any enhancements to either of their underlying hardware or software components.

3.2.1 The Problem with Hardware Transactional Memory

> *Some architectures have provided direct implementations of high-level concepts. In many cases these turn out to be more trouble than they are worth.*
>
> *— William Allan Wulf [1981]*

Hardware transactional memory can be bounded or best-effort, which restricts the type of transactions it can run. These systems cannot handle many events that the hardware designer of that particular system deem too difficult to handle inside transactions, such as context switches, interrupts, I/O, or cache line evictions. Therefore, a bounded or best-effort hardware transactional memory, by itself, cannot provide a complete transactional memory solution.

Unbounded hardware transactional memory, on the other hand, is designed to handle *any* transaction in hardware — at least in theory. The size of the transaction does not matter, nor do events such as context switches and exceptions, no matter how difficult designers might consider them to be.

One problem with unbounded proposals is that policy, such as whether to use eager or lazy conflict detection or how to determine priority, is hardcoded in the hardware. This makes unbounded proposals less flexible and less than ideal for certain workloads, particularly because different workloads perform better under different policies [Ceze, Tuck, Torrellas, and Cascaval, 2006; Bobba et al., 2007; Shriraman et al., 2008; Minh et al., 2008; Tomić, Perfumo, Kulkarni, Armejach, Cristal, Unsal, Harris, and Valero, 2009]. Unbounded hardware transactional memory could be designed to accommodate more than one policy, but that would further complicate the design.

Moreover, because unbounded hardware transactional memory is a direct implementation of a high-level solution directly in hardware, it adds complexity to the hardware that possibly serves no purpose other than supporting this particular high-level abstraction. Applications that do not use this abstraction still have to pay for this additional complexity, unless the same hardware can be leveraged for other purposes as well. Wulf [1981] made a similar observation regarding other proposed solutions in hardware, and this observation is now known as the design principle that hardware should provide primitives and not solutions [Hennessy and Patterson, 2006].

Another problem is that unbounded proposals are complex, too complex for processor manufacturers to practically consider [Chung, Minh, McDonald, Skare, Chafi, Carlstrom,

Kozyrakis, and Olukotun, 2006b; Damron, Fedorova, Lev, Luchangco, Moir, and Nussbaum, 2006; Lev, Moir, and Nussbaum, 2007; Larus and Rajwar, 2007; Cascaval, Blundell, Michael, Cain, Wu, Chiras, and Chatterjee, 2008]. Much of this additional complexity in unbounded proposals is dedicated for cases that are expected to be rare, and for cases that are not critical for performance [Baugh, Neelakantam, and Zilles, 2008].

One example that demonstrates how complex unbounded proposals are is *Unbounded Transactional Memory* (UTM) [Ananian et al., 2005]. UTM is the first unbounded proposal, and it requires extensive modifications to the memory interface. UTM is so complex that even its authors acknowledge that it is not feasible, and instead propose a simplified and *bounded* approximation in the same work.

TokenTM [Bobba et al., 2008] is a more recent unbounded proposal that uses the abstraction of tokens to track transactional sharing states. For TokenTM to be unbounded, it must track its tokens in *all* of memory, which includes cache memory, main memory, and virtual memory. To track its tokens, TokenTM adds at least 16 bits per cache line, where cost is a major design constraint [Jafri, Thottethodi, and Vijaykumar, 2010]. TokenTM also adds 16 bits per 64 byte memory block, both to main and virtual memory, incurring a 3% space overhead. Because it is difficult to add bits to memory and be able to retrieve the data bits and the token bits in one access [Jafri et al., 2010], the authors suggest *stealing* those 16 bits from the 64 error control coding (ECC) bits. However, reducing the number of ECC bits available by 25% weakens error protection, which is becoming a bigger concern as memory density increases [Jafri et al., 2010].

Furthermore, unbounded proposals leave certain difficult problems aside without proposing a specific solution, suggesting them as topics for future work. For example, UTM and VTM [Rajwar et al., 2005] — another early unbounded proposal — cannot handle I/O inside transactions, and VTM does not define how it handles exceptions inside transactions. TokenTM can handle many different events, such as context switching and paging, but leaves unspecified "richer workloads" as a topic for future research. The problem remains, however, that there are always going to be different types of I/O and unexpected corner cases; handling them all in hardware, especially when new issues arise, is expensive or altogether infeasible.

3.2.2 The Problem with Software Transactional Memory

Software transactional memory is flexible: its policy can change, even dynamically at runtime, without additional cost in hardware. Moreover, software transactional memory does not require hardware support beyond what is already available for parallel programming. Programmers can use software transactional memory today on almost any platform, albeit mainly with experimental compilers.

The main problem with software transactional memory is performance. For example, in *Dynamic Software Transactional Memory* (DSTM) [Herlihy et al., 2003b], one of the earliest

software proposals, performance at a single thread is an order of magnitude slower than using a single global lock. Although DSTM scales whereas a single global lock does not, DSTM's throughput running on 72 processors does not even approach that of a single global lock running on a single processor. DSTM, however, was a proof-of-concept system not optimized for performance.

Transactional Locking 2 (TL2) [Dice, Shalev, and Shavit, 2006] is a more recent proposal that uses readers-writer locks as its underlying implementation, and was designed with performance as a goal. In the original evaluation of TL2, the performance of a single thread running TL2 is about a factor of two to three slower than a single thread using a single global lock. In a later evaluation of TL2 by the authors of the STAMP benchmarks [Minh et al., 2008], TL2 is about twice as slow as hardware implementations for STAMP benchmarks dominated by transactions, and is an order of magnitude slower in some cases. Furthermore, hardware transactional memory scales for all STAMP benchmarks, whereas TL2 scales poorly or not at all.

Even though TL2 was originally published in 2006, it is *still* considered a "state-of-the-art" software transactional memory [Dice et al., 2009a; Ramadan, Roy, Herlihy, and Witchel, 2009; Gottschlich, Vachharajani, and Siek, 2010; Harmanci, Gramoli, Felber, and Fetzer, 2010]. Many recent software proposals use TL2 as the *only* software transactional memory to evaluate their work against [Lev et al., 2009; Ramadan et al., 2009; Dice and Shavit, 2010].

Recently, Dalessandro, Spear, and Scott [2010] and Dice and Shavit [2010] investigated designing software transactional memory optimized for performance when running a small number of threads. The rationale being that, in the near future, a typical multicore computer will not have a large number of cores, and would benefit from software transactional memory proposals designed specifically for a small number of cores, such as 64 cores.

As evaluated by their authors, in the single threaded case, both proposals are about three times slower than using a single global lock when running on a single thread, or about four times slower compared with a single thread that does not incur any synchronization overhead. In other words, assuming perfect scalability, the workload for these proposals must run on at least four processors just to break even with running on a single thread. Contrast this with hardware transactional memory, which depending on the implementation, does not incur more overhead than a single global lock would in the single-threaded case, and scales when running additional threads.

Cascaval, Blundell, Michael, Cain, Wu, Chiras, and Chatterjee [2008], in their provocatively titled work, "Software Transactional Memory: why is it only a research toy?", evaluate different software algorithms and conclude that it is inherently difficult to lower the overhead of software implementations to what they consider to be an acceptable level. One cannot generalize to all software transactional memory proposals based on their results; however,

the overhead in software transactional memory does appear to be insurmountable, at least for the time being.

* * *

To understand why software transactional memory performs poorly, even relative to using a single global lock, we need to understand the overheads involved in each.

When using a single global lock, in the single-threaded case, the overhead is that of acquiring the lock, which typically involves expensive synchronization instructions, such as *Compare&Swap* and a memory barrier; and of releasing the lock, which is typically a normal store instruction and possibly another memory barrier. This overhead is not proportional to the length of the critical section: the bigger the critical section, the more this overhead is amortized. There is also a space overhead involved, that of storing the lock, which typically does not exceed a single cache line.

Conversely, there are two types of overhead for a typical software transactional memory proposal: instrumentation overhead of the additional instructions the particular algorithm needs, and space overhead, used by the data structures required to maintain the transactional metadata. Both these overheads are typically proportional to the size of the transaction.

The sources of overhead vary between different software transactional memory implementations. As an example from my work on NZSTM, which is presented in the next chapter, below are the sources of overhead for a transaction running on a single thread, i.e., without contention.

Beginning and committing a transaction: requires allocating a transaction descriptor for beginning a transaction, and uses a synchronization instruction, *Compare&Swap*, for committing a transaction. It is a constant overhead, relative to the length of the transaction.

Opening an object for writing: includes using a synchronization instruction for setting the object's owner to the transaction, and allocating and taking a backup copy of the object's data. It is proportional to the number of objects modified inside the transaction.

Opening an object for reading: includes using a synchronization instruction for adding the reader into a visible readers' slot. It is proportional to the number of objects read inside the transaction.

Memory allocation and management: NZSTM allocates objects to track the transactional metadata. This additional allocation and memory management adds to system overhead, which is proportional to the number of objects accessed inside a transaction.

Other metadata management: includes operations such as keeping statistics for contention management.

The effect of each of the above items is workload-dependent. To gain a better intuition on the effect of these overheads, Table 3.1 presents a breakdown of the time the `redblack` benchmark, a microbenchmark described in the next chapter, spends on the different sources of overhead when running NZSTM on a single thread.

Table 3.1: *Execution time share of different overhead sources in NZSTM for the* `redblack` *benchmark*

Overhead Source	Share of Execution Time
opening objects for reading	50%
opening objects for writing	10%
memory management	5%
other metadata management	5%
useful work (*not an overhead*)	30%

Others who have reported on a breakdown of their results of the same benchmark show similar overheads [Shriraman, Marathe, Dwarkadas, Scott, Eisenstat, Heriot, Scherer, and Spear, 2006].

The area of software transactional memory is in active research, and in the future researchers might develop an implementation with low overhead. However, until that happens, I do not believe that software alone will make the transactional model a viable programming model.

3.2.3 The Promise of Hybrid Transactional Memory

Hybrid transactional memory promises that, by using both hardware and software, it can deliver the best of both worlds: performance that matches that of the underlying hardware, and the flexibility of software for the cases hardware cannot support. Hybrid transactional memory that has a pure software fallback mechanism can also work on existing systems, albeit in software speed. Such a system would allow developers to start programming with transactions today, while benefiting from improvements in hardware as they are developed.

Primitives Not Solutions

Hybrid transactional memory embodies Wulf's principle of "provide primitives, not solutions" [Baugh et al., 2008]. Hybrid transactional memory proposes that certain primitives be implemented in hardware, primitives such as best-effort hardware transactional memory, and that software leverage those primitives for performance. Depending on the hybrid system, some of the primitives could also be used for applications that do not use the transactional model. For example, best-effort hardware transactional memory and its underlying hardware can be used to implement lock elision [Rajwar and Goodman, 2001; Dice et al., 2009a], speed up existing concurrent algorithms [Dice et al., 2010], and for thread-level speculation [Chaudhry et al., 2009b]. Moreover, hybrid transactional memory does not specify any

hardcoded policies in hardware, nor does it require complex hardware to deal with every conceivable case a transaction might encounter.

There is precedent in the history of processor design of adding hardware primitives to support new models. Two examples are virtual memory and hardware virtualization.

Virtual memory is an abstraction that gives programmers the illusion of an unlimited and contiguous memory, or address space, while guaranteeing that concurrent processes that share the same physical memory do not violate each other's space. For virtual memory to work efficiently, it needs additional hardware support in the memory management unit to ensure that memory protection restrictions are respected, to perform virtual to physical address translation, and to optimize address translation using a *translation lookaside buffer* (TLB). It is worth noting that virtual memory, which is used on most desktop and server machines today, was controversial at first [Tucker, 2004], until Sayre [1969] demonstrated that virtual memory consistently performs better than manual memory management.

Hardware virtualization allows users to run a virtual machine on top of a different platform. Virtual machines comprise complete hardware platforms and appear to the users as self-contained units. Hardware virtualization is used on servers where isolation and security of the different running components is important. It also used on desktop machines so users could run different operating systems on top of the main operating system, for example, to run Windows on an Apple OS X computer.

Hardware virtualization is possible without dedicated hardware support; however, hardware-assisted virtualization reduces the overhead of running a virtual machine, improving its performance. Even though hardware-assisted virtualization has existed since it was introduced by the IBM System/370 in 1972 [Hennessy and Patterson, 2006], it arguably did not become a mainstream concept until Intel and AMD added virtualization support to their processors in 2005 and 2006.

These examples demonstrate that, when a new idea gains enough momentum, and there is enough demand for it, processor designers add the necessary *primitives* to support them.

Can Hybrid Transactional Memory Deliver on Its Promises?

Hybrid transactional memory aims to run as fast as its underlying hardware, in the common case, while being as flexible as its underlying software in the sizes and the types of workloads it can handle.

This section demonstrates that hybrid transactional memory can deliver on its promises; this is based on my own experiences in designing a hybrid system (Chapter 5), and on the findings of other research groups in this area. The following presents some the work that evaluates different types of hybrid systems. This work shows that hybrid transactional memory performs significantly better than software transactional memory, and that with sufficient hardware support, hybrid transactional memory can be practically as fast as the underlying hardware allows.

One of the early hybrid proposals, *HyTM* [Moir, 2005; Damron et al., 2006], describes a hybrid system that first attempts transactions using best-effort hardware support. If a hardware transaction aborts repeatedly, for a number of times determined by the system's policy, it falls back on using software transactions. The results Damron et al. present, when running on a single thread, show that HyTM is about four times faster than using the underlying software component by itself. However, HyTM adds overhead to hardware transactions, and is 60% slower than using a single global lock also running on a single thread.

HyTM incurs overhead because hardware transactions must explicitly check for conflicts with software transactions. The overheads are aggravated because the software component HyTM uses does not locate the ownership records of data locations with the data itself; instead, the ownership records are on a separate table. Therefore, a hardware transaction could incur two cache misses when accessing a location: the first to check that an object is not owned by a competing software transaction, and the second to access the data. By contrast, a pure hardware transaction would incur a single cache miss for a similar access because it can access the data directly.

As for scalability, HyTM is at least twice as fast as its underlying software component, and up to four times faster in some benchmarks. However, at 32 threads, HyTM is about half the speed of pure hardware transactions, particularly in benchmarks that exhibit high contention.

HyTM shows that, under the right circumstances, hybrid transactional memory can bridge a significant portion of the gap between software and hardware transactional memory. It also shows that hybrid systems are significantly faster than just using software, while requiring much less complicated hardware than unbounded proposals. Moreover, these results of an *early* hybrid system are significantly better than any software transactional memory can achieve today, to the best of my knowledge.

As part of my research, I created a hybrid transactional memory, *NZTM*, which is discussed in more detail in Chapter 5. NZTM uses the same method as HyTM of attempting a transaction in hardware first, then falling back on software. NZTM is optimized for hardware transactions by collocating an object's data with its metadata, i.e., its ownership record, in the common case. Therefore, a hardware transaction running in NZTM typically incurs only one cache miss, instead of two, to check for conflicts with a software transaction and access the data.

I evaluated NZTM using microbenchmarks and STAMP benchmarks. The evaluation shows that NZTM performs significantly better than its underlying software component. It also significantly reduces the overhead transactions incur relative to a pure hardware scheme. Depending on the workload, this overhead is about 20–50%.

Hybrid systems such as HyTM and NZTM demonstrate the potential of using best-effort hardware transactions as the basic primitive for a hybrid system, while using software transactions as a fallback mechanisms. These, and other similar systems [e.g., Lie, 2004; Ana-

nian and Rinard, 2005; Kumar, Chu, Hughes, Kundu, and Nguyen, 2006], offer a significant improvement over using pure software transactions. However, they do not close the performance gap between software and hardware transactional memory, because of the overhead the hardware path incurs of checking for conflicts with software transactions. Even if software transactions are rare, these checks always add overhead in the common case.

Lev, Moir, and Nussbaum [2007], in their work on *Phased Transactional Memory* (PhTM), present one approach to eliminating this overhead for workloads that run almost completely in hardware, and for workloads that do not have many transactions or critical sections. Lev et al. suggest that the system should automatically move between different phases, depending on the workload, with each phase optimized for its current workload.

For example, if the system decides that the current workload can run successfully using hardware support alone, it can attempt to run all transactions in hardware, and dispense with the overhead of checking for conflicts with software transactions. Conversely, if the system decides that many transactions need software support, it would run in the same manner as the hybrid proposals described earlier, where hardware transactions must check for conflicts with software transactions. PhTM dynamically switches between the different phases, at runtime, based on certain heuristics such as the level of contention or abort ratio of the current workload.

Some of the phases Lev et al. propose are the following.

Hardware: Hardware support is sufficient to commit all transactions, and there is no need to check for conflicts with software transactions.

Hybrid: Hardware support is available but might not be sufficient to commit all transactions; therefore, the system runs using best-effort hardware, falling back on software when needed.

Software: Hardware support is not available, or not sufficient for the current workload. The system runs all its transactions in software, without attempting to run them in hardware first — which in this case would be wasted attempts.

Sequential: A single global lock is used to protect transactions. This phase is suitable for workloads that are not dominated by transactions, such as when there are few threads running in the system and their transactions are infrequent and short.

PhTM does not eliminate all overhead, because transactions must at least check to see which phase they are in, an unnecessary check in a system that supports only one type of transactions. This check, however, is amortized across the whole transaction, and could be negligible for longer transactions. Furthermore, PhTM requires additional logic for the heuristics that decide whether and when to change phases; if the system does not change phases often, this overhead could be negligible.

The results Lev et al. present show that, for workloads the underlying system can commit successfully in hardware most of the time, PhTM bridges most of the gap with pure hardware for long running transactions, and significantly reduces the gap for smaller transactions, without sacrificing scalability as the number of threads increases.

Baugh, Neelakantam, and Zilles [2008] presented an evaluation of another implementation of PhTM, which assumes best-effort hardware transactional memory support capable of committing most transactions. Their evaluation shows that PhTM is capable of performance comparable to pure hardware transaction even when running a single thread, and that it scales as well as pure hardware transactions in most workloads.

In the same work, Baugh et al. presented another approach to eliminate the performance gap between hybrid transactional memory and pure hardware transactions, by introducing a new hardware primitive that aids in conflicts detection between hardware and software transactions.

Baugh et al. recognized that the main source of overhead when running a hardware transaction in a hybrid system, such as HyTM, is that of ensuring that hardware detects conflicts with concurrent software transactions. Rather than burden the fast hardware transactions with explicitly checking for conflicts with software transactions, Baugh et al. suggest that it should be the responsibility of the slow software transactions to ensure hardware transactions cannot access data in a conflicting manner.

To that end, Baugh et al. proposed a hardware-supported system of fine-grained memory protection. This system adds two tag bits to memory, a read bit and a write bit, which are set by software transactions when they acquire locations for reading or for writing, and are reset when a software transaction commits or aborts. When a thread accesses a location whose corresponding tag bit is set in a conflicting manner, e.g., it writes to a location whose read tag bit is set, the system triggers a fault, which runs a conflict management routine. This allows transactions, hardware and software, to run without explicitly checking for conflicts with other software transactions, reducing the overhead of conflict detection for both.

Baugh et al. presented an evaluation of a hybrid transactional memory that uses fine-grained memory protection, and attempts transactions using hardware support falling back on software when needed. They found that single-threaded performance is virtually the same whether using unbounded hardware support or their hybrid system, and that their hybrid system scales as well as pure hardware transactional memory for most workloads. These results show that using certain hardware primitives can close the performance gap between hybrid and pure hardware transactional memory, assuming most transactions commit successfully in hardware.

Another approach to hybrid transactional memory is to run transactions in software, and use hardware support to optimize the bottlenecks of the software implementation.

An example of this approach is a hybrid proposals by Shriraman et al. [2008][1], who identified four sources of overhead in a typical software transactional memory implementation: read-set validation, maintaining a copy of modified data, conflict detection, and conflict managements. Shriraman et al. proposed hardware primitives, fully visible and accessible in software, to eliminate these overheads. These primitives provide hardware support to perform conflict detection and conflict tracking between transactions, and allow transactions to leverage a processor's inherent buffering to relieve thread from explicitly maintaining two copies of data modified inside a transaction.

The functionality of the primitives Shriraman et al. propose is not new; it is present in many unbounded hardware proposals. In my opinion, one of their main contributions is the separation of the different aspects of unbounded hardware system into software-visible hardware *primitives*, which transactions could use to improve performance. Moreover, optimized software transactions do not require *all* these primitives, and can benefit even from a subset, which allows processor manufacturers to support these primitives incrementally. By separating the hardware support into clearly defined primitives, the transactional system also has flexibility in conflict management and other policies not present in an unbounded system.

The results Shriraman et al. present show that their proposal imposes little overhead in the single-threaded case compared with a single global lock, about 10–20%. Their proposal also scales as the number of threads increases. Shriraman et al. do not present an evaluation of their proposal against an unbounded hardware transactional memory.

The final approach to hybrid systems presented herein is one that takes the opposite approach: instead of using hardware to optimize a software implementation, a hybrid system can use software to handle the difficult cases in a hardware system. This approach takes advantage of the observation that much of the complexity in unbounded proposals is dedicated for cases expected to be rare, and cases not critical for performance [Baugh et al., 2008]. This is the approach Jafri, Thottethodi, and Vijaykumar [2010] use in their work.

Jafri et al. observed that, even after an unbounded hardware transactional memory has overflowed, typically by exceeding the resources available in the L1 cache, most of a transaction's active data remains in the L1 cache. They suggest that, instead of aborting the hardware transaction and falling back on software, transactions should track the overflowed data in a software structure, and rely on software to detect and resolve conflicts involving overflowed data. Unlike unbounded hardware transactional memory proposals, which rely on hardware to manage both overflowed data and the data that remains in the cache, they use software to manage the overflowed data and hardware to manage the data that remains in the cache. Because a transaction's active data remains in the cache, software is not invoked often, and fast hardware is used for the common case.

[1] Shriraman et al. [2008] extend the proposal by Shriraman, Spear, Hossain, Marathe, Dwarkadas, and Scott [2007].

As an example of applying these observations to a transactional implementation, Jafri et al. propose *LiteTM*, where they apply these observations by modifying TokenTM [Bobba et al., 2008] to reduce its complexity. To track its tokens, TokenTM adds 16 bits to each memory block in cache memory, main memory, and virtual memory. Instead, LiteTM adds two bits to each memory block. LiteTM uses these bits to mark memory blocks that transactions read or modify, similarly to Baugh et al. [2008]. As long as the memory blocks remain in the L1 cache, LiteTM relies on hardware, in the same manner as best-effort hardware, to detect and resolve conflicts. Once a cache line is evicted from the L1 cache, LiteTM uses software for detecting and resolving conflicts *only* with the evicted blocks, continuing to rely on hardware for the blocks that are in its cache.

Jafri et al. evaluate LiteTM, and compare it against TokenTM. They find that LiteTM performs within 4% of TokenTM on average, and within 10% of TokenTM in the worst case.

In my opinion, however, LiteTM has two drawbacks. The first one is that it implements the solution in hardware, rather than expose the primitive to the software. This hardcodes the policy in the hardware, and makes the solution less applicable to contexts outside transactional memory. The second one is that LiteTM does not address the issues that the TokenTM authors left for future work, namely investigating their proposal using "richer workloads".

3.3 Concluding Remarks

This chapter made a case for transactional memory as a programming model, focusing on its promise of simplifying the task of writing parallel programs that perform well. This chapter argued that, if transactional memory is going to be successful, it should adopt the hybrid approach. The argument is summarized as follows.

Best-effort and bounded hardware support are not sufficient, because even if the bounds are big, programmers must handle the corner cases of that particular implementation, which would, at least partially, negate the advantages of abstraction. Unbounded hardware transactional memory is too complex to implement in practice, leaves some difficult issues unresolved, and is contrary to the principle of designing hardware primitives and not solutions. Software transactional memory, although flexible, is too slow: for current software proposals, it takes at least four threads to break even with a single thread that does not use any synchronization mechanisms.

Therefore, hybrid transactional memory presents the best of the hardware and the software worlds. It can run as fast as the underlying hardware, in the common case, and is flexible because it has the option of falling back on software for any difficult or corner case the hardware designer might not have anticipated. Some hybrid proposals can also run on existing systems today, benefiting transparently from any improvements to their underlying hardware or software components. More important, hybrid systems can be designed to leverage hard-

ware *primitives*, without requiring a whole transactional memory solution to be hardcoded in hardware.

This thesis focuses on hybrid systems that first attempt a transaction using best-effort hardware transactional memory, and fall back on software for the cases which best-effort is not sufficient. I do not believe that this is necessarily the best way of implementing a hybrid system. Other hardware primitives may be better suited for supporting hybrid transactional memory. I chose this path, in part, because best-effort hardware transactional memory is the only primitive capable of supporting a hybrid system that has been implemented on real processors to date; therefore, it is the most logical starting point when researching hybrid transactional memory. This issue is further discussed in the concluding chapter of this thesis (Chapter 8).

Chapter 4

Nonblocking Zero-indirection Software Transactional Memory

> *All problems in computer science can be solved by another level of indirection ... except for the problem of too many levels of indirection.*
>
> — *Various attributions*[1]

This chapter describes *Nonblocking Zero-indirection Software Transactional Memory* (NZSTM). NZSTM is a nonblocking *object-based* software transactional memory, where data objects have headers that can be easily located whenever the application accesses an object. NZSTM is also designed to perform well when used in a hybrid transactional memory system, which is the topic of the next chapter.

The first object-based dynamic software transactional memory system was DSTM [Herlihy et al., 2003b], a nonblocking algorithm that requires two levels of indirection to access the data. The additional levels of indirection make it easier for the algorithm to be nonblocking, because with indirection, an object can be changed atomically just by changing a single-word value, that of the pointer to the object. Instructions that can change single-word values atomically, such as *Compare&Swap*, are available on most modern processors.

The problem with indirection, however, is that each level of indirection points to a separate object, which is typically located on a different cache line. Therefore, each level of indirection is a potential cache miss, which can have an adverse effect on performance (Figure 4.1).

Realizing the negative aspects of indirection, later software transactional memory proposals, such as OSTM and RSTM [Fraser, 2004; Marathe et al., 2006], reduce indirection to one level in the common case, while remaining nonblocking (Figure 4.2).

Others have proposed *blocking* implementations that store object data *in-place*, avoiding cache misses caused by indirection to reach the data [e.g., Dice et al., 2006; Ennals, 2006;

[1] The first part of the quote has been attributed to David Wheeler and to Butler Lampson. The second part of the quote has been attributed to Kevlin Henney.

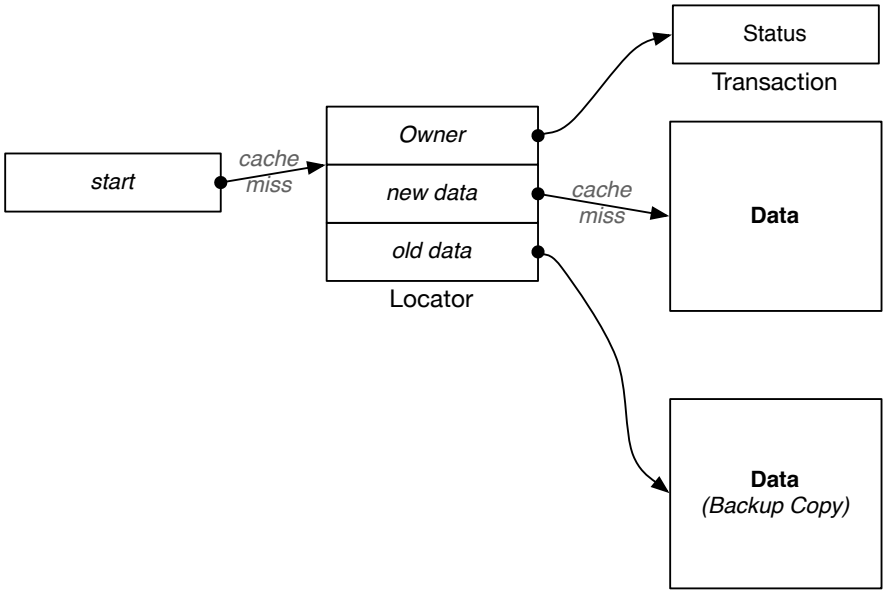

Figure 4.1: *The structure of DSTM's main transactional object. To access the data, a transaction follows the* `start` *pointer, and then follows either the* `new` *or the* `old` *data pointers, depending on the state of the transaction associated with the object. These objects are typically located on different cache lines, which could result in a cache miss for each object accessed. [adapted from Herlihy et al., 2003b]*

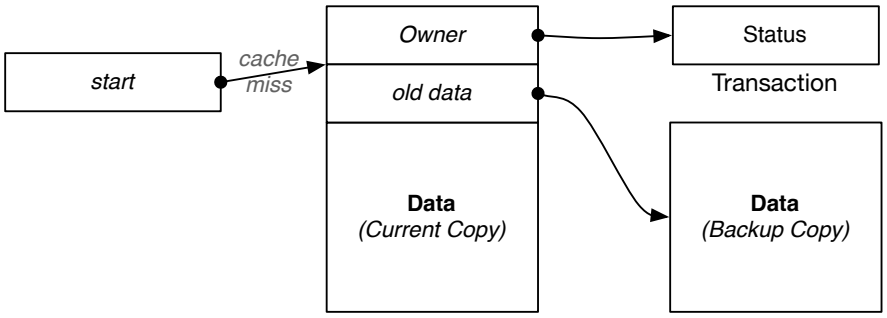

Figure 4.2: *The (simplified) structure of RSTM's main transactional object. To access the data, a transaction follows the* `start` *pointer to the object where the actual data is located in the common case. [adapted from Marathe et al., 2006]*

Saha, Adl-Tabatabai, Hudson, Minh, and Hertzberg, 2006; Herlihy, Luchangco, and Moir, 2006]. One such proposal is the *DSTM2 Shadow Factory* (DSTM2-SF) [Herlihy, Luchangco, and Moir, 2006], which allocates a backup field for every data field in the object. This backup field holds a backup copy of its corresponding data field when an object is being modified inside a transaction (Figure 4.3).

The performance experiments of proposals that store object data in-place confirm the intuition that this approach results in significantly better performance than those involving indirection in all cases. However, all of these proposals sacrifice the nonblocking progress properties provided by the earlier ones, and most of their proponents have implied, or argued

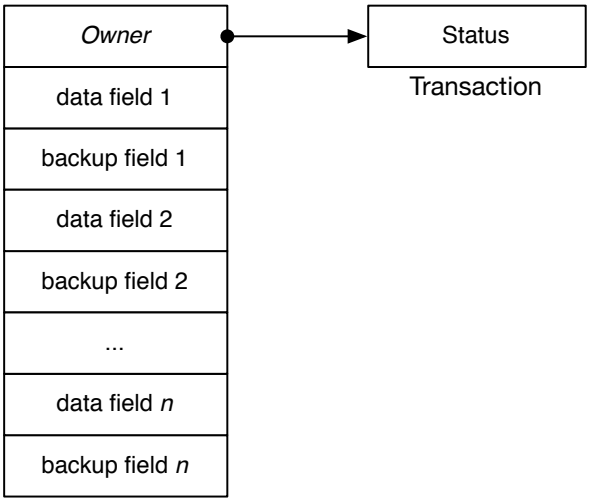

Figure 4.3: *The structure of the DSTM2 Shadow Factory main transactional object. The object data and metadata are located on the same object, and are likely to be on the same cache line as long as they fit. [adapted from Herlihy et al., 2006]*

directly, that this is fundamentally necessary in order to store object data in-place and collocate metadata with object data. For example, the DSTM2-SF algorithm is blocking because it acquires a lock when it copies data to or from the backup fields.

Proponents of blocking software transactional memory also argue that, in *some* cases, it is possible to avoid the disadvantages of blocking algorithms [Dice et al., 2006; Ennals, 2006]. For example, the Solaris `schedctl` function can *discourage* (not prevent) the scheduler from preempting a thread during the blocking part of a transaction. Nonetheless, without an implementation that is truly nonblocking, a program can still experience the disadvantages of blocking. For example, if a transaction experiences a long delay because of a page fault or being preempted, this can lead to many other transactions having to also wait for a long time.

Blocking is more than *merely* a performance concern, as some have implied. As explained in Chapter 2, it is *unacceptable* for an interrupt handler to be blocked by the thread it has interrupted; the design of interrupt handlers is often significantly complicated by this restriction. Transactional memory can help, but *only* if it is nonblocking. It is therefore important to continue research on nonblocking transactional memory, despite the appeal of simpler blocking implementations.

NZSTM shows that it is *not* necessary to sacrifice nonblocking progress guarantees in order to store data in-place in the common case. To that end, NZSTM stores object data in-place in the common case, and resorts to indirection only when a thread encounters a conflict with an unresponsive thread. By contrast, blocking proposals *must* block in such cases.

Spear, Shriraman, Dalessandro, Dwarkadas, and Scott [2007] have had similar insights about the importance of eliminating indirection to improve the performance of nonblocking software transactional memory. Their work was concurrent with and independent of this work, and concentrates on a different design point, namely the use of special *Alert On Update*

hardware to make nonblocking progress properties possible. Although NZSTM is explicitly designed to be able to take advantage of hardware support to achieve similar benefits, this proposal also includes an algorithm that can run on existing systems today, without additional hardware support.

Ananian and Rinard [2005] propose a nonblocking software transactional memory that eliminates indirection *only* when reading an object; however, transactions that perform writes still require indirection. Their algorithm employs a variant of the *LoadLinked-StoreConditional* atomic hardware primitives that is not supported in any modern system. NZSTM, by contrast, requires only *Compare&Swap*.

Marathe and Moir [2007] present a nonblocking word-based software transactional memory that eliminates much of the overhead of previous nonblocking word-based proposals by storing data in-place in the common case, and resorting to more complicated and expensive techniques to displace data only when necessary because of a conflict with an unresponsive transaction. The design philosophy for NZSTM was inspired in part by their work, but the details are quite different because they address word-based implementations, which cannot employ object headers and cannot collocate metadata with data.

The main contributions presented in this chapter are the following.

- I describe NZSTM, the first nonblocking object-based software transactional memory that does not require indirection to access data in the common case, and does not rely on special hardware support.

- I show how NZSTM can be substantially simplified using simple hardware transactions, if available. The effectiveness of this technique is evaluated on a Rock system, which supports best-effort hardware transactions.

- I present a correctness evaluation for NZSTM with model checking using *Spin*, a tool that mechanically verifies the correctness of distributed system models.

- I present a performance evaluation for NZSTM using a variety of benchmarks, and compare NZSTM with DSTM2-SF, a blocking software transactional memory that never requires indirection to access data. The results show that NZSTM's performance closely tracks DSTM2-SF's.

The remainder of this chapter is organized as follows. Section 4.1 describes the design of NZSTM. Section 4.2 discusses using model checking and runtime stress tests to evaluate the correctness of NZSTM. Section 4.3 presents a performance evaluation of NZSTM. Finally, Section 4.4 concludes this chapter.

4.1 The NZSTM Algorithm

The main goal in the design of NZSTM is a nonblocking object-based software transactional memory that provides performance competitive with object-based blocking designs, and that also performs well when used in a hybrid system. Towards that end, NZSTM's approach is to store data in-place, in the common case, to eliminate the costly indirection in previous nonblocking proposals.

A key difficulty in designing a nonblocking software transactional memory that stores object data in-place is the uncertainty that arises when one transaction, T_1, is updating an object, and another transaction, T_2, wishes to access the same object. T_2 cannot simply wait for T_1 to complete, because this would be blocking. T_2 can attempt to inform T_1 that it should stop modifying the object, but until T_2 can determine that T_1 has become aware that it should stop, it is not safe for T_2 or other transactions to update the object data in-place, because T_1 may still overwrite the data. Therefore, it is hard to see an alternative to storing the object data somewhere other than its natural home in this case. This leads to indirection and its associated overhead during the period that T_1 is unresponsive.

NZSTM differs from previous nonblocking software transactional memory proposals (e.g., DSTM and RSTM) in that, rather than actively abort a conflicting transaction, an NZSTM transaction can *request* that another transaction abort itself, and wait a short time until it does. If the transaction does abort, the uncertainty is resolved, and the transaction can continue to access the data in-place. Thus, NZSTM can generally avoid the overhead of introducing indirection, except when the conflicting transaction is unresponsive. By eliminating indirection, this approach largely eliminates the performance gap between previous blocking and nonblocking object-based transactional memory implementations.

This section describes the NZSTM algorithm. It starts by describing NZSTM's programming interface. It then describes a *blocking* object-based algorithm that stores object data in-place and collocates metadata with objects, which is extended to the nonblocking NZSTM.

To simplify the discussion, this section first describes a nonblocking algorithm that acquires objects exclusively and does not support read sharing, and then describes the read sharing algorithm used.

4.1.1 The NZSTM Programming Model

NZSTM is written in the C programming language as a statically-linked library, and uses a programming model derived from the Java-based DSTM [Herlihy et al., 2003b]. The NZSTM model requires programmers to explicitly identify all objects that transactions might access. It also requires transactions to explicitly acquire permissions for objects they will read or modify before they access those objects' data. The NZSTM interface is defined using the following C macros.

OBJECT_INIT(object, clone): Marks an object that could be accessed inside a transaction and initializes its data structures. The `clone` parameter identifies the object's `Clone` function, which creates a copy of the object's data.

BEGIN_TXN(): Begins a new transaction.

COMMIT_TXN(): Attempts to commit a transaction, and tries the transaction again if it aborts.

OPEN_WRITE(object): Attempts to acquire exclusive (write) permissions to the object.

OPEN_READ(object): Attempts to acquire non-exclusive (read) permissions to the object. The transaction can request to upgrade its permission later using `OPEN_WRITE()`.

The DSTM programming model was designed to facilitate experimentation, and is not intended for production use. Rather, it is envisaged that future language and compiler support might target such an interface, with programmers writing to a higher-level programming interface [Dalessandro, Marathe, Spear, and Scott, 2007]. Developing programming models, however, is outside the scope of my research. For this thesis, the DSTM interface, and the variant used, is sufficient for experimenting with alternative implementations.

4.1.2 NZSTM Data Structures

The basic data structures NZSTM uses are shown in Figure 4.4.

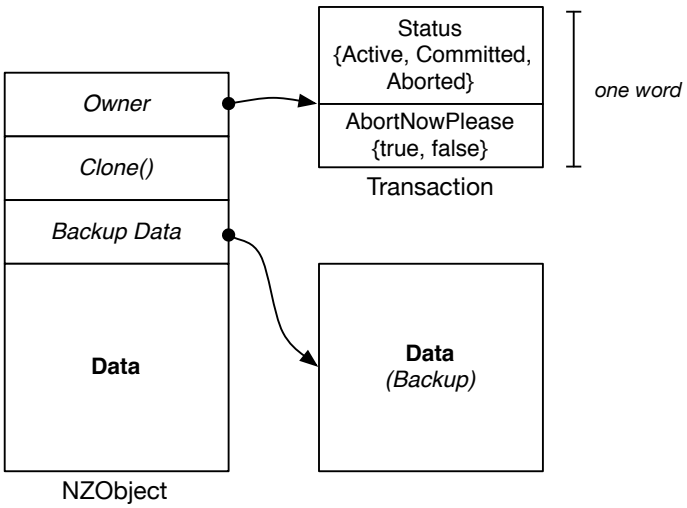

Figure 4.4: *The structure of an `NZObject` and a `Transaction`. The `Data` field is the actual data and can have any size or structure.*

The `NZObject` structure encapsulates a program object that NZSTM transactions can access, and serves as a container for its data and metadata. It is analogous to other containers,

such as DSTM's and RSTM's object headers. The `NZObject` structure contains the following fields. The `Owner` field, which if non-`NULL`, points to the last transaction to acquire this object. The `Clone` field points to the function that copies the object. The `Backup Data` field points to a backup copy of the object kept while a transaction that modifies the object is in progress. Finally, the `Data` field contains the actual object data. Because the object data is stored at a fixed offset from the start of the object, no indirection is required to access it.

NZSTM's `Transaction` is similar to the transaction descriptors used by previous nonblocking software transactional memory proposals (e.g., DSTM and RSTM). NZSTM creates a new `Transaction` object for every new transaction, and does not reuse these objects regardless of whether a transaction commits or aborts. Even when NZSTM attempts an aborted transaction again, it creates a new `Transaction` object. Having the transaction descriptor as a separate object facilitates making the changes to all objects modified by a transaction appear atomic; this will become clearer once the detailed description of the algorithm is presented.

Each `Transaction` object contains the transaction's `Status` field, which can be in one of the following states: `Active`, which indicates that the transaction is currently running; `Committed`, which indicates that the transaction has committed successfully; and `Aborted`, which indicates that the transaction has aborted and is no longer running.

In contrast to previous nonblocking proposals, NZSTM's `Transaction` has an additional flag, the `AbortNowPlease` flag. When this flag is set, the transaction interprets it as a request to abort itself. This flag is stored together on the same word as the `Status` field, so both may be accessed atomically using a single *Compare&Swap*, or similar atomic single-word instructions. This operation must be atomic to allow NZSTM transactions to request that other transactions abort, while simultaneously ensuring that the transaction being asked to abort has not committed in the interim.

At initialization, the `Data` field of an `NZObject` contains the initial value of the object data, `Clone` points to a function that creates a copy of the data, and the remaining fields are `NULL`.

The `Clone` function should ideally be created automatically by the compiler or the runtime environment. Alternatively, it could be modeled after the concept of a *copy constructor* in C++ [Stroustrup, 2000]. Because this prototype of NZSTM does not assume compiler support, it is the programmers' responsibility to write the `Clone` function for all transactional objects.

4.1.3 A Blocking Algorithm

In this blocking software transactional memory, a thread begins a transaction by creating a new `Transaction` object, with its status set to `Active` and its `AbortNowPlease` flag not set. The thread then executes its transaction, *acquiring* each object it accesses, by opening it for reading (shared), or writing (exclusive). When it completes the execution of its

transaction, the thread attempts to atomically change its transaction's status from `Active` to `Committed`, while ensuring that `AbortNowPlease` is not set. During execution, another transaction that detects a conflict with this one may either wait or attempt to abort it; this decision is made by an out-of-band contention manager, which may apply different policies.

Unlike DSTM and similar nonblocking proposals, a transaction, T_1, does *not* explicitly abort a conflicting transaction, T_2. Instead, T_1 *requests* that T_2 abort itself; this request is made by atomically setting T_2's `AbortNowPlease`, and confirming afterwards that T_1 itself has not been asked to abort, i.e., that T_1's `AbortNowPlease` is not set. When a transaction observes that its own `AbortNowPlease` flag is set, it *must* abort and set its own `Status` field to `Aborted` as an acknowledgement. The requesting transaction waits for this acknowledgement before proceeding to acquire the object on which the conflict occurred; because of this waiting, this algorithm is blocking.

Normally, the `Data` field of an `NZObject` contains the object's current data. A transaction that wishes to access the object acquires ownership by atomically placing a pointer to its `Transaction` in the object's `Owner` field. Before modifying the `Data` field, the acquiring transaction creates a copy of the data (using the object's `Clone` function), and points the object's `Backup Data` field to that copy. If the transaction aborts, the backup copy can be lazily restored by another transaction, undoing the aborted transaction's effects.

NZSTM does not store the backup copy in-place with the `NZObject`, unlike DSTM2-SF, which incurs 100% space overhead as a result. Instead, NZSTM allocates the memory for the backup copy from a thread-local memory pool. Unless a transaction aborts, this backup is not accessed by other threads. Thus, a thread can achieve good cache locality by reusing its thread-local memory for these backups.

To acquire an object, a transaction, T, first determines if it has already acquired the object by examining its `Owner` field. If it has not, T must ensure that there are no conflicts with other transactions before acquiring ownership of the object. If the `Owner` field is `NULL` or points to a committed or an aborted transaction, there is no conflict, and T can atomically change the `Owner` field to point to its `Transaction`. If the `Owner` field points to an active transaction, then a contention manager is consulted, and depending on the outcome, T either waits or requests that the active transaction abort itself.

Once all conflicts have been resolved and T has pointed the `Owner` field to its own `Transaction`, if the previous owner of the object was an aborted transaction, T restores the backup copy (indicated by the `Backup Data` field) if there is one. Otherwise, T creates a new backup copy. Finally, T *validates* by checking that its own `AbortNowPlease` flag is not set — if it is set, T must abort and set its own `Status` field to `Aborted` as an acknowledgement. If `AbortNowPlease` is not set, then T has successfully acquired the object, and can now access the data.

Validating at other times, by checking that the `AbortNowPlease` flag is not set, is not necessary. However, it may be desirable for performance reasons; for example, validating

before asking another transaction to abort, or before waiting for another transaction to abort, may avoid unnecessary aborts and waiting.

The following simplified code summarizes the basic blocking algorithm. For the interested reader, Appendix B contains a more precise description.

To begin a new transaction

```
BEGIN_TXN()
{
    Transaction *txn = new Transaction;
    txn->Status = Active;
    txn->AbortNowPlease = false;
    myThread->CurrentTxn = txn;
}
```

To commit a transaction

```
COMMIT_TXN()
{
    Transaction *txn = myThread->CurrentTxn;

    atomic {
        /* can be performed using Compare&Swap */
        if (txn->Status == Active && txn->AbortNowPlease = false) {
            txn->Status = Committed;
        } else {
            txn->Status = Aborted;
        }
    }

    if (txn->Status == Aborted) {
        /* Use C's longjmp to transfer control and restart. */
        restart_transaction();
    }
}
```

To acquire an object (exclusively)

```
OPEN_WRITE(object)
{
    while(true) {
        Transaction *currentOwner = object->Owner;

        if (currentOwner == myThread->CurrentTxn) {
            /* the object is already acquired by this thread's current transaction */
            break;
        } else if (currentOwner == NULL || currentOwner->Status != Active) {
            /* the object is not currently acquired by an active transaction */
            atomic {
                /* can be performed using Compare&Swap */
                if (object->Owner == currentOwner) {
                    object->Owner = myThread->CurrentTxn;
                } else {
                    continue;
                }
            }

            if (currentOwner->Status == Aborted && object->BackupData != NULL) {
                /* Restore the backup copy. */
                object->Clone(object->BackupData, object->Data);
            } else {
```

```
                /* Create a new backup copy. */
                object->Clone(object->Data, object->BackupData);
            }

            break;
        } else if (currentOwner->AbortNowPlease) {
            /* The transaction has been asked to abort, wait for it to abort. */
            while(currentOwner->Status != Aborted);
        } else {
            /* The transaction is active, consult the contention manager. */
            if (contentionManager->shouldAbort(currentOwner)) {
                atomic {
                    /* can be performed using Compare&Swap */
                    if (currentOwner->Status == Active) {
                        currentOwner->AbortNowPlease = true;
                    }
                }
            }
        }
    }

    /* validate */
    if (myThread->CurrentTxn->AbortNowPlease) {
        myThread->CurrentTxn->Status = Aborted;

        /* Use C's longjmp to transfer control and restart. */
        restart_transaction();
    }
}
```

4.1.4 NZSTM: Making the Algorithm Nonblocking

This section describes two ways to make the software transactional memory algorithm described above nonblocking, thereby ensuring that a transaction can always make progress even in the face of conflicts with unresponsive transactions.

Inflating the Object and Displacing Data

NZSTM can *inflate* an object, and use techniques similar to existing nonblocking object-based software transactional memory proposals, to temporarily displace the logical data into a different location than the `Data` field of the `NZObject`. NZSTM resorts to this approach only when an aborted transaction is unresponsive, which my evaluation shows to be rare. By contrast, previous nonblocking object-based software transactional memory proposals involve at least one level of indirection in *all* cases.

When an object needs to be inflated, NZSTM inflates it into an object similar to DSTM's transactional object (Figure 4.5). Once an object is inflated, transactions use the DSTM algorithm to acquire the object (Appendix A). This inflation introduces two levels of indirection; however, it is applied *only* when needed, and can be reversed once the unresponsive transaction becomes responsive again.

In the inflated object (Figure 4.5), the pointer to the first level is analogous to DSTM's `start` pointer. This pointer is integrated into the `NZObject` by overloading the `Owner` field. NZSTM indicates that the `Owner` field should be treated as an inflated object by setting its

low order bit, effectively changing the meaning of the `Owner` field to an `Inflated Object` field.

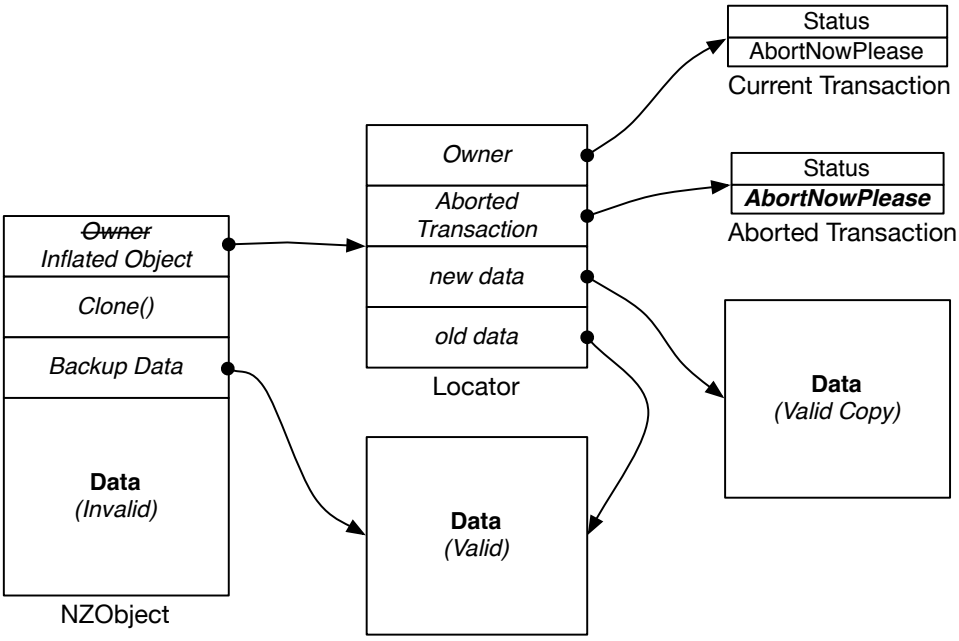

Figure 4.5: *An `NZObject` immediately after being inflated. The `Owner` field's low order bit indicates how the object is interpreted.*

Because of the indirection and additional logic, accessing an inflated object costs considerably more than accessing a non-inflated one. However, I expect that the need to inflate objects is rare. Moreover, NZSTM inflates objects *only* in cases in which blocking algorithms would have no choice but to wait for an unresponsive transaction, which is likely to be considerably more expensive.

This approach is now described in more detail. When a transaction, T, has requested the current owner of an object to abort, and the owner has not responded, T may decide not to wait any longer, and to inflate the object into a DSTM-like object. To do so, (1) T creates a DSTM-like `Locator`, a data structure used to track the object's metadata; (2) T points the `Owner` field of the `Locator` to T's `Transaction`; (3) T points the locator's `old data` field to the backup copy created by the unresponsive transaction, or to a new copy of the data if there is no backup[2]; (4) T points the locator's `new data` field to a copy of this backup which it creates using the `Clone` function. NZSTM's `Locator` object also has an `Aborted Transaction` field, which points to the unresponsive thread's `Transaction` (Figure 4.5).

After creating a new `Locator`, T checks that T itself does not have any pending abort requests by checking its own `AbortNowPlease` flag (aborting if it does), that the unresponsive transaction is still unresponsive, and that the object has not been acquired or inflated by another transaction. If any of these assumptions do not hold, T tries to acquire the object

[2]This can happen only if the unresponsive transaction became unresponsive in the process of acquiring the object, in which case it has not yet modified the object.

again. In the absence of interference from other threads, this is guaranteed to eventually succeed; hence it is nonblocking, or more specifically, *obstruction free* [Herlihy, Luchangco, and Moir, 2003a].

If these checks succeed, T attempts to atomically swap the `Owner` field from pointing to the unresponsive transaction's `Transaction` object to the newly created `Locator`, in the process setting the `Owner` field's low order bit to indicate that the object now points to a DSTM-like `Locator` rather than an NZSTM `Transaction`. This results in a state like the one in Figure 4.5. Henceforth, the object is treated as a DSTM object and the nonblocking DSTM algorithm applies (Appendix A), with the addition that each new `Locator` introduced contains the `Aborted Transaction` field from the replaced `Locator` to preserve the identity of the unresponsive transaction.

Once an unresponsive transaction finally aborts itself, NZSTM knows it is no longer modifying the `Data` field of the object. It is then desirable to *deflate* the object, by restoring it to a normal `NZObject`, so subsequent transactions can again enjoy the performance benefits of accessing the object data in-place. A transaction, T, can do this as follows.

T acquires the object exclusively according to the normal DSTM method (Appendix A). T then verifies that the unresponsive transaction has indeed aborted, and that T itself has no pending abort requests. If that is the case, T atomically swaps the `Backup Data` field of the `NZObject` to point to the valid data. If this swap is successful, T atomically swaps the `Transaction` field to point to itself, and copies the backup data back to the `Data` field (Figure 4.6.). This ensures that, if T were to abort at any time, the valid data is readily accessible through the `Backup Data` field. If any of the preceding steps fail, T abandons the deflation.

Once the object is deflated, T, and any subsequent transactions, can again access this object as a normal `NZObject`.

If an unresponsive thread crashes, then objects its transaction has acquired cannot be deflated. This means that these objects' inflation, and the slowdown associated with it, becomes permanent. Nevertheless, this slowdown is still preferable to the alternative in blocking systems, where a crashed thread could cause data corruption, or bring the whole system to a halt.

When using this nonblocking algorithm, an NZSTM transaction must check if an object it tries to acquire is inflated to determine whether the transaction can access the data in-place or whether it should apply the DSTM algorithm. This additional check might incur additional overhead compared with the blocking algorithm.

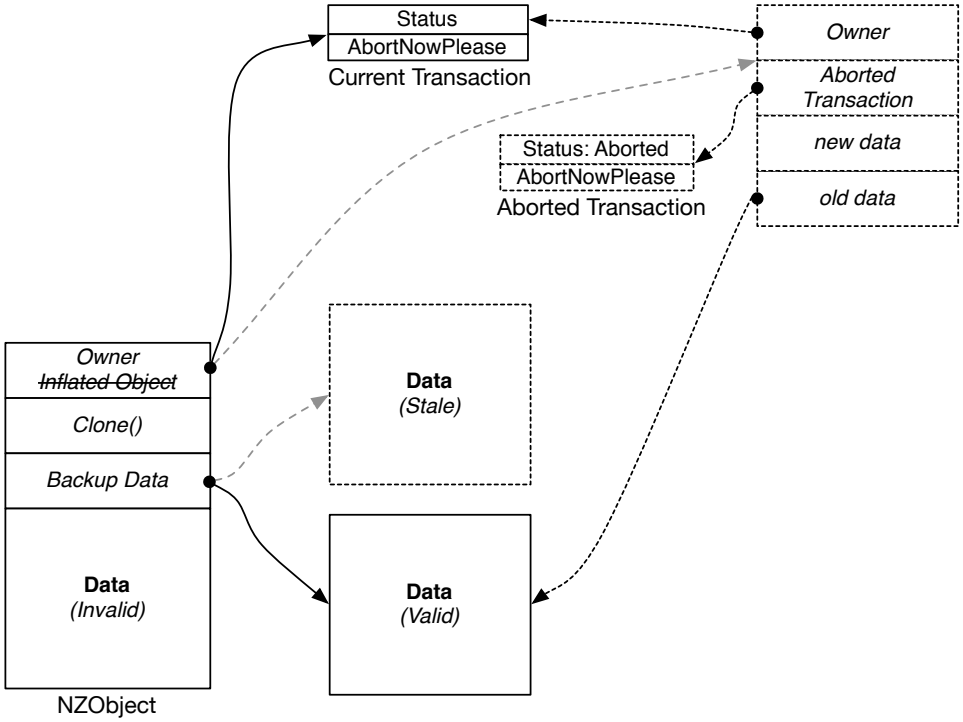

Figure 4.6: *A successfully deflated* `NZObject` *immediately after deflation*

The following simplified code summarizes the nonblocking algorithm. For the interested reader, Appendix A contains a detailed description of the DSTM algorithm, and Appendix B contains a more precise description of NZSTM.

To acquire an object (exclusively)

```
OPEN_WRITE(object)
{
    while(true) {
        Transaction *currentOwner = object->Owner;

        if (currentOwner->Inflated) {
            /*
             * If the object is inflated, treat it as a DSTM object.
             * A description of the DSTM algorithm is available in Appendix A.
             */
            if (open_dstm(object)) {
                break;
            }
        } else if (currentOwner == myThread->CurrentTxn) {
            /* The object is already acquired by this thread's current transaction. */
            break;
        } else if (currentOwner == NULL || currentOwner->Status != Active) {
            /* The object is not currently acquired by an active transaction. */
            atomic {
                /* can be performed using Compare&Swap */
                if (object->Owner == currentOwner) {
                    object->Owner = myThread->CurrentTxn;
                } else {
                    continue;
                }
            }

            if (currentOwner->Status == Aborted && object->BackupData != NULL) {
```

```
                /* Restore the backup copy. */
                object->Clone(object->BackupData, object->Data);
            } else {
                /* Create a new backup copy. */
                object->Clone(object->Data, object->BackupData);
            }

            break;
        } else if (currentOwner->AbortNowPlease) {
            /*
             * The transaction has been asked to abort, wait for it to abort or timeout
             * and inflate the object.
             */
            while(currentOwner->Status != Aborted && !timeout());

            if (currentOwner->Status != Aborted && inflate_object(object)) {
                break;
            }
        } else {
            /* The transaction is active, consult the contention manager. */
            if (contentionManager->shouldAbort(currentOwner)) {
                atomic {
                    /* can be performed using Compare&Swap */
                    if (currentOwner->Status == Active) {
                        currentOwner->AbortNowPlease = true;
                    }
                }
            }
        }
    }

    /* validate */
    if (myThread->CurrentTxn->AbortNowPlease) {
        myThread->CurrentTxn->Status = Aborted;

        /* Use C's longjmp to transfer control and restart. */
        restart_transaction();
    }
}
```

To inflate an object owned by a unresponsive transaction

```
bool inflate_object(object)
{
    Locator *locator = new Locator;
    Transaction *currentOwner = object->Owner;

    locator->Owner = myThread->CurrentTxn;
    if (object->BackupData == NULL) {
        object->Clone(object->Data, locator->Old);
    }
    if (object->BackupData != NULL) {
        locator->Old = object->BackupData;
    }
    object->Clone(locator->Old, locator->New);
    locator->AbortedTxn = object->Owner;

    if (myThread->CurrentTxn->AbortNowPlease) {
        myThread->CurrentTxn->Status = Aborted;

        /* Use C's longjmp to transfer control and restart. */
        restart_transaction();

        /* unreachable */
    }

    if (currentOwner->Status != Active ||
        !currentOwner->AbortNowPlease ||
        object->Owner != currentOwner) {
        return false;
```

```
    }

    atomic {
        /* can be performed using Compare&Swap */
        if (object->Owner == currentOwner) {
            object->Owner = locator;
            object->Inflated = true;
            return true;
        } else {
            return false;
        }
    }
}
```

To deflate an object back into a normal NZSTM object

```
deflate_object(object, locator)
{
    /* called by open_dstm() after the DSTM algorithm has acquired the object */

    void *currentBackup = object->BackupData;

    if (locator->AbortedTxn->Status == Aborted) {
        if (myThread->CurrentTxn->AbortNowPlease) {
            myThread->CurrentTxn->Status = Aborted;

            /* Use C's longjmp to transfer control and restart. */
            restart_transaction();

            /* unreachable */
        }

        atomic {
            /* can be performed using Compare&Swap */
            if (currentBackup == object->BackupData) {
                object->BackupData = locator->New;
            } else {
                return;
            }
        }

        atomic {
            /* can be performed using Compare&Swap */
            if (object->Owner == locator) {
                object->Owner = myThread->CurrentTxn;
                object->Inflated = false;
            } else {
                return;
            }
        }

        object->Clone(object->BackupData, object->Data);
    }
}
```

Using an Atomic Single-Compare Single-Store

The need to wait for an unresponsive transaction in the blocking algorithm presented arises because it is not safe to restore the backup to the `Data` field and continue to access the object in-place: the aborting transaction might write to the `Data` field. A transaction can avoid such *late writes* by atomically pairing each write with a check of the `AbortNowPlease` flag, using an operation called *Single-Compare Single-Store* (SCSS) [Moir, 2005]. This ensures that the write can only happen if the writing transaction has not been asked to abort.

SCSS can be implemented using a short hardware transaction that modifies a location with a new given value only if another location's value matches an expected value. The code listing below demonstrates applying SCSS to atomically pair a write with a check of the `AbortNowPlease` flag.

```
bool SCSS(Transaction *txn, int *data_field, int value)
{
    atomic {
        /*
         * requires hardware support beyond Compare&Swap
         * best-effort hardware transactions with certain guarantees may suffice
         */
        if (txn->AbortNowPlease) {
            return false;
        } else {
            *data_field = value;
            return true;
        }
    }
}
```

A conservative SCSS implementation that sometimes fails, even if the expected value is in the second location, is acceptable; however, if it could fail indefinitely, a software alternative would be needed [Goodman, Moir, Tabba, and Wang, 2009b], which would re-introduce much of the complexity I aim to eliminate by using SCSS. Therefore, this approach assumes an SCSS implementation that makes sufficient guarantees so that a software alternative is not necessary. Rock does not explicitly provide such a guarantee. In practice, all SCSS experiments running on Rock were successful (Section 4.3), despite the lack of a software alternative for the hardware transactions used to implement SCSS.

This simple operation eliminates a great amount of complexity from NZSTM. NZSTM no longer needs code that implements the complete DSTM algorithm (Appendix A) to access inflated objects, no longer needs code for inflating and deflating objects to switch between the two styles, and no longer needs to check whether an object is inflated or not.

Depending on the hardware support, using short hardware transactions, instead of simple writes, could result in at least a small slowdown in the common case. However, the dramatic code simplification this technique offers will often justify a small performance overhead. This observation demonstrates the value of hardware transactional memory supporting even very small and simple low-latency transactions.

4.1.5 Read Sharing in NZSTM

The issue of read sharing is largely independent of the NZSTM algorithm; different read sharing algorithms could work with NZSTM [e.g., Marathe et al., 2006; Herlihy et al., 2003b; Lev and Moir, 2004; Olszewski et al., 2007]. This section describes the read sharing algorithm used in the evaluation. This read sharing method is not necessarily the best to use with NZSTM; but because this issue is tangential, a full investigation of this topic is outside

the scope of this thesis. This read sharing algorithm is inspired by the one used in RSTM [Marathe et al., 2006].

NZSTM uses visible reads, where reader and writer transactions accessing the same object can detect and resolve conflicts with each other. A visible reads algorithm is used because it makes it easier to integrate hardware and software transactions. With visible reads, hardware transactions that need to access an object exclusively can easily identify potential conflicts with software transactions [Damron et al., 2006]. The next chapter elaborates more on this aspect.

This proposed read sharing implementation restructures the `NZObject` as shown in Figure 4.7.

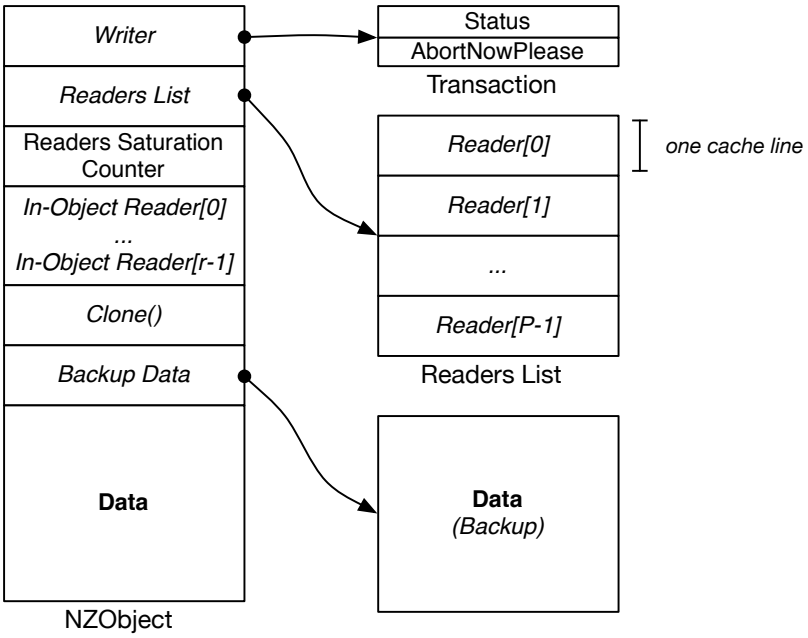

Figure 4.7: *An `NZObject` using the proposed visible readers scheme. For performance reasons, the size of each reader slot in the Readers List is one cache line.*

The restructured `NZObject` contains, in-place, r slots for readers, and a saturation counter that counts the number of occupied slots. When all slots are occupied, the next reader to acquire the object allocates a readers list consisting of P cache lines, where P is the total number of processors in the system. The reader then adds a pointer to its transaction to the cache line whose offset matches the processor number the transaction is running on.

The algorithm for opening an object for read is as follows. When a transaction, T_r, wants to open an object for reading, it checks for active writers that might currently own the object. If an active writer exists, then depending on the contention management policy, T_r either waits for the writer, asks it to abort, or inflates the object if the writer is unresponsive.

Assuming there are no active writers, or that all potential conflicts have been resolved, T_r checks the `Readers Saturation Counter` to determine whether all in-place reader slots have been occupied; this is indicated by the value of the counter being one more than

the number of available slots. If the counter is *not* saturated, T_r briefly acquires the object exclusively, by atomically swapping the `Writer` field to point to its own transaction, adds itself to the first slot not occupied by an active reader, and increments the saturation counter. Before T_r releases the object, it restores the backup data if the previous writer is an aborted transaction and `Backup Data` points to valid data (rather than `NULL`).

If the counter *is* saturated, T_r creates a new readers list, if one does not already exist, by allocating memory for it and atomically setting the `Readers List` field to point to the newly created list. T_r then adds a pointer to its transaction to the appropriate cache line on the readers list, and checks again for conflicts with active writers. If no active writers exist, T_r has succeeded in acquiring the object for read and can now access the data. If there is an active writer, T_r refers to the contention manager to decide how to resolve the conflict.

Before T_r finishes acquiring the object, it validates by checking its own `AbortNowPlease` flag, and aborts itself if the flag is set, relinquishing ownership of the object if necessary. This validation is done to ensure that the set of objects the transaction has opened for read are consistent: if T_r must abort because of a conflict with another transaction on a different object while attempting to open this object for read, then the reads may not be consistent, in which case it is not safe to return to the user code [Larus and Rajwar, 2007; Guerraoui and Kapalka, 2008].

If a transaction, T_w, is opening the object for writing, the procedure is not very different from what it would be assuming there were no read sharing. First, T_w checks and resolves conflicts with any writer transactions. Then it acquires the object by using the `Writer` field in an analogous way to the `Owner` field in the exclusive version. Having acquired the object, T_w cannot proceed until it ensures that no active readers are currently accessing the object. If the `Readers Saturation Counter` indicates that the in-place slots are not saturated, then the writer needs to resolve conflicts only with those readers using the in-place slots. If the slots are saturated, then the writer needs to resolve conflicts with the in-place readers *and* with the whole readers list.

Should a writer transaction decide to abort a reader transaction, then the writer only asks the reader to abort by setting its `AbortNowPlease` flag, and need *not* wait for the reader to acknowledge the abort. The writer can immediately access the object having ensured that all readers are either not active, or have been asked to abort. The reason being that, in this algorithm, every time a reader reads a value from an object, it validates by checking its own transaction status field to ensure it has not been asked to abort before using the read data. This ensures that the reader will use only consistent data without the need to inflate for unresponsive readers, reducing the chances of inflating an object.

Once a writer has resolved all potential conflicts with existing readers, and there are no more active readers accessing the object, the writer resets the `Readers Saturation Counter` to zero. This ensures that future writer transactions do not need to traverse the readers list unless the counter saturates again. This is the only method for resetting the satu-

ration counter in software transactions, because readers do not locally track objects they have acquired for reading, and therefore cannot decrement the counter once they commit or abort.

Displacing data for unresponsive writers is the same as the algorithm described earlier in Section 4.1.4. For simplicity, and because I believe that this scenario is rare, once an object is inflated, it can be accessed only exclusively until it is deflated — no read sharing is allowed for inflated objects.

This treatment of read sharing favors readers over writers, which might be desirable assuming readers significantly outnumber writers. If the number of readers is below the saturation threshold then the overhead writers incur is low. This is because, to resolve all conflicts, writers need to access only the cache lines that contain the object itself and its metadata, and do not need to traverse a readers list that resides on other cache lines.

4.1.6 Contention Management in NZSTM

NZSTM uses an out-of-band contention manager in the same manner as other software transactional memory proposals (e.g., DSTM and RSTM). NZSTM invokes the contention manager when a transaction tries to acquire an object in a manner that conflicts with the current object's readers or writer. The contention manager decides which transaction has priority, and based on that, it instructs the transaction that invoked it either to wait or to ask the other transaction to abort.

Having the contention manager as a separate out-of-band module makes it easier to apply different policies to a transactional system. It also makes the system more robust, because the contention manager's decision affects only the system's performance, not its correctness. Even if a transaction were to ignore the contention manager's decision, the system would still be correct.

Contention management, and how priority is determined, has been widely covered in other work [e.g., Herlihy et al., 2003b; Scherer and Scott, 2005; Spear, Dalessandro, Marathe, and Scott, 2009]. Two contention management policies, relevant to the work presented in this thesis, are the following [Scherer and Scott, 2005].

Karma: This is a priority-based policy, where transactions abort only lower priority transactions they conflict with; otherwise, they wait. Each transaction's priority is proportional to the number of objects it has opened in the transaction so far. A transaction resets its priority to zero when it commits, but keeps the priority it has accumulated if it aborts.

Timestamp: This is also a priority-based policy. Each transaction's priority is proportional to the age of the transaction. A transaction's age can be determined using different methods, such as the system time at the start of a transaction, a Lamport clock [Lamport, 1978], or the total number of transactions the thread has executed so far. Variations to the policy can also be achieved by giving threads the right to modify their timestamps in controlled ways.

I also developed a new contention management policy, *KarmaDD*, which is a variant of Karma combined with deadlock detection. In KarmaDD, each transaction's priority is proportional to the number of objects it has already acquired in *this* transaction attempt; priority does not accumulate and is reset every time a transaction commits or aborts. When a conflict is detected, a transaction is not aborted, even if it is a low priority transaction, unless a deadlock has been detected or a timeout is triggered (to ensure progress).

KarmaDD's deadlock detection is modeled on the scheme Moore et al. [2006] use for their hardware transactional memory proposal. Whenever a transaction, T_L, detects a conflict with a higher priority transaction, T_H, T_L raises a flag and waits until T_H commits or aborts. When a transaction, T_H, detects a conflict with a lower priority transaction, T_L, whose flag is raised, T_H infers that there is a potential cycle and requests that T_L abort.

In my experiments, KarmaDD's performance was consistently better than the other policies. Therefore, it is the default policy used for the evaluation.

4.1.7 Memory Management in NZSTM

The NZSTM algorithm allocates new memory at runtime to manage its metadata. Manual memory management is difficult in parallel algorithms that allocate and share new memory dynamically at runtime with an undetermined number of threads [Marathe et al., 2006; Hudson, Saha, Adl-Tabatabai, and Hertzberg, 2006]. Without additional communication between threads, each individual thread cannot determine when it is safe to deallocate an object, because it cannot be certain whether other threads might still be accessing that object. This is why many parallel programs and algorithms rely on reference counting and automatic garbage collection.

NZSTM is written in C, a language that does not support garbage collection by default. The issue of memory management is outside the scope of my research, and ideally, should not affect the design. Therefore, this work uses the *Boehm-Demers-Weiser conservative garbage collector* [Boehm, Demers, and Weiser, 2010].

The Boehm-Demers-Weiser garbage collector is available as a multi-platform open-source library for C and C++. It is competitive with manual memory management for single-threaded workloads. It is also scalable, and outperforms manual memory management, at least for some multithreaded workloads [Boehm et al., 2010].

4.2 Correctness Evaluation

In developing NZSTM, I have used model checking to increase confidence in the correctness of the NZSTM algorithm. I have also used stress tests, forced inflations in NZSTM, which rarely occurred in the performance evaluation, coupled with assertions and sanity checks. This section discusses these aspects of the correctness evaluation.

4.2.1 Formal Verification and Model Checking

> *One can never guarantee that a proof is correct, the best one can say is "I have not discovered any mistakes."*
>
> *— Edsger W. Dijkstra*

I created a model of the NZSTM algorithm in *Promela*, and mechanically checked various useful properties of it using *Spin* 5.2.5 [Holzmann, 2003]. Model checking was helpful in developing the algorithm, resolving some subtle bugs that did not manifest in normal testing, and increasing confidence.

Promela is a C-like language that can describe models to check using Spin. Spin is an open-source mechanical verification system developed at Bell Labs and designed for the formal verification of high-level models for systems of concurrently executing processes. Spin can perform exhaustive searches of all possible executions of a given model while applying sanity checks and assertions. Spin can also find unreachable code, deadlock, and cycles (livelock) in a model.

If Spin finds an error in the model, it generates a trace file that shows the exact steps needed to reproduce the error. Spin can also find the minimum number of steps to reach an error. This aspect is very helpful in debugging non-deterministic concurrent systems, where concurrency errors are hard to reproduce, and where attaching a debugger often hides such errors.

Because of the complexity of transactional memory algorithms and the amount of state they require to model, it is not generally feasible to establish their correctness using model checking. Any additional complexity in a model, such as modeling additional variables or threads, expands the state-space that needs to be checked *exponentially*. Therefore, these tools only check the correctness of limited cases in order to increase confidence in the algorithms. Ananian and Rinard [2005] and O'Leary, Saha, and Tuttle [2009] have also used Spin for model checking different properties of transactional memory proposals.

Using Spin, I performed complete state-space searches for up to three concurrent threads, each thread accessing up to three objects for either writing or reading using the read sharing algorithm described. Increasing the number of thread to four increases the search space so that complete searches were not possible because of memory limitations. Instead, verification using the non-exhaustive *bitstate hashing* [Holzmann, 2003] was performed for up to four threads.

Spin's unreachable code reporting, deadlock detection, and cycle detection were used. For the models tested, all code paths were taken at least once, no deadlock occurred, and no cycles (livelock) occurred. NZSTM is only obstruction-free and not lock-free, and thus livelock *is* possible; it is the role of the contention manager to ensure this does not happen in practice. Spin reports that the model is livelock-free because a retried transaction allocates a new `Transaction` object, so even when the algorithm exhibits livelock, no state is repeated.

I verified the models with Spin running on an eight core Niagara UltraSPARC machine, each core running at 1167 MHz, with 32 GiB of memory for the whole machine. Below is a description of one of the simpler tests, along with a representative sample of its model checking cost.

This particular test models a memory with two integer objects initialized to zero. Each transactional thread increments then decrements an integer object by one, followed by reading the other integer and asserting that it is zero. Spin exhaustively checks every possible interleaving of these operations and every possible combination of which integer to read and which one to modify. In addition to checking invariants related to the transactional model, Spin also checks that, at the end of a transaction, the value of all integer objects remains zero.

Model checking NZSTM with read sharing required 49,835 states for two threads, and 60,664,348 states for three threads. On the other hand, checking the blocking version of NZSTM, with read sharing, required 13,412 (74% fewer) states for two threads, and 1,624,991 (97% fewer) states for three threads.

Model checking NZSTM using SCSS requires virtually the same number of states as the *blocking* algorithm. SCSS is an atomic operation, which Spin models as a single-step operation, and therefore does not expand the state-space. This highlights the amount of complexity in a nonblocking algorithm that even short hardware transactions could reduce.

For the interested reader, the Promela model for NZSTM is available in Appendix B.

4.2.2 Runtime Correctness Evaluation

> *Program testing can be used to show the presence of bugs, but never to show their absence!*
>
> *— Edsger W. Dijkstra*

To increase confidence in the correctness of NZSTM, I used stress tests and runtime assertion checking [Hoare, 1969, 2002]. Because the performance evaluation shows that object inflation rarely occurs in NZSTM, I also created tests that induce object inflation.

The stress tests involve running all the available benchmarks, described in the next section, multiple times. The version of NZSTM in the stress tests includes assertions that ensure NZSTM's data structures maintain the algorithm's invariants at different points of its execution. For example, every time a transaction accesses object data, it asserts that the transaction's status field is `Active`. If any of the assertions fail, the program halts with an error message outlining which assertion failure was the cause.

In addition to assertions, most of the benchmarks have an integrity test associated with them that runs after the benchmark has finished. This test ensures that a benchmark's data structures maintain the invariants of that particular benchmark. For example, the integrity test for a benchmark based on a sorted linked list of integers, whose values must be from 0

to 255, would traverse the list, checking that the value of each node is greater than that of its preceding node, that the smallest node's value is 0 or more, and that the largest node's value is 255 or less.

All performance evaluations and stress tests run these integrity tests at the end. Because the integrity test runs *after* the benchmark time has been measured, it does not affect the reported performance of the benchmark, and helps gain confidence in the correctness of the algorithm and of the running benchmark code. However, performance evaluation disables all assertions and sanity checks that occur *during* the runtime of the benchmark, because these assertions affect the reported performance.

To test the nonblocking algorithm, object inflation was induced by modifying NZSTM so each thread would pick a random transaction, on average one in every one hundred transactions, and force it to sleep for a duration long enough for other threads to assume it is unresponsive. I ran these tests, with the assertions mentioned above and with the integrity checks at the end, using different benchmarks.

The techniques used to evaluate the correctness of NZSTM are neither definitive nor conclusive. They were, however, very useful in designing and debugging the algorithms.

4.3 Performance Evaluation

This section reports on NZSTM's performance evaluation using a prototype of Sun's Rock processor (Section 2.8).

4.3.1 Benchmarks

This evaluation uses microbenchmarks and STAMP benchmarks with varying workloads.

The microbenchmarks, adapted from the Java-based DSTM, are the following. The `linkedlist` benchmark is a concurrent set implemented using a single sorted linked list. Each thread randomly chooses to insert, delete, or look up a value in the range of 0 to 255, with the *high* contention distribution of operations being 1:1:1 (insert:delete:lookup) and the *low* contention distribution of operations being 1:1:8. The `redblack` and `hashtable` benchmarks are also concurrent sets, implemented using a red-black tree and a chained hashtable.

I also use the `kmeans`, `vacation`, and `genome` STAMP benchmarks, with the same parameters used by their authors [Minh, Trautmann, Chung, McDonald, Bronson, Casper, Kozyrakis, and Olukotun, 2007]. These parameters are shown in Table 4.1.

Table 4.2 presents a qualitative summary, relative to the STAMP benchmarks, of the benchmarks' runtime transactional characteristics: length of transactions (number of instructions), time spent in transactions (relative to the total benchmark runtime), size of the transactions' read and write sets, and amount of contention.

Table 4.1: *STAMP parameters used in the evaluation*

Benchmark	Parameters
`kmeans-high`	`-m15 -n15 -t0.05 -i random1000_12`
`kmeans-low`	`-m40 -n40 -t0.05 -i random1000_12`
`vacation-high`	`-n8 -q10 -u80 -r65536 -t4096`
`vacation-low`	`-n4 -q90 -u80 -r65536 -t4096`
`genome`	`-g256 -s16 -n16384`

Table 4.2: *Qualitative summary, relative to STAMP, of the benchmarks' runtime transactional characteristics. [cf. Minh et al., 2008]*

Benchmark	Transaction Length	Transaction Time	Read/Write Set Size	Contention
`linkedlist`	medium	high	medium	high
`redblack`	short	high	medium	low
`hashtable`	very short	high	small	low
`kmeans`	short	low	small	low
`vacation`	medium	high	medium	low/medium
`genome`	medium	high	medium	low

4.3.2 Experiments

The benchmarks are tested using 1, 2, 4, 8, and 16 threads on the Rock machine, each thread running on a separate core. The benchmarks are compiled using GCC 3.4.6, with optimization set to level 3. The benchmarks begin by initializing the relevant data structures, and then start taking measurements, recording the elapsed wall clock time to complete. Each benchmark is run three times, and the average time of the three runs is recorded.

I evaluate the following proposals for executing transactions.

NZSTM: The system assumes no special hardware support and runs using NZSTM software transactions with visible reads. Visible reads have one reader's slot in-place in the object ($r = 1$), and allocate as many cache lines (P) for the reader's list as the number of running threads.

BZSTM: The system assumes no special hardware support, and runs using *blocking* NZSTM software transactions. Objects never get inflated, so it is not necessary to check if the objects are inflated. I compare against this system to evaluate the overhead that checking for inflated objects incurs.

DSTM2-SF: The system assumes no special hardware support and runs using DSTM2-SF software transactions. I compare against this system because it is a blocking object-based software transactional memory designed from the ground up as a blocking algorithm. In addition, Herlihy et al. [2006] show that DSTM2-SF significantly outperforms the nonblocking DSTM. This DSTM2-SF implementation is a good-faith reproduction, which uses the same visible reads and contention management policy as NZSTM.

SCSS: The system runs using NZSTM software transactions, and assumes hardware with sufficient guarantees to support an SCSS implementation, which obviates the need for inflating objects.

4.3.3 Results

Because of the careful attention to cache performance in NZSTM, and because I believe that unresponsive transactions — and thereby inflation — are rare, I expect NZSTM's performance to be similar to the other proposals evaluated. This includes DSTM2-SF, which was designed to be blocking specifically for performance reasons. However, I expect that NZSTM would incur some overhead, because of the additional checking to ensure that an object is not inflated. This overhead should be small, especially because Rock's aggressive speculation would hide much of the checking latency.

Figure 4.8 shows the speedup of running the benchmarks using the different proposals, relative to a single global lock (not shown) running on a single processor. The data is normalized to a single global lock because it demonstrates the performance achievable on systems that have no transactional memory support with a similar level of programming complexity as using transactions.

The different proposals performed mostly within 10% of each other, except in the write-dominated `kmeans`. In most cases, NZSTM lags slightly behind BZSTM, by about 2–5%. This is due to overhead for checking for inflated objects; it is *not* due to any actual object inflation, which did not occur in these experiments.

SCSS was virtually identical with NZSTM in all benchmarks, except for `kmeans`. Because `kmeans` is a write-dominated benchmark, the overhead of wrapping every store with a short hardware transaction significantly impacts its performance.

The nonblocking NZSTM is competitive with the blocking DSTM2-SF, sometimes lagging behind it slightly, and at other times outperforming it slightly. In `kmeans`, NZSTM significantly outperforms DSTM2-SF. This is likely because of the additional space overhead in DSTM-SF. The size of the main transactional object in `kmeans`, not including metadata, is 100 bytes. This requires two cache lines on Rock, whose cache line size is 64 bytes. By contrast, objects in the other benchmarks easily fit on a single cache line. Because DSTM2-SF allocates space for backup data in-place with an object, it requires four cache lines to store a `kmeans` transactional object; whereas NZSTM requires only two cache lines for the same object. Therefore, DSTM2-SF can incur twice as many cache misses when accessing a `kmeans` object. NZSTM uses thread-local memory for backups, which it reuses after successful transactions, thus improving cache locality.

Compared with using a single global lock, all proposals in this evaluation are significantly slower when running on a single thread, ranging from being twice as slow to being ten times as slow. The benchmark with the smallest gap is `kmeans`, where NZSTM is slower only by

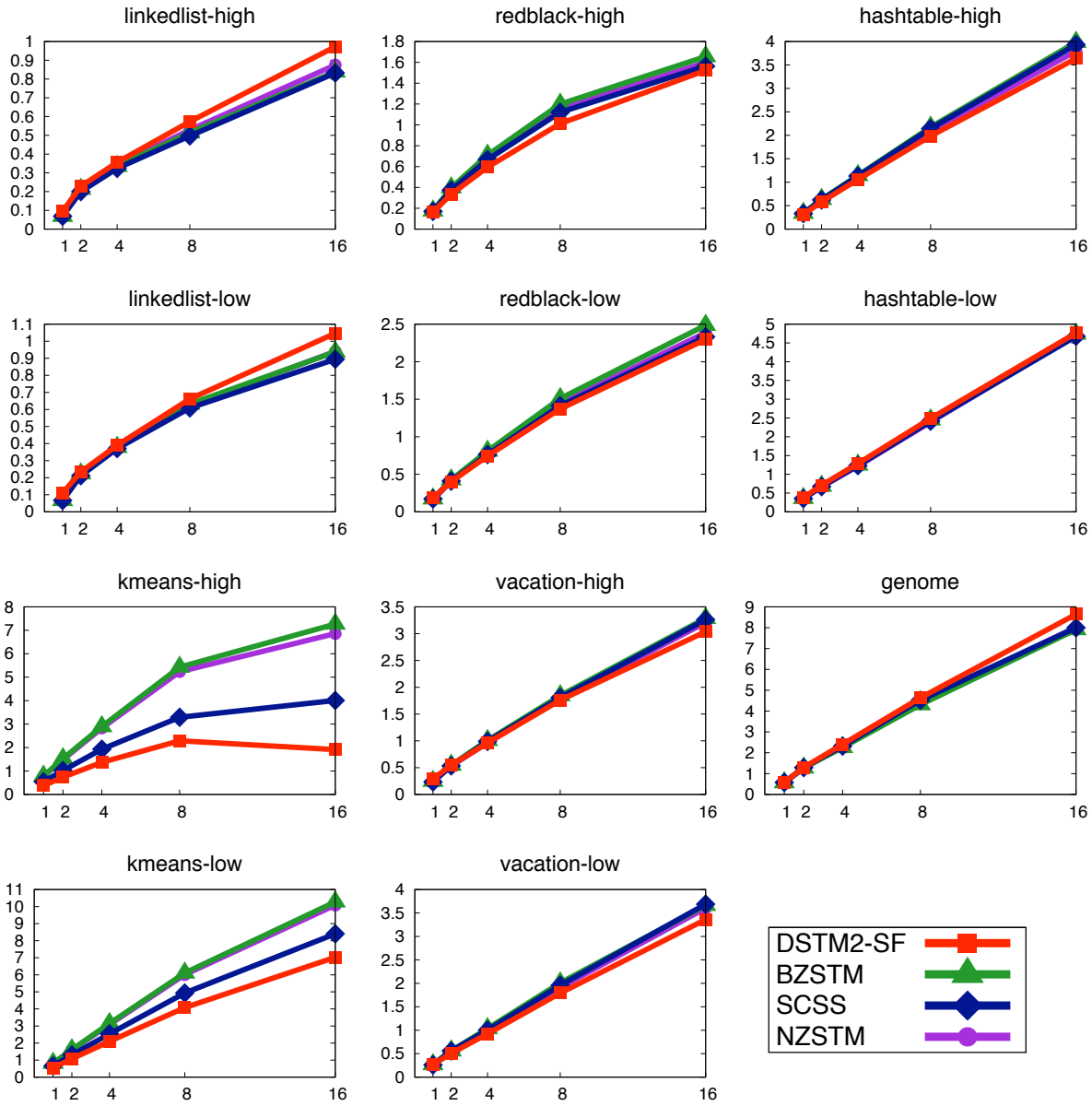

Figure 4.8: *Results of running the benchmarks on the Rock machine. The x-axis represents the number of threads, and the y-axis represents the speedup relative to a single thread using a single global lock (not shown).*

about 20% in the single-threaded case. The gap is small in the `kmeans` benchmark because its transactional component is small; it spends less than 10% of its time inside a transaction.

4.3.4 Discussion

The results show that nonblocking software transactional memory algorithms are not inherently slower than their blocking counterpart. In particular, when comparing a blocking version against a nonblocking version of the same base algorithm (BZSTM against NZSTM),

the difference is barely noticeable in most cases. This might be because Rock's aggressive speculation hides much of the latency of NZSTM's checking if objects are inflated.

Furthermore, when comparing an algorithm designed as a blocking algorithm (DSTM2-SF) against a nonblocking algorithm (NZSTM), other considerations, namely how the algorithm handles backup copies of objects acquired exclusively, can have a bigger impact on performance than whether the algorithm is blocking or nonblocking.

The results also show that, for a small number of threads, none of these proposals are competitive with using a single global lock. For most benchmarks, software transactional memory performance breaks even with using a single global lock on a single thread after running on four threads. For the `linkedlist` benchmark, which has high contention, these proposals break even with a single global lock when running on sixteen threads. However, the performance of all proposals tested scales as the number of threads increases, unlike a single global lock, which does not scale without hardware support.

Finally, the results demonstrate the value of hardware support for transactions — even if those transactions are small. The SCSS-based NZSTM algorithm performed mostly as well as the blocking algorithms, while providing nonblocking progress guarantees and being almost as simple as its blocking counterpart.

4.4 Concluding Remarks

Blocking is unacceptable in some cases, but previous nonblocking object-based software transactional memory proposals use expensive indirection, resulting in worse performance than their blocking counterpart. This chapter introduced NZSTM, a nonblocking object-based software transactional memory that eliminates indirection, in the common case, resulting in performance competitive with blocking proposals.

NZSTM demonstrates that nonblocking software transactional memory proposals are not inherently slower than blocking ones. Moreover, with the support of only small, simple, hardware transactions, nonblocking proposals do not need to be significantly more complicated than blocking ones either.

The next chapter demonstrates how NZSTM is used in a hybrid transactional memory context, which exploits best-effort hardware transactional memory.

Chapter 5

Hybrid Nonblocking Zero-indirection Transactional Memory

This chapter presents *Nonblocking Zero-indirection Transactional Memory* (NZTM). NZTM is a nonblocking hybrid transactional memory, which comprises a software component based on NZSTM, and can exploit best-effort hardware support to improve performance, when available. The purpose of using NZSTM as the software component for NZTM is to make the hardware case fast, by not requiring it to go through indirection to access the data in the common case.

The earliest hybrid proposals are based on software transactional memory proposals that, unlike NZSTM, do not collocate the object data with its transactional metadata. For example, HyTM [Damron et al., 2006] uses a word-based software transactional memory that has a separate ownership record table. Both hardware and software transactions must consult this table before they access the data. Another early hybrid proposal is the one by Kumar et al. [2006], which uses DSTM as its software component. As explained in the previous chapter, DSTM requires two level of indirection to access the data.

This method of data organization means that the object data and its metadata could be located on two or more different cache lines, possibly causing a transaction to incur a cache miss for every location it accesses. This additional overhead is one of the reasons why software transactional memory is significantly slower than hardware transactional memory. If a hybrid transactional memory uses a software component that does not collocate an object's metadata with its data, then hardware transactions, which hybrid proposals target as the common case, incur this additional overhead as well.

A possible solution to this problem is to use a blocking software component that collocates an object's metadata with its data. As mentioned previously, blocking algorithms introduce a different set of problems and are undesirable in certain contexts.

Consequently, the main motivation in designing NZTM is to ensure that hardware transactions incur as little overhead as possible. NZTM should also be nonblocking, particularly

because I do not believe that nonblocking algorithms are inherently slower than blocking ones, nor are they more complex in the presence of hardware support.

In NZTM, transactions can run using best-effort hardware support, and fall back on NZSTM software transactions when they abort. NZTM is optimized for the case in which hardware support is available and able to commit most transactions. An object's metadata is collocated with the object data in the common case, and therefore, NZTM achieves near-optimal cache behavior in the common case.

Ananian and Rinard [2005] also propose a nonblocking hybrid transactional memory that eliminates indirection for hardware transactions in the common case. However, software transactions that perform writes require indirection in all cases, which hardware transaction that follow have to check for and handle.

The main contributions presented in this chapter are the following.

- I describe NZTM, a hybrid transactional memory based on NZSTM. NZSTM is particularly suited for a hybrid system because it requires no indirection to access an object's data in the common case.

- I present a performance evaluation of NZTM with best-effort hardware support using Sun's ATMTP simulator. NZTM is compared against LogTM-SE [Yen et al., 2007], an unbounded hardware transactional memory. The results show that NZTM is competitive with LogTM-SE.

- I also present a performance evaluation of NZTM on Rock using Rock's best-effort hardware support. The results show that NZTM running on Rock performs well on smaller workloads. However, because of the limitations on the type of transactional load Rock can run, some of the larger workloads do not benefit from Rock's hardware support.

The remainder of this chapter is organized as follows. Section 5.1 describes the design of NZTM. Section 5.2 presents a performance evaluation of NZTM using a simulator and the Rock machine. Finally, Section 5.3 concludes this chapter.

5.1 The Design of NZTM

The main goal in the design of NZTM is a nonblocking object-based hybrid transactional memory that is fast and competitive with unbounded hardware transactional memory. NZTM is designed to eliminate the costly indirection, and to commit as many transactions as possible using the fast hardware path.

NZTM is written in the C programming language, and uses the same NZSTM programming model and data structures described in the previous chapter. To simplify the discussion,

this section first describes NZTM based on the NZSTM algorithm that does not allow read sharing between software transactions. NZTM can work with different software read sharing algorithms with little modification; therefore, this section then describes the modifications made so NZTM can support the visible read sharing algorithm used in the previous chapter.

5.1.1 The Basic NZTM Algorithm

In a similar manner to HyTM [Damron et al., 2006], NZTM attempts transactions using hardware support and if (repeatedly) unsuccessful, runs transactions using NZSTM software support. The decision of when to give up on attempting a transaction in hardware is a matter of policy, which could be determined by the type of workload that is running, the limitations of the underlying hardware, and the reason the transaction aborted.

When an NZTM hardware transaction acquires an object, it checks for conflicts with software transactions, and explicitly aborts itself if it discovers any; it can then try again either in hardware or in software, depending on the policy. If the transaction does not discover any conflicts, it can safely proceed, because a subsequent conflict with a software transaction will modify data that the hardware transaction has accessed, thereby automatically aborting the hardware transaction. Hardware transactions automatically abort in this case because NZTM's best-effort hardware leverages the underlying cache coherence protocol for conflict detection (Section 2.5).

A variety of approaches of checking for conflicts with software transactions are possible. Conservative approaches are simpler, but more likely to revert to software, which hurts performance. In the simplest and most conservative scheme, a hardware transaction accessing an `NZObject` (Figure 5.1) aborts itself if the `Owner` field is non-`NULL`, and proceeds otherwise. This scheme is conservative because if the `Owner` field is non-`NULL`, but points to an aborted or a committed transaction, there is no conflict.

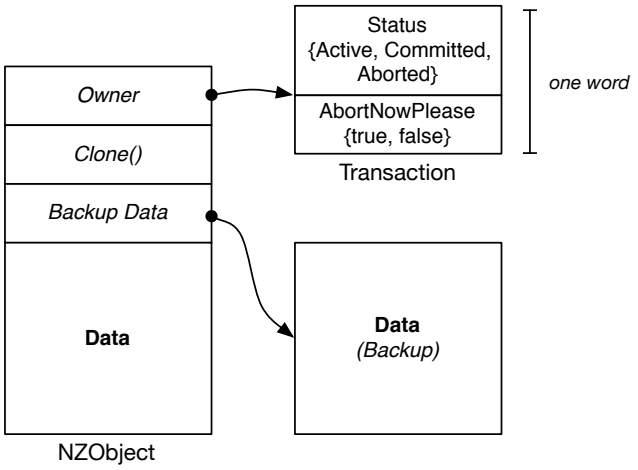

Figure 5.1: *The structure of NZTM's main transactional object*

In NZTM, hardware transactions check in more detail to determine if there is a conflict, by inspecting the status of the previous software owner transaction, identified by the `Owner` field. If the previous software owner is a committed transaction, the hardware transaction accesses the data directly. If the previous software owner is an aborted transaction, the hardware transaction first restores the backup data, and then accesses the data in-place. If the object is inflated, i.e., the `Owner` field's low-order bit is set, the hardware transaction first deflates the object, and then accesses the data in-place.

Once a hardware transaction has ensured that there are no conflicts and that the object data is in-place, it sets the object's `Owner` field to `NULL`. This eliminates the need for subsequent hardware transactions to perform similar checks.

Although it is possible for hardware transactions to access the backup data or even the inflated data directly, NZTM attempts to restore the object to its original in-place state. This eliminates indirection for data accesses by future hardware transactions, and also by the current transaction if it accesses the same object later.

Hardware transactions abort themselves if they discover a conflict with a software transaction. Without hardware support beyond best-effort, hardware transactions are unable to communicate with software transactions or a software contention manager to try to resolve the conflict. Such communication typically involves modifying shared memory; however, hardware transactions are designed to preserve isolation, and are unable to make such modifications visible inside a transaction. In a hybrid system such as NZTM, software transactions are likely to be transactions that were attempted in hardware first, and used software as a fallback mechanism after aborting. Therefore, software transactions are likely to have higher priority than hardware transactions, which makes aborting hardware transactions in favor of software a good policy anyway.

It is important to note that NZTM performs these checks by controlling what *code* is executed within a hardware transaction, *not* by assuming any special support in the hardware beyond best-effort.

The algorithm described so far does not support read sharing between hardware and software transactions. Because software transactions must acquire the `Owner` field of an object, regardless of whether they are modifying the object or only reading it, neither hardware nor software transactions can differentiate between software readers and software writers. However, this algorithm does allow read sharing between *hardware* transactions, because hardware transactions do not modify any data structures to acquire an object, but rely instead on the underlying coherence protocol to detect and resolve conflicts. Cache coherence protocols typically allow read sharing, and so do best-effort hardware transactional memory proposals that leverage them.

5.1.2 Read Sharing between Hardware and Software Transactions

For many workloads, accessing objects only for reading is more common than accessing them for writing inside a transaction [Minh et al., 2008]. It is therefore desirable that hardware and software readers are able to coexist without conflicts. Otherwise, even a few software transactions could trigger many conflicts with hardware transactions causing the system to constantly fall back on software.

This section now describes how NZTM allows read sharing between hardware transactions and software transactions that use the visible read sharing algorithm described in Section 4.1.5.

A hardware transaction opening an NZObject (Figure 5.2) for reading needs to check only for conflicts with a software writer, because hardware and software readers are not in conflict. When a hardware reader acquires an object, it checks that the object's `Writer` field does not point to an active software transaction, in which case the hardware transaction has to abort and try again. If the `Writer` field points to a committed or an aborted transaction, the hardware transaction can proceed, first restoring any backup data in case of an aborted software transaction, and setting the `Writer` field to `NULL`, which eliminates the need for subsequent hardware transactions to check the state of the `Writer`.

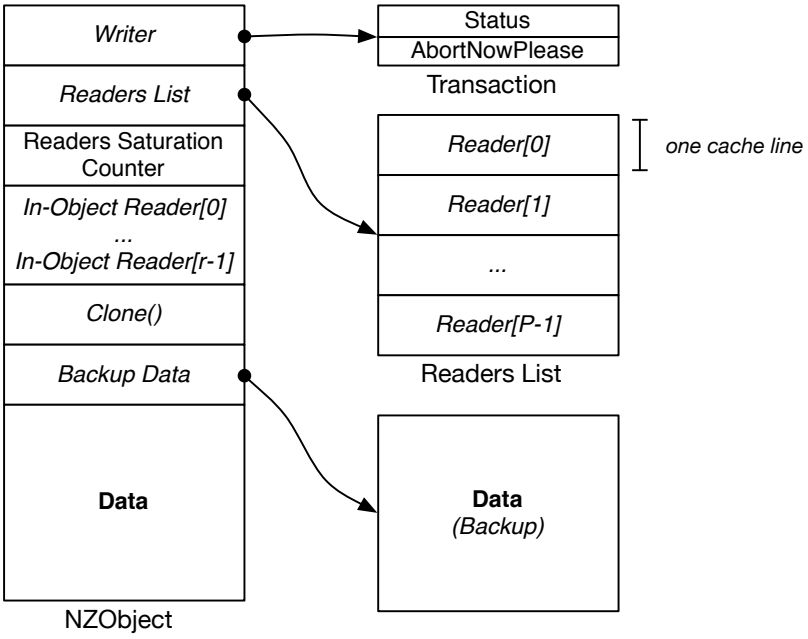

Figure 5.2: *An `NZObject` using the proposed visible readers scheme*

A hardware transaction opening an object for writing must check for conflicts with both software readers and writers. When a hardware writer acquires an object, it performs the same check on the `Writer` as above. If successful, it first ensures there are no conflicts with software readers by checking the `Readers Saturation Counter`. If the counter is zero, the hardware writer can proceed because it knows that there are no software readers. If the

counter is greater than zero, the hardware transaction must check for the existence of software readers in the in-place reader's slots. The hardware transaction also checks the reader's list if the `Readers Saturation Counter` is saturated. Once the hardware transaction has ensured that there are no active software readers, it resets the `Readers Saturation Counter` to zero, which eliminates the need for subsequent hardware transactions to perform similar checks.

5.1.3 Policy on Retrying Hardware Transactions

One of the design decisions that affect the performance of a hybrid system is whether to try a transaction again in hardware, if it aborts, or whether to fall back on software.

If the transaction tries again in hardware, then it might still commit successfully in hardware, benefiting from the speed of hardware transactions. On the other hand, if the transaction aborts because of a limitation in the hardware, then another attempt is wasted work. In that case, it is desirable that the transaction should fall back on software immediately.

If the transaction falls back on software, it is more likely to commit successfully, particularly if the hardware transaction aborted because of a hardware limitation. Software transactions are slower than hardware transactions, and the goal of NZTM it to improve performance by using the hardware path as often as possible. Therefore, falling back on software prematurely could result in a system that runs mainly in software, wasting the opportunities hardware support provides.

As with many other policy decisions, there is no policy that perfectly suits all workloads. The best policy depends on the current workload the system is running, as well as the limitations of the underlying hardware. Different best-effort hardware transactional memory implementations might have different types of limitations. However, because best-effort proposals typically leverage the cache coherence protocol, they are likely to share many of those limitations.

In their evaluation of the Rock processor, Dice et al. [2009b] presented a detailed analysis of the different reasons that might abort a hardware transaction in Rock. They also presented different recommendations on what to do based on each reason for aborting, which the CPS register indicates (Section 2.8). Below is a summary of the reasons why a hardware transaction might abort, and what the best course of action might be. This summary is mainly based on the findings of Dice et al., as well as my experiences using ATMTP and Rock.

Fundamental limitations: Some limitations in Rock are fundamental; unless the aborted transaction that runs again takes a different code path, it will never succeed in hardware. These limitations include running out of space in the write buffer or cache memory, attempting to execute an unsupported instruction, and generating an exception. NZTM's policy in this case is to fall back on software immediately.

Coherence conflicts: When a cache line that has been accessed by a Rock transaction is invalidated, the transaction aborts. This invalidation could be caused by a conflict with another hardware transaction or with software (transactional or non-transactional). NZTM's policy in this case is to try again in hardware a few more times, and then fall back on software.

Conflicts with software transactions: When an NZTM hardware transaction acquires an object, it aborts itself if it finds that the object is owned by a software transaction. Rock's CPS register does not distinguish between aborts caused by unsupported instructions and deliberate aborts, because the mechanism NZTM uses to explicitly abort a hardware transaction is by attempting to execute an unsupported instruction. The code I run does not use unsupported instructions unless it is explicitly aborting a transaction; therefore, aborts caused by an unsupported instruction imply a conflict with a software transaction. NZTM's policy in this case is to try again in hardware a few more times, and then fall back on software.

Transient limitations: Some events that abort a hardware transaction will not necessarily reoccur if the transaction is retried. These events include mis-speculation, interrupts, and TLB misses. NZTM's policy in this case is to try again in hardware a few more times, and then fall back on software.

In the Rock-like ATMTP simulator, unlike the actual Rock processor, all transient limitations are fundamental. Because ATMTP completely rolls back the state of the aborted processor, events such as TLB misses will always reoccur if the hardware transaction accesses the same memory location again when retried. Therefore, when running on ATMTP, NZTM immediately falls back on software if the transaction was aborted by such events. On the other hand, because ATMTP does not perform aggressive speculation, aborts caused by mis-speculation do not happen.

Based on my experiences using ATMTP and Rock, unless a hardware transaction aborts because of a fundamental limitation, the number of hardware transaction attempts should be proportional to the number of threads running in the system. Specifically, if a hardware transaction aborts because of a conflict with another transaction, it tries again a number of times equal to the number of threads running in the system. If it aborts because of a transient limitation, such as an interrupt, it tries again three more times when running on Rock, but falls back on software immediately when running on ATMTP.

Table 5.1 summarizes how NZTM behaves when a hardware transaction aborts, both on Rock and on ATMTP.

Table 5.1: *Course of action when a hardware NZTM transaction aborts on Rock and on ATMTP*

Cause	Rock	ATMTP
Transient limitation	try again three times	fall back on software
Fundamental limitation	fall back on software	
Conflict (coherence or software)	try again a number of times equal to the number of running threads	

5.2 Performance Evaluation

This section reports on the performance evaluation of NZTM.

5.2.1 Experiment Environments

I evaluate NZTM using Sun's Rock processor and the ATMTP simulation framework, both described in Section 2.8.

The simulation framework is based on Virtutech Simics 3.0.30, in conjunction with the University of Wisconsin GEMS 2.1 memory models. The simulator models processors that have best-effort hardware support, using Sun's Rock-like ATMTP simulator. It can also model *LogTM-SE* [Yen et al., 2007], an unbounded hardware transactional memory.

The simulation parameters are shown in Table 5.2.

Table 5.2: *Simulated machine configuration*

Item	Model
Processor	in-order, single-issue, single-threaded, multicore
Cache line size	64 bytes
L1 cache	256 KiB, 4-way set-associative, 1 cycle latency
L2 cache	4 MiB, 8-way set-associative, 20 cycle latency
Physical Memory	8 GiB, 450 cycle latency
Processor Network Topology	point to point
ATMTP Specific	
Transactional cache	L1 cache
Transactional write buffer size	256 entries
Conflict resolution policy	requester-wins
Function calls in transactions	allowed
LogTM-SE Specific	
Conflict resolution policy	timestamp with deadlock detection

The ATMTP parameters are set to be bigger and more tolerant to certain events than Rock, e.g., by allowing function calls inside transactions. When using the simulator, the goal is to evaluate how well NZTM could perform using best-effort support in general, rather than how the limitations of a particular implementation of best-effort hardware (e.g., Rock) could affect it.

5.2.2 Benchmarks

This evaluation uses microbenchmarks and STAMP benchmarks with varying workloads. These are the same workloads used for evaluating NZSTM (Section 4.3).

The microbenchmarks, adapted from the Java-based DSTM, are the following. The `linkedlist` benchmark is a concurrent set implemented using a single sorted linked list. Each thread randomly chooses to insert, delete, or look up a value in the range of 0 to 255, with the *high* contention distribution of operations being 1:1:1 (insert:delete:lookup) and the *low* contention distribution of operations being 1:1:8. The `redblack` and `hashtable` benchmarks are also concurrent sets, implemented using a red-black tree and a chained hash-table.

I also use the `kmeans`, `vacation`, and `genome` STAMP benchmarks, with the same parameters used by their authors [Minh et al., 2007]. These parameters are shown in Table 5.3.

Table 5.3: *STAMP parameters used in the evaluation*

Benchmark	Parameters
`kmeans-high`	`-m15 -n15 -t0.05 -i random1000_12`
`kmeans-low`	`-m40 -n40 -t0.05 -i random1000_12`
`vacation-high`	`-n8 -q10 -u80 -r65536 -t4096`
`vacation-low`	`-n4 -q90 -u80 -r65536 -t4096`
`genome`	`-g256 -s16 -n16384`

Table 5.4 presents a qualitative summary, relative to the STAMP benchmarks, of the benchmarks' runtime transactional characteristics: length of transactions (number of instructions), time spent in transactions (relative to the total benchmark runtime), size of the transactions' read and write sets, and amount of contention.

Table 5.4: *Qualitative summary, relative to STAMP, of the benchmarks' runtime transactional characteristics. [cf. Minh et al., 2008]*

Benchmark	Transaction Length	Transaction Time	Read/Write Set Size	Contention
`linkedlist`	medium	high	medium	high
`redblack`	short	high	medium	low
`hashtable`	very short	high	small	low
`kmeans`	short	low	small	low
`vacation`	medium	high	medium	low/medium
`genome`	medium	high	medium	low

5.2.3 Experiments

I evaluate the following proposals for executing transactions.

NZSTM: The system assumes no special hardware support, and runs using NZSTM software transactions with visible reads. Visible reads have one reader's slot in-place in the object ($r = 1$), and allocate as many cache lines (P) for the reader's list as the number of running threads.

NZTM/ATMTP: The system runs NZTM, taking advantage of best-effort hardware support using the Rock-like ATMTP.

NZTM/Rock: The system runs NZTM, taking advantage of Rock's best-effort hardware support.

LogTM-SE: The system takes advantage of unbounded hardware support using LogTM-SE with *perfect* filters. LogTM-SE's perfect filters ensure that all transactional conflicts, at the cache line level, are true conflicts. Such filters are *unimplementable* in real hardware [Yen et al., 2007]; I use them because they represent an upper bound of how well LogTM-SE can perform. In contrast to ATMTP and Rock, transactions in LogTM-SE are not limited by the available hardware resources such as caches and store buffers. Moreover, LogTM-SE transactions do not impose software overheads unless they abort, in which case, LogTM-SE invokes a *blocking* software abort handler. Therefore, simulating LogTM-SE with these settings is a good-faith attempt at modeling the best unbounded hardware this simulation environment allows.

By default, both ATMTP and Rock use a *requester-wins* policy [Moir et al., 2008; Chaudhry et al., 2009a]: if there is a conflict between transactions, the transaction that issued the cache request always wins, aborting the other transaction. On the other hand, LogTM-SE uses a deadlock detection conflict resolution mechanism: it does not abort a transaction unless it detects potential deadlock, in which case the youngest transaction is aborted.

This section includes an evaluation of the software-only NZSTM, running on both the simulator and the Rock machine, for two reasons: to provide intuition on the accuracy of the simulation, and to evaluate the improvement the hybrid NZTM gains over the software NZSTM.

5.2.4 Results

This section first presents the results of the evaluation using the simulated environment, and then presents the results of the evaluation on the Rock machine.

Simulator Evaluation

The benchmarks are tested using 1, 3, 7, and 15 threads, each running on its own processor. Because of limitations of the simulator environment, one free processor is left to handle interrupts (based on advice from the GEMS developers [personal communication]). The benchmarks are compiled using GCC 3.4.6, with optimization set to level 3. The benchmarks begin by initializing the relevant data structures, and then start taking measurements, recording the simulated machine's elapsed clock cycles to complete.

Because of the careful attention to cache performance in NZTM, it should perform similarly to LogTM-SE as long as its best-effort hardware transactions commit successfully. I

expect some overhead, because NZTM must check each data object it accesses to ensure there are no conflicts with software transactions, whereas the unbounded LogTM-SE does not need such checks. If hardware transactions abort repeatedly, which is more likely to happen in high contention workloads, then NZTM's performance would be noticeably worse than that of LogTM-SE, because more of NZTM's transactions would run in software.

Figure 5.3 shows the speedup of running the benchmarks using the different transactional memory proposals, relative to LogTM-SE running on a single processor.

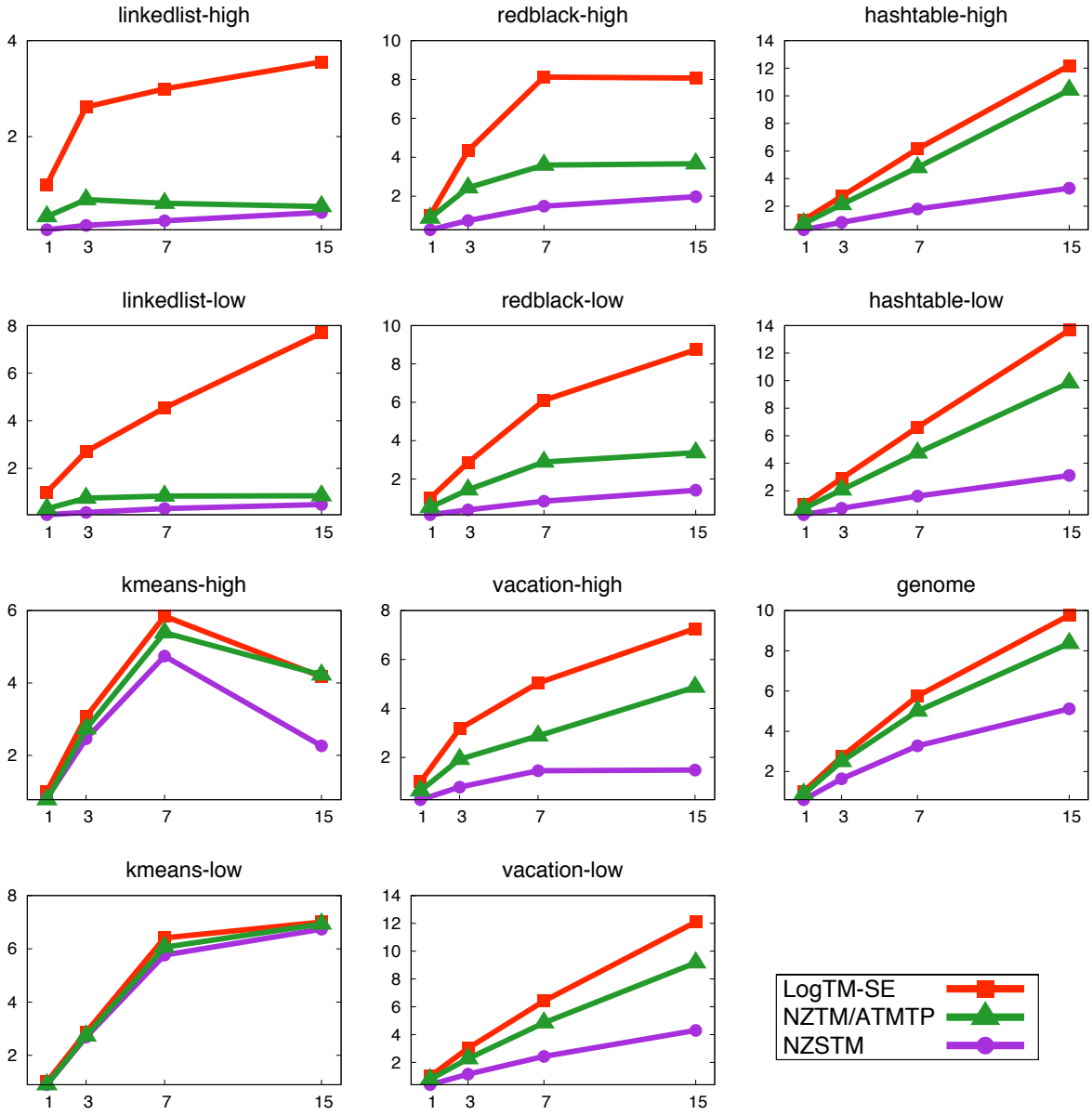

Figure 5.3: *Results of running the benchmarks on the simulator. The x-axis represents the number of threads, and the y-axis represents the speedup relative to a single thread of LogTM-SE.*

For the software NZSTM, the simulator results are consistent with the Rock machine results, as presented in the previous chapter. This increases confidence in the accuracy of the simulation.

Most of the benchmarks do not push the resource limits of ATMTP; those that abort because of resource limitations do so only occasionally. The main reason transactions abort is because of conflicts with other transactions.

The results show that LogTM-SE has the best performance of all the systems evaluated. This is not surprising, because LogTM-SE is unbounded, uses perfect filters, which are unimplementable in hardware, and has a better contention management policy than ATMTP's requester-wins. Moreover, NZTM hardware transactions have additional instrumentation to detect conflicts with software transactions, which is not necessary in an unbounded system such as LogTM-SE.

From these results, the following general observations can be made. In experiments with an overall low number of conflicts, as in `hashtable`, `genome` and `kmeans`, NZTM's performance is within 10–15% of LogTM-SE's. This gap is due to the additional instrumentation overhead NZTM's hardware transaction need to check for conflicts with software transactions.

NZTM is not as competitive with LogTM-SE in experiments with higher conflict rates. This is because of LogTM-SE's better contention management, which aborts conflicting transactions only on potential deadlock, and because LogTM-SE transactions do not need to fall back on a slower software mechanism when they abort. ATMTP's contention management, on the other hand, uses a simple requester-wins policy: when ATMTP detects a conflict between two transactions, one transaction always aborts. This could lead to many transactions falling back on software. Improvements to the underlying hardware's contention management should therefore reflect positively on NZTM.

NZTM significantly outperforms NZSTM in all cases. For most workloads, NZTM is at least twice as fast as NZSTM, and is three times as fast for some workloads.

Looking more specifically at the individual benchmarks, `hashtable` has a small number of conflicts; the results show that at 15 processors, less than 1% of NZTM transactions abort, and most of its transactions succeed in hardware. Therefore, this benchmark is a good indicator of the inherent overhead imposed by NZTM to detect conflicts with software transactions.

The `linkedlist` and `redblack` benchmarks have a higher number of conflicts than `hashtable`, leading to a larger performance gap between NZTM and LogTM-SE. The `linkedlist` benchmark has more conflicts than `redblack`; at 15 processors, about 19% of transactions in `linkedlist` abort, compared with 14% for `redblack`. The number of conflicts is proportional to the number of running threads, which is why the performance gap grows bigger in `linkedlist`, compared with `redblack`, as the number of threads increases. I expect a better hardware contention management policy would improve NZTM's performance for such benchmarks.

In the `kmeans` benchmark, less than 10% of the workload is transactional, and there are few conflicts. This results in the performance of NZTM and LogTM-SE being closer than in most other benchmarks. An important observation is that the additional metadata and instrumentation overhead attached to transactional objects does not noticeably impact performance outside transactions.

The `vacation` benchmark uses linked list and red-black tree data structures, and its performance is therefore similar to the two microbenchmarks that are based on these structures. In addition, its transactions are significantly larger, in terms of runtime and size of the read and write sets, than all the other evaluated benchmarks. At 15 processors, about 25% of all NZTM hardware transactions abort because of resource limitations.

The `genome` benchmark does not have many conflicting transactions; therefore, its performance is similar to `hashtable` under the different proposals.

Rock Evaluation

I experimented with running NZTM on Rock, and encountered a few challenges using Rock's best-effort hardware support. The main challenge being that transactions abort for many more reasons than they do when running on the simulator. The ones I found to be particularly challenging were aborts caused by mis-speculation, and aborts caused by function calls inside transactions.

Dice et al. [2009b] present a detailed discussion of working around some of the causes of aborts in Rock, specifically aborts caused by mis-speculation and by function calls inside transactions. In these two cases, hardware transactions abort because there are limited resources available for checkpointing register state. When Rock executes a hardware transaction, it is already using these limited checkpointing resources; therefore, if it performs additional speculation to that of being in a transaction, and this speculation fails, Rock's only recourse is to abort the transaction. If NZTM were able to stop Rock from performing additional speculation inside transactions, fewer transactions would abort.

To address this problem, a new branch instruction is introduced in Rock, *branch if not ready* (`brnr`). This instruction specifies a register; if the contents of that register are not ready, i.e., cannot be accessed without speculation, the branch is taken. This instruction can be used to stall program execution until the contents of the specified register are ready and can be accessed without speculation.

For example, assume code that loads a value into a register, `o0`, and then increments the contents of that register. To ensure that the increment operation does not use a speculative value of the register, the `brnr` instruction could apply a *speculation barrier* in the manner illustrated by the following assembly code.

```
    ld    <address>, %o0    ! Load a memory location into register o0.
wait:
    brnr  %o0, wait         ! Spin until the load of register o0 has completed.
    nop                     ! SPARC control transfer instructions have a delay slot.
    add   %o0, 1 ,%o0       ! Increment the register's value.
```

This new instruction can also allow function calls to be made inside a transaction without aborting it. In this case, `brnr` is used to stall program execution until the contents of *any* register are ready immediately following a `restore` instruction. The `restore` instruction is part of the SPARC assembly function call idiom, and is used to pop the calling function's register set from the *register window* before returning from the function [Weaver and Germond, 2000]. The only explanation given to why a `brnr` speculation barrier prevents `restore` instructions from aborting a transaction is "for technical reasons related to the resources Rock has available for checkpointing register state" [Dice et al., 2009b].

To illustrate, a typical function return sequence in SPARC assembly is as follows.

```
    ret                     ! Return from the function.
    restore                 ! Restore the calling function's register set.
                            ! Note: Although this restore appears after the ret instruction,
                            ! it is in fact executed during the ret instruction's delay slot.
```

Care must be taken when transforming the code segment above, because the `restore` instruction is executed inside the `ret` instruction's *delay slot* [Weaver and Germond, 2000]. A valid transformation of that code, using the `brnr` speculation barrier, is as follows.

```
    restore                 ! Restore the calling function's register set.
wait:
    brnr  %g0, wait         ! Spin until the load of any register has completed.
    nop                     ! Fill the brnr delay slot.
    retl                    ! Return from the function.
    nop                     ! Fill the retl delay slot.
```

Instrumenting code with the `brnr` instruction imposes additional overhead. This instrumentation prevents speculation, speculation that could improve performance when successful, by overlapping instructions with stalls. This instrumentation also incurs the overhead of executing the `brnr` instruction itself.

NZTM shares the same workload code path between hardware and software transactions. Therefore, using `brnr` adds overhead to the software path as well, where `brnr` serves no useful purpose. Generating separate code paths for hardware and software transactions would solve this problem, but at the cost of bigger code, less locality in the instruction cache, and therefore a higher probability of TLB misses, which abort hardware transactions. Moreover, this instrumentation does *not* guarantee that transactions will commit successfully in hardware; it only makes it more likely.

Another challenge on Rock is debugging transactions and determining the cause of their aborts. The simulator has perfect knowledge of the state of the system, and it is possible to trace the exact causes of an aborted transaction. On Rock, the only method of getting information on the cause of an aborted transaction is by using the CPS register, which does not provide detailed information. Typical methods, such as logging inside a transaction,

do not work, because when a transaction aborts, all logging is rolled back. Attaching a debugger to observe a transaction as it runs does not work either. By trying to observe a transaction's transient modifications, the debugger conflicts with the transaction, causing it to abort. In addition, debuggers typically use interrupts to communicate with the threads they are debugging, which also abort hardware transactions.

To evaluate NZTM on Rock, I wrote a Python script that instruments the benchmarks using the `brnr` speculation barrier. Initial tests showed that instrumenting every load from a register by adding a `brnr` imposes a big penalty, making hardware transactions run slower than non-instrumented NZSTM *software* transactions. Therefore, the script instruments only `restore` instructions, to allow function calls inside transactions.

The `linkedlist`, `hashtable`, and `kmeans` benchmarks did not require any instrumentation, because they do not make any function calls inside transactions. The remaining benchmarks were instrumented.

In the tests, a significant number of the microbenchmarks' transactions committed successfully using hardware in the absence of contention, i.e., when running on a single thread. As for the STAMP benchmarks, only the `kmeans` benchmark's transactions committed successfully using hardware. Both the `genome` and `vacation` benchmarks exceed the resources available for Rock's best-effort hardware support, and virtually all of their transactions fall back on software.

Table 5.5 presents the benchmarks and the percentage of transactions that successfully commit in hardware, when running on a single thread, using Rock's hardware support.

Table 5.5: *Benchmark hardware transaction commit rate on Rock in the absence of contention*

Benchmark	Hardware Commit
`linkedlist-high`	30%
`linkedlist-low`	50%
`redblack-high`	90%
`redblack-low`	95%
`hashtable-high`	100%
`hashtable-low`	100%
`kmeans-high`	100%
`kmeans-low`	100%
`vacation-high`	0%
`vacation-low`	0%
`genome`	0%

The benchmarks are tested using 1, 2, 4, 8, and 16 threads, each thread running on a separate core. The benchmarks are compiled using GCC 3.4.6, with optimization set to level 3. The benchmarks begin by initializing the relevant data structures, and then start taking measurements, recording the elapsed wall clock time to complete. Each benchmark is run three times, and the average time of the three runs is recorded.

NZTM is optimized for the case in which hardware support is available and able to commit most transactions. Rock's hardware transactional support is limited and not capable of

committing most transactions; therefore, many transactions would fall back on software. I do not expect that the results of evaluating NZTM using Rock would be comparable to the results using ATMTP, which was capable of committing most transactions in hardware. However, workloads that succeed in hardware most of the time should benefit from Rock's hardware support.

Figure 5.4 shows the speedup of running the benchmarks using the different schemes, relative to a single global lock (not shown) running on a single processor. The data is normalized to a single global lock because it demonstrates the performance achievable on systems that have no transactional memory support, with a similar level of programming complexity as using transactions.

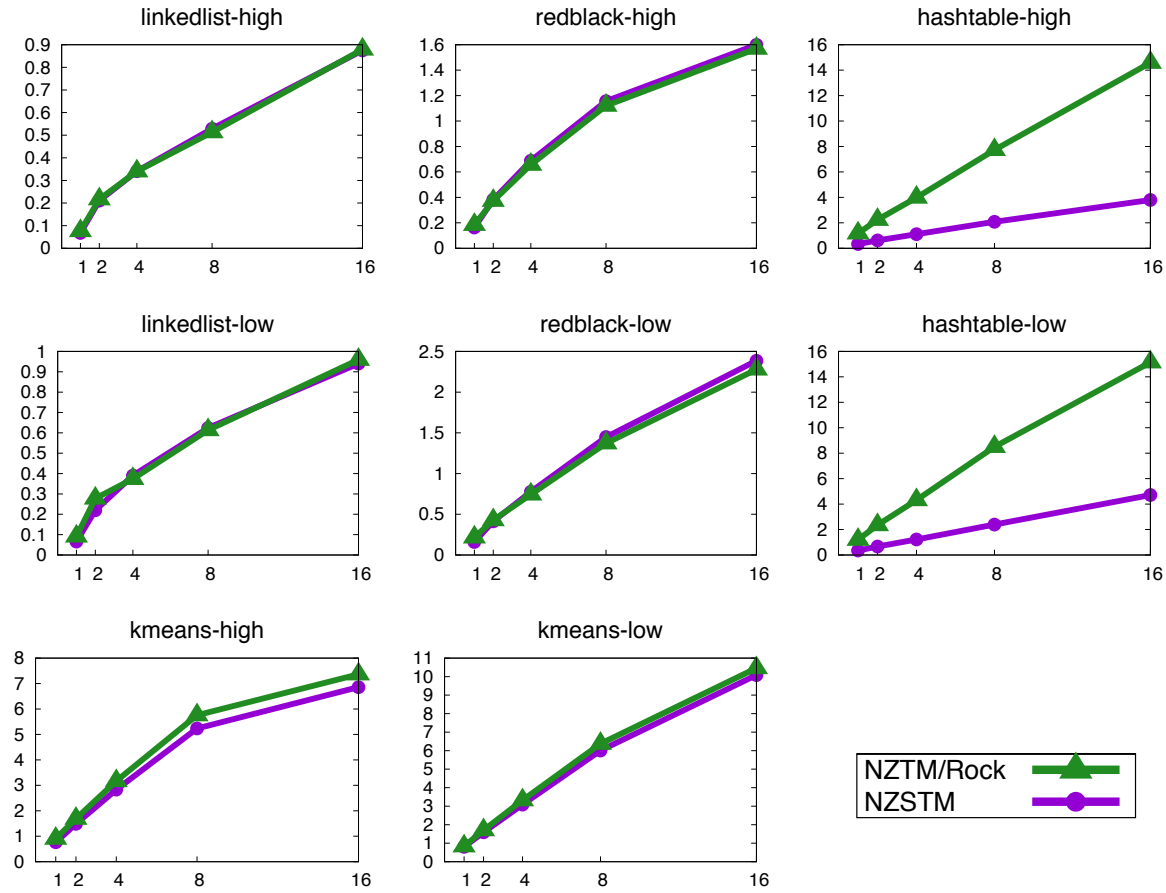

Figure 5.4: *Results of running the benchmarks on the Rock machine. The x-axis represents the number of threads, and the y-axis represents the speedup relative to a single thread using a single global lock.*

NZTM is not performing as well as it did in the simulator, because of the limitations of Rock's hardware support. This is causing many of NZTM's hardware transactions to abort and fall back on software. Trying to run a transaction in hardware first, and then aborting, incurs more overhead compared with immediately running a transaction in software, which is why NZSTM performs slightly better than NZTM in some of these tests.

I investigated the cause of aborts, and found the main reason was mis-speculation. This is indicated by the CPS register's *Unresolved Control Transfer* (UCTI) bit being set for more than 80% of aborted transactions. Moreover, the ratio of aborted transactions increases as the number of threads increases, which would not typically be surprising; however, most of the increase in the number of aborted transactions is caused by mis-speculation rather than conflicts between transactions. This could be because Rock shares its branch predictor between its cores [Chaudhry et al., 2009a], rendering its speculation less accurate as the number of running cores increases.

As mentioned earlier, I tried instrumenting load and store instructions using `brnr` to reduce these aborts. This instrumentation added overhead to the transactions without producing any significant reduction in the number of aborted transactions: with every load and store instruction instrumented, more transactions abort because of TLB misses.

The only benchmarks whose performance benefits from Rock's hardware support were `hashtable` and `kmeans`, benchmarks characterized by short transactions and low contention. For the `kmeans` benchmark, the improvement does not appear as pronounced as `hashtable`, because most of the workload in `kmeans` is non-transactional. However, the transactional component of `kmeans` runs *twice* as fast using NZTM as it does using NZSTM.

5.2.5 Discussion

The evaluation using the simulator and using Rock shows that NZTM's performance depends on the underlying hardware support. This is not surprising, considering NZTM is a hybrid transactional memory that aims to speed up its transactions by running as many of them in hardware as possible.

When NZTM's underlying hardware is sufficient to commit most transactions, it performs significantly better than the software NZSTM, and is also competitive with unbounded hardware transactional memory. If the underlying hardware is less capable of successfully committing transactions, the performance of NZTM suffers proportionally with the number of aborted hardware transactions.

As long as most of its transactions commit successfully in hardware, the biggest overhead in NZTM is the instrumentation hardware transactions need to check for conflicts with software transactions. Hardware support that puts the burden of checking for conflicts between hardware and software on software transactions, rather than on hardware transactions, would mitigate these overheads. Such hardware was proposed by Baugh et al. [2008], and if present, NZTM needs little modification to be able to take advantage of that support.

The evaluation of NZTM on Rock shows that the level of hardware support currently available is not sufficient for running anything but short transactions, at least in a hybrid context. This is to be expected, particularly because longer code is more likely to make function calls, trigger exceptions, and mis-speculate — all of which cause hardware transactions

to abort. This is not to dismiss the usefulness of Rock's hardware support; as this evaluation shows, Rock is capable of running and improving the performance of short transactions. Moreover, as explained in the previous chapter, Rock's hardware support — assuming it provides certain guarantees — could also simplify the NZSTM algorithm.

In my opinion, the main improvement that could be made to Rock, at least as far as NZTM is concerned, would be to prevent mis-speculation from aborting transactions. As mentioned, more than 80% of the aborts in Rock are caused by mis-speculation. This problem could be solved by adding more checkpointing resources in hardware that allow Rock transactions to rollback to the instruction that caused the mis-speculation, rather than abort the transaction. A simpler solution would be to disallow speculation entirely inside transactions; this would deny transactions the benefits of speculation, but because the cost of mis-speculation in transactions is so high, this solution might be justified. My later work on value prediction in transactions supports this argument (page 132).

Other issues that also affect performance are aborts caused by TLB misses. It is not clear, based on the available documentation, why TLB misses cause transactions to abort; therefore, it is difficult to comment on how such aborts could be prevented. Nevertheless, if the number of TLB misses were reduced, then aborts caused by TLB misses should also be reduced. Therefore, increasing the size of Rock's TLBs could be an adequate solution.

Another aspect that would help in developing programs that use Rock's transactional memory support is detailed information on what caused a transaction to abort. The main source of information is the CPS register, which sets flags that indicate what caused a transaction to abort (Section 2.8); however, these flags do not provide sufficient information. For example, if a transaction aborts because of a TLB miss, it is not known which instruction or data access triggered the TLB miss. If Rock were to provide the value of the program counter of the instruction that caused the abort, that would narrow down its cause and help in optimizing the program. Hardware support for debugger integration would also be very helpful; this would entail adding a hardware/software interface to expose the transactional state.

I was expecting that Rock's requester-wins conflict resolution policy would affect performance, but it did not. As mentioned, transactions were mainly aborting for reasons other than conflicts between transactions. However, if the other aspects of Rock improve, then its conflict resolution policy could become a bottleneck. Therefore, it might be beneficial if Rock were to apply a more fair policy, such as the timestamp with deadlock detection policy LogTM-SE uses.

When discussing best-effort hardware transactional memory, one of the first limitations mentioned is the size restriction on transactions. For the benchmarks in this evaluation, most transactions' read sets and write sets are within Rock's size limitations. It is the other causes for abort that proved to be more problematic.

Despite Rock's limited best-effort transactional memory support, I believe that this evaluation demonstrates the usefulness of hybrid transactional memory. NZTM can run on exist-

ing systems today; it has been compiled successfully on Intel x86 and on SPARC machines. NZTM also runs on Rock and takes advantage of its hardware support to improve performance. For now, this improvement is limited to short transactions; however, NZTM improves as the underlying hardware does.

5.3 Concluding Remarks

This chapter introduced NZTM, which demonstrates that it is possible to design a nonblocking hybrid transactional memory competitive with unbounded hardware transactional memory. NZTM shows that the hybrid approach is a viable one for using transactional memory today, in particular because the benefits from using NZTM increase as its underlying hardware support improves.

NZTM was evaluated using both a simulator and a real machine with hardware transactional memory support. The evaluation shows that NZTM does not eliminate all overhead, compared with unbounded hardware transactional memory, mainly because hardware transactions in NZTM must check for conflicts with software transactions. However, with little modification, NZTM can take advantage of hardware techniques that promise to eliminate this overhead, such as the techniques proposed by Baugh et al. [2008].

The hybrid approach of Damron et al. [2006], which is optimized for best-effort hardware capable of committing most transactions in hardware, might not be the most suitable approach on Rock. This evaluation, and the evaluation of other groups [Dice et al., 2009a, 2010], shows that Rock's limited transactional support seems to be better suited for small transactions with low contention. Therefore, unless Rock's hardware support for transactions improves, research into different ways of leveraging limited transactional support in a hybrid system is needed.

In my opinion, the NZTM algorithm itself presents a compelling argument for transactional memory. As the reader might have noticed, the NZTM algorithm is significantly simpler than the NZSTM algorithm, because NZTM relies on transactional memory in its design.

Chapter 6

Parallel Python

> *Just Say No to the combined evils of locking, deadlocks, lock granularity, live-locks, nondeterminism and race conditions.*
>
> — *Guido van Rossum,* author of the Python programming language, defending the use of a single global lock in CPython *[van Rossum, 2007]*

This chapter reports on my experiences of using Rock's best-effort hardware transactional memory to improve concurrency in Python [van Rossum, 2009a]. Python is a high level programming language with a design philosophy that emphasizes code readability. Since its release in 1991, Python has become one of the most popular programming languages[1], and is now a standard component of many operating systems such as Apple's OS X, Sun's Solaris, and various Linux distributions.

There are many different implementations of Python; the reference implementation is *CPython* [van Rossum, 2009a], a byte-code interpreter written in C. CPython was developed before the multicore era, and therefore not designed with parallelism in mind. CPython supports multiple threading; however, CPython protects its critical sections using a single global lock, known as the *Global Interpreter Lock* (GIL) [van Rossum, 2009b]. The GIL protects *all* accesses to CPython's data structures; therefore, the GIL serializes all parallel threads that access CPython's data structures. In practice, multiple threads can run in parallel only when executing external modules that do not acquire the GIL, or when performing blocking operations such as I/O, in which case the thread releases the GIL before the operation begins, and attempts to reacquire the GIL once the operation is over.

A thread typically acquires the GIL to execute a number of byte-code instructions. This number is an adjustable runtime parameter, and can be as small as a single instruction. Because there is overhead involved in acquiring and releasing the GIL, the longer a thread holds the GIL the more it can amortize this overhead, which is why this number is set to one hundred

[1] According to the TIOBE Programming Community Index for May 2010 [TIO, 2010], Python ranks as the seventh most popular programming language.

byte-code instructions by default. Figure 6.1 shows an example of two concurrent threads in CPython.

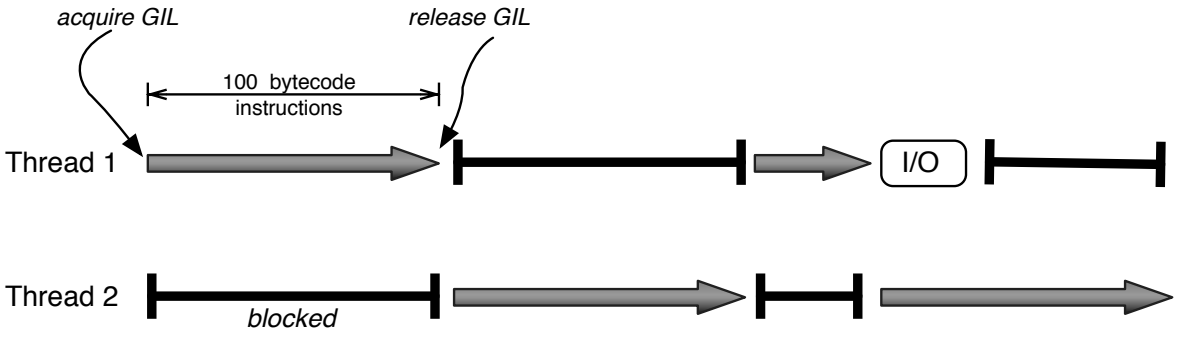

Figure 6.1: *An example of running two concurrent threads in CPython*

CPython creates and starts using the GIL only after the interpreter spawns more than one thread. Therefore, a single thread does not incur any locking overhead.

Neither the GIL, nor any aspect of its implementation, are part of the Python programming language specification. The GIL is an implementation detail used in the language's reference implementation, CPython, and others such as *PyPy* [Rigo and Pedroni, 2006]. For example, *Jython* [Juneau, Baker, Wierzbicki, Munoz, and Ng, 2010], a Python implementation that runs on the Java Virtual Machine, does not use global locks, but relies on the Java Virtual Machine's native concurrency mechanisms instead.

The GIL has often been a contentious issue within the Python community, with many criticisms of the Python language stemming from its use of the GIL [Beazley, 2009]. Expert CPython developers tried to decompose the GIL into multiple fine-grained locks; however, the resulting overhead in the single-threaded case — which is the common case for most Python programs — was a slowdown of a factor of two; so the attempt was abandoned [Stein, 2001]. This is yet another testament to the difficulties involved in parallel programming.

This chapter investigates whether Rock's transactional support could improve concurrency in CPython. The CPython interpreter was modified to use best-effort hardware transactions, and fall back on the GIL when unable to commit in hardware. The modifications were minimal; however, some of CPython's shared data structures were altered to handle false conflicts arising from CPython's management of the shared data. The modified CPython interpreter can run small, simple, workloads and scale almost linearly, while improving the concurrency of more complex workloads.

The main contributions presented in this chapter are the following.

- I describe the changes to make CPython more concurrent and scalable, as well as the obstacles encountered in the process.
- I present a preliminary performance evaluation of the modified CPython interpreter on Rock. Others have investigated the use of transactions to improve Python's perfor-

mance [Riley and Zilles, 2006; Blundell et al., 2010]; to the best of my knowledge, this work is the first to evaluate using transactions with Python on a real machine, rather than on a simulator.

The remainder of this chapter is organized as follows. Section 6.1 describes the modifications to the CPython interpreter and the challenges encountered. Section 6.2 presents a preliminary performance evaluation of the concurrent CPython on Rock. In light of the evaluation, Section 6.3 discusses some of the possible design alternatives. Section 6.4 discusses some of the related work in using transactional memory to improve concurrency in Python. Finally, Section 6.5 concludes this chapter.

6.1 Concurrent CPython

The target implementation was CPython 2.6.4 [Pyt, 2009], the most recent stable version at the time of the evaluation.

Because a best-effort hardware transactional memory is used, my approach was to apply lock elision to the GIL [Rajwar and Goodman, 2001; Dice et al., 2009a]. The modified CPython first attempts to run a GIL critical section in hardware, and acquires the GIL only if the hardware transaction repeatedly aborts.

The modified CPython does not use software transactions as a fallback mechanism. This is because the transactional memory libraries that can be applied to the CPython codebase either require manual instrumentation of the code (e.g., NZTM), or impose certain restrictions on what type of code can run inside transactions (e.g., Sun's transactional memory compiler). The CPython codebase is large, with over 300,000 lines of C code; given the resources available, restructuring such a codebase was impractical. Therefore, lock elision was the only viable approach, because it requires modification only to the code involved in lock acquisition and release.

The granularity of the GIL was reduced to one byte-code instruction instead of the default one hundred, because Rock's best-effort hardware is not likely to be able to commit large transactions. This change was hardcoded to remove the overhead of checking the number of byte-code instructions the current critical section has run so far.

The code for acquiring and releasing the GIL was replaced with code that attempts the transaction in hardware, and falls back on the GIL if it repeatedly aborts. The hardware transaction first checks that the GIL is not acquired, and then proceeds with the critical section. This check is necessary to detect conflicts with other threads that acquire the GIL; if another thread acquires the GIL, this would immediately abort concurrently running hardware transactions. The code listing below presents a simplified version of the modifications.

```
void acquire_gil()
{
   /* Attempt the transaction in hardware, fall back on the GIL if it aborts. */
   if (chkpt()) {
      if (gil_is_locked()) {
         /*
          * If the GIL is held by another thread, abort the transaction.
          * This results in this thread falling back on software,
          * and attempting to acquire the GIL.
          */
         abort();
      }
   } else {
      /* Acquire the GIL because the hardware transaction aborted. */
      gil_acquire();
   }
}


void release_gil()
{
   if (gil_is_locked()) {
      /*
       * If the GIL is acquired and this thread is in a critical section, it means that
       * this thread is the one who has the GIL, therefore, release the GIL.
       */
      gil_release();
   } else {
      /*
       * If the GIL is not acquired by anyone, and the thread reaches this point,
       * then it is running a hardware transaction and it is safe to try to commit.
       */
      commit();
   }
}
```

When a hardware transaction aborts, it either tries the transaction again or acquires the GIL. The policy for deciding whether to fall back on the GIL when a hardware transaction aborts is the same one NZTM uses (Section 5.1.3). If the transaction aborts because of a fundamental limitation then it does not try again in hardware; otherwise, it tries again in hardware a few more times.

CPython releases the GIL when performing operations such as I/O, or when running external modules that do not access CPython's internal data structures. Because many of these operations are difficult to handle in hardware transactions, this eases the process of using transactions in CPython, and makes it more likely they will commit successfully in hardware.

When running in single-threaded mode, CPython does not check or acquire the GIL, because the overhead of acquiring the GIL is not necessary. The modified CPython maintains this aspect: when running in single-threaded mode, the modified CPython does not run any hardware transactions. CPython enters multi-threaded mode when the first additional thread is spawned, which initializes and acquires the GIL. There is no mechanism in CPython to exit multi-threaded mode, even after all spawned threads are destroyed.

Eliding the GIL was not by itself enough to make CPython run concurrently. The first tests just after eliding the GIL resulted in no additional scalability or concurrency. Most transactions were aborting because of conflicts with other transactions, even though the Python code of the tests does not logically share any data.

The cause of this problem was conflicts over global data structures, such as the ones used for memory management and for maintaining the current thread's state. Because CPython was not designed for concurrency, it defines many variables as global variables when they are conceptually thread-local. I annotated these variables with the `__thread` keyword to make them thread-local [Stallman, 2004].

Not all global variables can be marked as thread-local without breaking some of the CPython invariants. For example, take the object `_Py_TrueStruct`, which designates the value representing `True`. This object is shared among all threads. Because CPython knows that there is only one instance of this object in the system, it often uses the value of *pointers* to this object to test whether a certain object is `True`, rather than dereference the pointer to check the actual value it contains. If `_Py_TrueStruct` becomes a thread-local object, then all code that uses this shortcut must also be modified. This is because changing the `_Py_TrueStruct` into a thread-local object creates multiple instances of this object, all of which are a `True` object. Therefore, checking if an object is `True` just by checking whether it points to `_Py_TrueStruct` would no longer be sufficient. The same problem applies to other CPython objects, such as the Python *singletons* `False` and `None`.

Another difficulty in parallelizing CPython is its use of *reference counting* for memory management. With reference counting, every time an object is referenced a reference counter is incremented. When an object is no longer referenced, the counter is decremented. When the reference counter reaches zero, CPython knows that the object is not needed any more and deallocates it.

When used with transactions, the problem with reference counting is that every transaction that accesses an object also modifies its reference counter. Transactions that access an object, even just for reading, modify the object. If the object is shared, then transactions that conceptually have no conflicts are conflicting, as far as the underlying transactional memory system is concerned.

This problem is exacerbated by false sharing, at the cache line level, between CPython objects. Some objects, such as the Python `True` and `False` objects, are sometimes allocated on the same cache line. Because Rock's hardware transactional memory detects conflicts at cache line granularity, a change to one object's reference counter is construed as a conflict with the other.

I solved the problem by creating a new `DoNotDeallocate` flag associated with every Python object. The flag is set for objects that exist for the whole lifetime of a CPython runtime instance. This flag indicates that there is no need to dynamically deallocate these objects and no need to track or update their reference counter. This eliminates false conflicts when accessing these objects in a transaction. It also obviates the need for declaring Python singletons, such as `True`, as thread-local variables. Instead, all singletons are flagged as objects that should not be deallocated.

Adding this flag could incur additional overhead, because every time a thread accesses an object, it must check its `DoNotDeallocate` flag before deciding whether to adjust the object's reference counter. In practice, such checks can be eliminated, as well as all calls to try to increment or decrement an object's reference counter, in code where it can be determined that the object is one of these lifetime objects.

This solution is not suitable for all shared CPython objects, only for objects that have the lifetime of the interpreter. CPython also creates temporary shared objects, which need to be deallocated to prevent memory leaks. For example, CPython dynamically creates objects that represent Python code, functions, metadata such as variable names, and constants used in Python functions. The same solution of using the `DoNotDeallocate` flag was applied to these objects even though it results in a memory leak. In my experiments, the memory leak was small, allowing me to proceed with the evaluation. Section 6.3 discusses possible solutions to this problem.

6.2 Evaluation

I evaluated the modified CPython on Rock. I did not use the ATMTP simulator because the main goal was to evaluate how a *real* hardware transactional memory implementation could improve the concurrency of a real application.

To increase the likelihood of transactions committing successfully in the presence of function calls, I instrumented `restore` instructions in the modified CPython interpreter using the `brnr` speculation barrier (Section 5.2.4). CPython was compiled using GCC 3.4.6, with optimization set to level 3.

I used two simple benchmarks that repeatedly modify a local variable in a loop: `iterate` and `count` [Beazley, 2009]. Although both benchmarks accomplish the same general goal, each uses a different set of CPython byte-code instructions. The code listing below is for these two benchmarks.

```
def iterate(iterations):
    for x in xrange(0, iterations):
        pass


def count(iterations):
    while iterations > 0:
        iterations -= 1
```

These simple benchmarks are not representative of realistic workloads. I use them as a litmus test for whether CPython could run concurrently, and to evaluate eliding the GIL for a few simple CPython byte-code instructions.

For a more complex benchmark, I used `pystone`, a synthetic benchmark included in the CPython distribution. The `pystone` benchmark is a translation of the Dhrystone benchmark [Weicker, 1984], a computationally-intensive integer benchmark.[2]

When multiple threads are spawned, each CPython thread runs its own instance of these benchmarks. The `iterate` and `count` benchmarks perform 2,000,000 iterations each, and the `pystone` benchmark performs 10,000 iterations. The number of hardware transaction attempts is set to four, before falling back on the GIL.

I first evaluate the effects of the modifications on the single-threaded case, relative to the unmodified CPython. Before any threads are spawned, the modifications do not incur any overhead. CPython does not check or attempt to acquire the GIL in the single-threaded case, and the modifications do not attempt to run the code using hardware transactions in the single-threaded case either. The other modifications do not incur any overhead in the single-threaded case.

To observe the overhead the modifications incur on a single thread once the interpreter goes into multithreaded mode and starts using transactions, the benchmarks force CPython to go into the multithreaded mode by spawning and immediately destroying a thread. Table 6.1 presents the slowdown the modifications incur when using hardware transactions on a single thread, compared with running a single unmodified CPython thread, which incurs neither GIL nor transactional overhead.

Table 6.1: *Concurrent CPython slowdown relative to a single unmodified CPython thread*

Benchmark	Slowdown
`iterate`	2.8
`count`	3.9
`pystone`	2.1
average	2.9

The modified CPython incurs a slowdown of about a factor of three. This slowdown is mainly because Rock hardware transactional memory instructions and the code that attempts to elide the GIL add significant overhead. Although this overhead is high, the modified CPython incurs this overhead *only* when running multiple threads, whereas the original implementation would *serialize* those threads.

Figure 6.2 presents the results for the performance and scalability of the benchmarks going from 1 to 16 threads, relative to running a single unmodified CPython thread, which does not incur any synchronization overhead. These tests force the modified CPython to go into the multithreaded mode, by spawning and immediately destroying a thread, even when running only 1 thread.

Because these benchmarks do not conceptually share any data, ideally, they would scale linearly with the number of threads. Rock's hardware support is best-effort; if a transaction

[2] In turn, Dhrystone is inspired by Whetstone [Curnow and Wichman, 1976], a computationally-intensive floating-point benchmark.

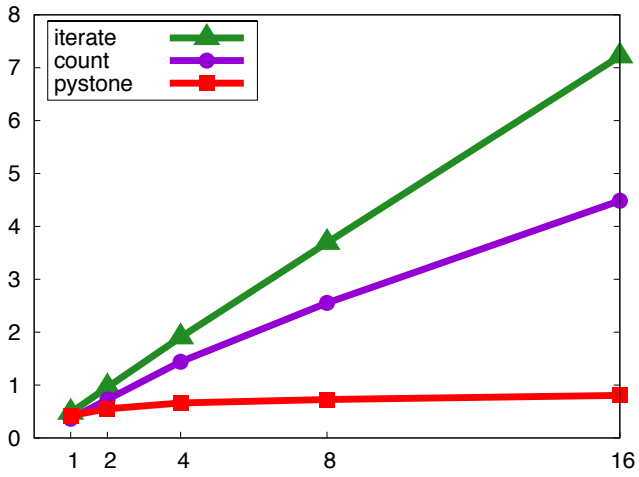

Figure 6.2: *Results of running the benchmarks using the modified CPython on the Rock machine. The x-axis represents the number of threads, and the y-axis represents the speedup relative to a single unmodified CPython thread that does not acquire the GIL.*

aborts, it acquires the GIL, which serializes other threads. Although I expect to see improvement in performance, I do not expect it to be linear with the number of threads.

The two simple benchmarks, `iterate` and `count`, scale well. Most of their hardware transactions commit, and from three threads onwards, they perform better than the unmodified single-threaded CPython. This is consistent with the initial observation that the modified CPython incurs a slowdown of a factor of three in the single-threaded case. I note that `iterate` scales better than `count`. The `iterate` benchmark uses CPython's built-in `xrange` function. The `xrange` function is optimized to be fast, and has a smaller memory footprint [van Rossum, 2009d]; therefore, transactions involving this function are more likely to commit successfully on their first attempt.

The `pystone` benchmark is able to take advantage of some of the parallelism, but its performance does not improve much beyond four threads. The reason this benchmark does not perform as well as the other two is the high number of aborted transactions: about half of its transactions repeatedly abort and eventually fall back on the GIL. These transactions are aborting because of mis-speculation, TLB misses, and function calls in transactions. The aborts caused by mis-speculation have similar behavior to what was experienced with NZTM (Section 5.2.4). To reduce the number of aborts due to mis-speculation, I tried to instrument load instructions as well with a `brnr` speculation barrier; however, the additional overhead hurt performance more than it helped.

In these experiments, I noticed that there are more TLB misses in CPython than the benchmarks used for evaluating NZTM, and therefore, more transactions aborting because of TLB misses. This might be because CPython is a bigger program that accesses more data, which in turn causes it to miss in the TLB more often than the smaller benchmarks used in the NZTM evaluation.

Finally, even though all `restore` instructions are instrumented with a `brnr` speculation barrier, the modified CPython still experienced aborts caused by function calls. I was unable to explain why the speculation barrier was not preventing these aborts as it did for NZTM; however, the nature of Rock's hardware support means that no transaction is guaranteed to commit, regardless of the safeguards.

Applying Amdahl's law, if half of the transactions fall back on the GIL, then the best speedup to expect is a factor of 2, assuming an infinite number of processors. This is consistent with my findings: the relative performance of `pystone` running on 16 threads to 1 thread is about 1.9.

These experiments, and Amdahl's law, highlight the potential problems of falling back on a mutually exclusive solution when transactions abort. As long as most hardware transactions commit, and the serial component remains small, then lock elision works well. However, even a small serial component has a big impact on scalability. A big serial component, such as the one in the `pystone` experiments, is devastating to scalability.

These experiments also highlight that, with little modification and some hardware support, existing programs can benefit immensely from transactional support — even from limited best-effort support such as the one in Rock. I expect that the modified CPython would improve as its underlying hardware support improves, without additional modifications to the software.

6.3 Design Alternatives

The main challenge in parallelizing CPython was handling the operations that are treated as conflicts by the underlying system, even though they are conceptually nonconflicting. The two main causes of this problem in CPython are its use of global variables instead of thread-local variables, and its use of reference counting for memory management.

The solution to the problem of global variables did not require much effort. The solution I use instructs the compiler to create a thread-local copy of these global variables, which I believe is a satisfactory solution. On the other hand, the memory management problem is a more complex one.

CPython uses reference counting, instead of other methods of garbage collection, because it is more portable than the alternatives [van Rossum, 2009c]. Although portable, the Boehm-Demers-Weiser conservative garbage collector does not run on all systems CPython supports. Nevertheless, using the Boehm-Demers-Weiser conservative garbage collector instead of reference counting would solve the problem reference counting causes [Riley and Zilles, 2006].

The reference counting implementation CPython uses does not scale because CPython was not designed with scalability in mind. However, many research groups have proposed different techniques of making reference counting scalable. One possibility is to use the

Scalable Non-Zero Indicator (SNZI) [Ellen, Lev, Luchangco, and Moir, 2007] to represent reference counts. SNZI uses a hierarchal algorithm, and is typically implemented as a tree data structure distributed across several memory locations. Ellen et al. show that, because of SNZI's hierarchical nature, it scales well when used as a reference counter, and works particularly well in the context of transactional memory.

Another alternative is to modify the underlying hardware so that it is aware of dependences and can forward the values of modified data, such as modified reference counts, between transactions. Ramadan, Rossbach, and Witchel [2008] propose a dependence-aware hardware transactional memory, which could mitigate the reference counting problem. Blundell et al. [2010] have encountered the same problem and also propose a hardware solution, which is discussed in the next section. However, such support requires extensive changes to the underlying hardware, and does not solve the problem for existing systems.

In my opinion, given the resources available, the best solution to this problem is to use the Boehm-Demers-Weiser garbage collector. I did not investigate this solution, primarily because it was unclear if continued access to the Rock machine would be possible.

Another obstacle in the scalability of the modified CPython is caused by using the GIL as a fallback mechanism. As long as most transactions commit in hardware, the GIL is successfully elided, and the system scales. As the abort rate increases, the GIL becomes a serious bottleneck. Using a hybrid solution that falls back on a software transactional memory could mitigate this bottleneck, because most software transactional memory proposals allow software transactions to run concurrently. It is important, however, that the software component does not introduce much overhead, especially in the single-threaded case. Otherwise, the software transactional memory's overhead might negate most of the benefits from parallelism.

6.4 Related Work

Riley and Zilles [2006] proposed substituting GIL critical sections with hardware transactions. Instead of CPython, they targeted the PyPy Python implementation, which supports using the Boehm-Demers-Weisser garbage collector, thus eliminating conflicts caused by updating reference counts.

Riley and Zilles evaluated their proposal on a simulated unbounded hardware transactional memory that does not model instruction latency and cache behavior, and therefore does not accurately measure performance. Their goal was to examine the feasibility of using transactions as a substitute for the GIL. For the workloads they evaluate, all transactions commit successfully. Based on the average memory footprint of their transactions, they concluded that best-effort hardware support is sufficient: the average number of bytes read in the biggest benchmark they used was less than 1 KiB, and the average number of bytes written by that benchmark was less than 640 bytes.

Rock's transactional write buffer holds only 32 entries. Therefore, assuming a 64 bit write buffer entry, the write buffer would be able to hold a maximum of 256 bytes, which is not sufficient for most transactions to commit in hardware for the workloads Riley and Zilles evaluated.

Concurrently with and independently of the work in this chapter, Blundell, Raghavan, and Martin [2010] proposed a transactional hardware mechanism that symbolically tracks modifications and constraints that a transaction applies to variables, and transparently applies the modifications and checks if the constraints still hold before a transaction commits. For example, if a transaction increments a reference counter, then the system applies the increment operation at commit time, regardless of the current value of the reference counter. Moreover, if a transaction checks that the value of a reference counter is greater than zero, then the transaction can commit as long as that constraint holds at commit time, i.e., the reference counter is still greater than zero.

The work of Blundell et al. targets conflicts on auxiliary data, which often cause bottlenecks in transactions with otherwise nonconflicting operations — exactly the type of conflicts CPython's reference counting creates. Blundell et al. also modify CPython to use transactions to protect its critical sections, and present a simulator evaluation of applying their hardware scheme to the modified CPython. Their results show that, with their proposed modifications, CPython can scale almost linearly, at least up to 32 threads.

6.5 Concluding Remarks

This chapter demonstrates that, with little modification, programs not designed for concurrency can leverage transactional memory support to scale as the number of available cores increases. Towards that end, this chapter presented an evaluation of using hardware transactions in CPython on a real machine that supports hardware transactional memory. Although the results are not conclusive, they demonstrate the potential even limited best-effort hardware transactional memory has for improving performance.

I did not investigate this topic further or pursue other directions, primarily because it was unclear if continued access to the Rock machine would be possible. These preliminary results are included in this thesis because I believe they help make the case for best-effort hardware transactional memory, and hybrid transactional memory by extension, a stronger one.

Chapter 7

Transactional Conflict Decoupling and Value Prediction

This thesis argues that, if transactional memory is going to be a viable programming model, then hybrid transactional memory is the best approach to adopt. Most of the hybrid transactional memory proposals discussed in this thesis rely on underlying hardware that infers transactional conflicts from coherence conflicts. This inference could lead to false transactional conflicts that adversely affect the performance of the system as a whole.

Drawing inspiration from Huh, Chang, Burger, and Sohi [2004], this chapter demonstrates that, by decoupling transactional conflicts from coherence conflicts, hardware transactional memory can reduce false conflicts between transactions; and that by speculating on data values, it can reduce the delays coherence conflicts incur. Specifically, this chapter explains how decoupling and data speculation can improve performance in the presence of false sharing and *silent stores* (including *temporally silent stores*) [Lepak and Lipasti, 2000, 2002]. The cost of these added mechanisms is justified by showing that, because transactions are already in speculative mode, hardware transactional memory can speculate on data in ways that are infeasible in the absence of transactional support.

False sharing, in particular, can be difficult to mitigate, and its effects are especially pronounced in hardware transactional memory. Typical methods for mitigating false sharing require data restructuring, which goes against the transactional memory promises of abstraction and composition. Restructuring requires low level knowledge of the system, thereby breaking abstraction. It is also difficult to restructure external library code that suffers from false sharing, which hinders composition. In my opinion, for transactional memory to become viable for parallel programming, it should avoid the worst effects of false sharing.

The main contributions presented in this chapter are the following.

- I describe how decoupling transactional conflicts from coherence conflicts in hardware transactional memory reduces apparent transactional conflicts, and how value

prediction complements the decoupling by reducing the delays of coherence conflicts in transactions.

- To mitigate the effects of false sharing in transactions, I present *DPTM*, an enhanced best-effort hardware transactional memory that relies on a conventional coherence protocol, and adds only minor, processor-local, modifications.

- I present an evaluation of various design points for DPTM, and compare them with data restructuring by padding, a commonly used method to avoid false sharing. The evaluation shows that DPTM significantly improves performance in benchmarks that exhibit false sharing, even more than restructuring by padding does, without having adverse effects on benchmarks that do not exhibit false sharing.

The remainder of this chapter is organized as follows. Section 7.1 discusses the false sharing problem, and why it is particularly troublesome in hardware transactional memory. Section 7.2 explores how decoupling transactional conflicts from coherence conflicts, as well as value prediction, could reduce transactional conflicts and mitigate their effects. Section 7.3 presents DPTM, an example of using these ideas to mitigate the effects of false sharing. Section 7.4 presents an evaluation of different design points for DPTM. Section 7.5 discusses some of the related work. Finally, Section 7.6 concludes this chapter.

7.1 The False Sharing Problem

False sharing at the cache line level can have a big impact on the performance of parallel programs [Goodman and Woest, 1988; Bolosky and Scott, 1993; Kadiyala and Bhuyan, 1995; Hennessy and Patterson, 2006; Herlihy and Shavit, 2008]. It occurs when different processors access distinct data objects that share the same cache line, and at least one processor modifies one of the objects. Because the cache line is the unit of granularity for coherence, such *logically* nonconflicting accesses are serialized.

The effects of false sharing are not easy to mitigate. Most methods of mitigating false sharing are oriented towards the restructuring and padding of data objects, so that logically nonconflicting accesses to different objects are also nonconflicting as far as the coherence protocol is concerned [e.g., Torrellas, Lam, and Hennessy, 1994; Jeremiassen and Eggers, 1995; Harris, Fraser, and Pratt, 2002; Huh et al., 2004; Moore et al., 2006].

Restructuring data to mitigate false sharing can be difficult to apply in practice. First, the programmer needs to identify the data objects that cause false sharing. Once the objects are identified, they are aligned to cache line boundaries, which requires knowledge of machine-specific details such as the cache line size. Then, typically, each object is padded so it occupies a whole cache line by itself, which increases memory use and fragmentation, and adversely impacts locality. These changes are often machine-specific, so the benefits of code

modified in such ways might not be portable. Moreover, using high-level languages, such as Java, where the underlying virtual machine does not lay out the data structure the way it is specified in the program [Neto, 2008], can make data restructuring impractical.

False sharing may also be introduced to a program by the use of external libraries that suffer from it. Modifying external code is often difficult or infeasible, and also goes against the principles of abstraction and composition in software engineering.

False Sharing in Transactions

False sharing is a bigger problem when it occurs in conjunction with hardware transactions, because it leads to the aborting or serialization of whole transactions that could have otherwise executed concurrently [Herlihy and Moss, 1995; Moore et al., 2006]. It is a problem that has often been observed by experts on transactional memory and parallel programming.[1]

The example in Figure 7.1 shows the timeline of two concurrent transactions. These transactions access different data on the same cache line at one point during their execution, i.e., they exhibit false sharing. The transactions in this example at no point have any true conflicts, and so any order of their component operations is allowable.

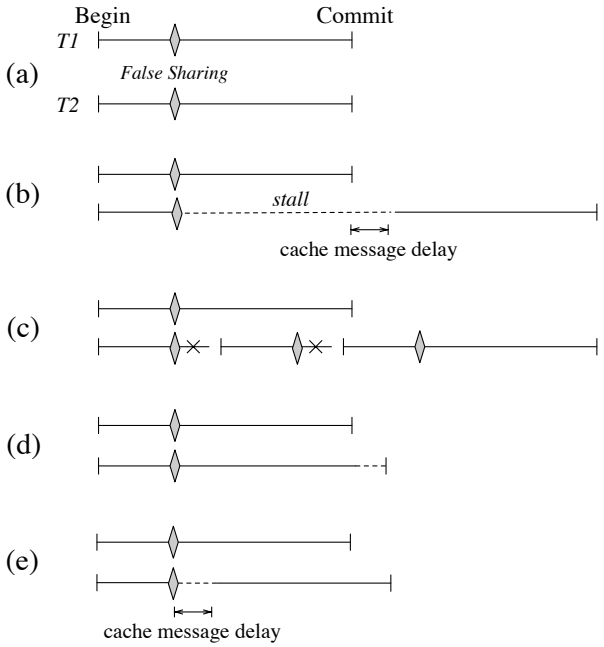

Figure 7.1: *The effect of a single instance of false sharing in transactional memory. (a) the ideal case (b) hardware transactional memory* stalling *(c) hardware transactional memory* aborting *(d) a solution that mitigates the problem of false sharing (e) false sharing in* software *transactions*

[1] Some of the experts who have encountered false sharing are Herlihy and Moss [1995]; Harris et al. [2002]; Cintra and Torrellas [2002]; Ananian and Rinard [2005]; Scherer and Scott [2005]; Moore et al. [2006]; Ramadan et al. [2007]; Grossman [2007]; Bobba et al. [2008]; Yoo, Ni, Welc, Saha, Adl-Tabatabai, and Lee [2008]; Adl-Tabatabai et al. [2006a]; Gajinov, Zyulkyarov, Unsal, Cristal, Ayguade, Harris, and Valero [2009].

Ideally, these transactions should be able to run completely in parallel, as shown in (a). However, because most hardware transactional memory proposals infer transactional conflicts from coherence conflicts [Baugh et al., 2008], such implementations do not distinguish between true and false sharing, which leads to stalling transactions (b), or aborting them altogether (c).

I expect a solution to the problem of false sharing to result in an execution similar to the one shown in (d). Such a solution would likely not eliminate all the false sharing delays because some cache data still needs to be communicated. However, it should be able to mitigate their effects by overlapping the delays with other speculative operations.

Although false sharing can impact performance negatively outside the context of hardware transactional memory, its impact is not as severe. The two concurrent *software* transactions, shown in (e), also suffer from one instance of false sharing. Because software transactional memory implementations do not associate transactional conflicts with coherence conflicts, the only impact this instance of false sharing has is that of the delay associated with the coherence protocol's handling of this conflict.

The typical solution to the problem of false sharing by restructuring and padding, in addition to the difficulties it poses in non-transactional systems, also goes against some of the promises of transactional memory, that of abstraction and composition.

Composition is the ability to combine and use different code in transactions. Using external code that is not optimized for false sharing, or that is optimized for a different machine, undermines the usefulness of composition in transactional memory. If programmers have to worry about external modules causing false sharing, they would be reluctant to use external modules in their programs.

Abstraction aims to hide irrelevant complexity and detail, but restructuring requires exposure to architectural and other low level details, knowledge a typical programmer may not have, and should not need. Therefore, transactional memory should abstract away the adverse effects of false sharing.

7.2 Coherence Decoupling and Value Prediction in Transactions

From the first hardware transactional memory proposal [Herlihy and Moss, 1993], the means used by most transactional memory proposals to detect cache coherence conflicts has also been employed and extended to detect transactional conflicts. Coherence conflict detection, however, is implementation dependent, usually classifying certain patterns as a coherence conflict even in some cases where no conflict exists.

One pattern is false sharing. While logically no sharing — and no conflict — occurs, this widely recognized problem is nevertheless treated as a coherence conflict. Likewise

with *silent stores* (including *temporally silent stores*) [Lepak and Lipasti, 2000, 2002], where two or more threads truly share data, and at least one of them writes to the data, ultimately leaving its value unchanged: no conflict occurs, yet virtually all coherence implementations treat them as a coherence conflict.

None of these patterns constitute a transactional conflict, but are characteristically treated as such because the coherence detection mechanism treats all addresses within a cache line as one. Yet these patterns may result in significant performance degradation, because their occurrence during transactional speculation can cause a transaction to stall or even to abort. Aborting may result in much more serious degradation and could, for example, be the cause of livelock.

To mitigate the effects of false conflicts in transactions, I argue that hardware transactional memory should decouple transactional conflicts from coherence conflicts. Because transactions are *already* in speculative execution mode, this makes it easier to use value prediction in transactions.

Hardware transactional memory proposals that associate transactional conflicts with coherence conflicts also associate cache line states with transactional states, i.e., modified data resides on exclusive cache lines, and read data resides on valid cache lines. These associations enable transactions to detect potential conflicts using the coherence protocol, as explained in Section 2.5, but could result in *false* transactional conflicts in the presence of false sharing and silent stores.

One alternative is to speculate inside a transaction without this association, then restore it before the transaction tries to commit. This could be done by ensuring that cache lines representing the read and write sets are in their expected state at commit time, and that the read set data matches the current data values.

Therefore, when a cache line containing read data is invalidated, instead of aborting, a transaction continues without triggering a transactional conflict, speculating that the data it has read will still be *valid*. The transaction tracks which parts of the cache line contains read data; and before it can commit, the transaction must *validate* by re-acquiring the cache line to ensure that the read data is unchanged. If it has changed, only then is a transactional conflict triggered, thereby associating transactional conflicts with *modifications* to the shared data.

This same mechanism can also be used for value prediction in transactions: a transaction could speculate on any value as long as it ensures that the read cache line is valid and holds the predicted value at the time it commits.

For example, in many hardware transactional memory proposals, when a thread attempts to read data that is not in a valid state in its cache, it requests the cache line with the data and stalls waiting for the request. Rather than stall, a transaction could predict the data value because it *already* is in speculative execution and does not have to take an additional checkpoint. When it predicts accurately, the transaction mitigates the effects of the stall.

One method of prediction would use the data value in a *stale* cache line, i.e., an invalid cache line still containing the data it held when it was last invalidated, as suggested by Huh et al. [2004]. If the cache line is stale, a transaction could predict that the particular part it wants to read has not changed, and speculate using that value. Such predictions would be accurate in the cases of false sharing and silent stores because the data the transaction is interested in has not changed.

When a transaction speculates on the value of a cache line, it must ensure that the speculation was correct before it is able to commit. A transaction must track the parts of the cache lines on which it is speculating, and validate those cache lines by commit time. It validates the cache line by acquiring the line, and ensuring that the value it predicted matches the current data value in the cache line.

7.3 DPTM Description

7.3.1 Overview

This section demonstrates how data speculation could mitigate the effects of false conflicts by example of *Decoupling and Prediction Transactional Memory* (DPTM), an enhanced best-effort hardware transactional memory proposal.

As a baseline, I assume a conventional best-effort hardware transactional memory, similar to the one in Rock. That is, one that uses eager conflict detection and lazy version management, tracks its read and write sets in a transactional cache (such as the L1 cache), and keeps transactional writes in a write buffer until the transaction commits. I also assume a cache coherence protocol that distinguishes stale cache lines from other invalid states. The baseline hardware transactional memory associates transactional conflicts with coherence conflicts; therefore, when a cache line containing transactional data receives an invalidation request (due to a coherence conflict), this triggers a transactional conflict.

DPTM modifies this baseline as follows.

DPTM decouples transactional conflicts from coherence conflicts by using value-based conflict detection [Ding et al., 2007; Olszewski et al., 2007]. The values in a transaction's read set are monitored for change, and a transactional conflict is triggered only if a value changes. This reduces the effects of false sharing, because transactional conflicts are now restricted to changes *only* to the data used inside a transaction, rather than coherence conflicts over *whole* cache lines.

Furthermore, because transactional states are no longer associated with coherence states, DPTM does not require cache lines that are part of a transaction's read set to be in a valid state or its write set to be in an exclusive state, until commit time. This could improve performance by reducing the window in which conflicts might occur — something DPTM has in common

with other lazy conflict detection transactional memory proposals [e.g., Hammond et al., 2004; Ceze et al., 2006].

DPTM can speculate when attempting a load from a stale cache line by using the value of the stale data and validating it later. When speculating, it assumes that the cache line became stale because of a coherence conflict caused by false sharing. When this prediction is accurate, it could eliminate some of the stalling caused by false sharing.

In addition to mitigating the effects of false sharing, DPTM also mitigates the effects of silent stores. In value-based conflict detection, which DPTM uses, silent stores do not trigger transactional conflicts because the actual data values do not change [Ding et al., 2007; Olszewski et al., 2007]. Furthermore, DPTM's speculation on a stale cache line that became stale because of a silent store would likely be accurate also because the actual data values do not change.

Below is a more detailed description of DPTM. Section 7.3.3, which follows, also presents a description of DPTM using simplified C-like code adapted from the simulated model.

7.3.2 Detailed Description

DPTM does not alter how the baseline begins its transaction. It alters the loading and storing of values inside a transaction, the handling of coherence conflicts and how a transactional conflict is interpreted, and the process of committing a transaction.

Loading a Value

When a transaction in DPTM attempts to load data not present in its transactional cache, it proceeds, as in the baseline, by issuing a request for the cache line and stalling for the request. Otherwise, if the cache line is present and in a valid state, the load is a cache hit.

If the cache line is stale, and DPTM *predicts* that the value has not changed, then it may serve the load using the stale data while simultaneously issuing a cache request for the data. The load proceeds as if it were a cache hit. On the other hand, if DPTM *predicts* that the value has changed, it behaves conventionally, as if the line were not present.

By default, DPTM speculates only on stale data that is already part of a transaction's read set. This is conservative, because the transaction has already read that data, and if it has changed, the transactions must abort anyway. Therefore, DPTM has little to lose in the case of a mis-prediction.

Associated with all cache lines in the transactional cache are read mark bits that indicate which parts of each line have been read [McDonald, Chung, Chafi, Minh, Carlstrom, Hammond, Kozyrakis, and Olukotun, 2005]. Each bit monitors reads from a subset of its cache line, i.e., the bit is set when its associated subset is read inside a transaction. These bits are

used for validation, because only the parts of stale cache lines with their associated bits set need to be validated before the transaction is able to commit.

The number of read mark bits added per cache line determines the granularity level of DPTM's transactional conflict detection; the greater the number of bits the finer the granularity, and the more cases of false sharing that can be detected. For example, for a 64 byte cache line and a conflict detection granularity level of 4 bytes, DPTM requires an additional 16 bits for each cache line. The conflict detection granularity could conceivably go down to the individual bit level.

When a processor receives a response to a cache request, the data in the cache whose read mark bits are set is validated against the returned data. If the data is unchanged, validation succeeds and the transaction proceeds as normal. If the data has changed, validation fails, the transaction aborts, and all read mark bits are cleared. In all cases, the old cache line data is replaced with the returned data.

Storing a Value

When a transaction performs a store, the baseline requests exclusive permissions for the cache line and stalls while it obtains these permissions, after which it stores the data in the transactional write buffer. Stores in DPTM do not request exclusive permissions at the time of the store; instead, a transaction stores the data in the write buffer immediately and does not stall. However, the transaction must obtain exclusive permissions for all cache lines in the write buffer before it can commit. DPTM, as a starting point in its design, requests exclusive permissions only at commit time.

As an extension, DPTM can choose to request exclusive permissions immediately, and does not need to stall for the request. If it postpones the request until commit time, it cannot overlap the stalling for the request with other operations.

In both cases, the time taken for the store instruction is equivalent to the time taken for a cache hit.

Conflict Management

When a transaction in the baseline hardware transactional memory receives an invalidation request for a cache line that is a part of its read or write sets, it invokes a conflict resolution mechanism. If the mechanism decides to acknowledge the request, the receiver's cache line is invalidated, aborting its transaction.

Using DPTM, a transaction that receives an invalidation request acknowledges the request and does *not* abort, anticipating that the invalidation might be due to a coherence conflict caused by false sharing, a silent store, or that the sender of the invalidation is a doomed transaction, i.e., a transaction that will eventually abort. Therefore, the transaction schedules a request for the now invalidated cache line and continues execution.

When the transaction's request is finally served and it receives the current cache line data, the transaction validates that the read data on that cache line, as specified by the read mark bits, is unchanged. If it has changed, the transaction aborts, and the read mark bits are cleared.

Cache Line Evictions

In the baseline hardware transactional memory, if a cache line that is part of a transaction's read set is evicted from its transactional (L1) cache, the transaction aborts. DPTM relies on the values in its transactional cache for validation; it can no longer track the original value that it has read after the value's cache line has been evicted. Therefore, DPTM also aborts the transaction on such evictions, because it cannot validate the evicted line.

Evictions in the non-transactional (L2) cache, unlike in the baseline, do not abort a DPTM transaction. The baseline's coherence model maintains strict inclusion; therefore, an L2 cache eviction triggers an L1 cache *invalidation*, not an eviction, if the evicted line is also in the L1 cache. If the invalidated L1 cache line is part of a transaction's read set in the baseline, the transaction aborts.

By contrast, when an L1 cache line that is part of a transaction's read set is invalidated in DPTM (whether caused by an L2 eviction or a coherence conflict), DPTM does not abort the transaction, but instead schedules a request for the cache line. The cache line is then validated when the request completes and its data arrives.

Committing a Transaction

At commit time, the baseline hardware transactional memory flushes its write buffer by writing all the values in its write buffer to memory. Because it already has all its cache lines in their correct commit states, whereby the coherence states are associated with their transactional states, this is sufficient to complete the transaction.

In DPTM, when a transaction is ready to commit, parts of its read set might not be in a valid state, and parts of its write set might not be in an exclusive state. Therefore, it employs a two stage commit.

In the first stage, DPTM attempts ensure that all its cache lines are in their correct commit states. It first issues shared cache requests of all stale lines in the cache that are part of the transaction's read set but not its write set. All the while, it validates each incoming cache line, aborting the transaction if data that is part of its read set has changed. DPTM then issues exclusive cache requests to all cache lines that are part of the transaction's write set but are not already in an exclusive state.

Once all the cache lines have been validated and are in their correct commit states, DPTM moves to the second commit phase, which is the same as the baseline hardware transactional memory's commit. When it finally commits, DPTM clears all the read mark bits.

During the second commit phase, as in the baseline, DPTM must keep all read lines in a valid state, and all written lines in an exclusive state. This results in DPTM serializing transactions' commit phases in the presence of false conflicts, whereas the baseline system serializes *whole* transactions in the presence of false conflicts.

The contention management policy during DPTM's commit phases is different from the one during a transaction. Invalidation requests for cache lines that are part of a transaction's read or write sets are denied. To prevent deadlock, the simplest policy is for the committing transaction to abort if a cache request it has sent was denied.

A more elaborate policy, which DPTM uses during its first commit phase, is as follows: if a cache line invalidation request arrives during the first commit phase, the transaction acknowledges it only if the requester is also committing and has higher priority; otherwise, it denies the request. Therefore, deadlock cannot occur, assuming each transaction's priority is unique. Invalidation requests during the second commit phase, as in the baseline, are always denied: at that point the transaction is guaranteed to commit and there is no fear of deadlock.

7.3.3 Description by Code

The simplified C-like code listing below is loosely adapted from the simulated model. This code illustrates DPTM events whose handling differs significantly from the baseline.

Loading an address in DPTM

```
int loadAddress(int *address)
{
    int *line_address = getLineAddress(address);

    for (;;) {
        if (isLineValid(lineAddress)) {
            /* the line is present and in a valid state (S,E, or M) */
            setMarkBits(address);
            return *address;
        } else if (isLineStale(lineAddress) && isMarkBitsSet(address)) {
            /*
             * the line is present in a stale state (I),
             * the line was also read before in this transaction
             */
            issueSharedRequest(lineAddress);
            return *address;
        } else {
            issueSharedRequest(lineAddress);

            /* blocks until the cache line data arrives */
            stallProcessor();
        }
    }
}
```

Receiving a cache line in DPTM

```
void incomingCacheLine(int *line_address, int incoming_data[CACHE_LINE_SIZE])
{
    for (int i = 0; i < CACHE_LINE_SIZE; i+=sizeof(int)){
```

```
        int *address = line_address + i;

        /* validates, assuming a granularity of sizeof(int) bytes */
        if (*address != incoming_data[i]) {
            clearAllReadMarkBits();
            abortTransaction();
        }
    }

    update_cache(incoming_data, line_address);

    if (isProcessorStalled()) {
        unstallProcessor();
    }
}
```

Handling a cache line invalidation request in DPTM

```
void invalidateCacheLine(int *line_address)
{
    if (isLineReadset(line_address)) {
        /* schedule a request for the cache line*/
        issueSharedRequest(lineAddress);
    }

    acknowledge();
}
```

Committing a transaction in DPTM

```
void commitTransaction()
{
    int *line_address;

    /* DPTM's first commit phase */

    foreach (line_address in getReadOnlyAddresses()) {
        if (!isLineValid(line_address) && isTransactionActive()) {
            issueSharedRequest(lineAddress);
            stallProcessor();
        }
    }

    foreach (line_address in getWriteAddresses()) {
        if (!isLineExclusive(line_address) && isTransactionActive()) {
            issueExclusiveRequest(lineAddress);
            stallProcessor();
        }
    }


    while(getPendingRequests() > 0) {} /* wait for any pending cache requests */

    baselineCommitTransaction();

    clearAllReadMarkBits();
}
```

7.3.4 Additions and Design Alternatives

This section discusses some of the design alternatives for DPTM.

Eager or Lazy Conflict Detection

DPTM detects transactional conflicts on changed values rather than on coherence conflicts. Because value changes are observable only once a transaction commits, DPTM's conflict detection is *lazy*, unlike the baseline which uses eager conflict detection. Because both eager and lazy conflict detection have been shown to be superior under different conditions [Ceze et al., 2006; Bobba et al., 2007; Minh et al., 2008; Shriraman et al., 2008; Tomić et al., 2009], I introduce *SendSets*, an optional addition to DPTM that allows eager conflict detection while retaining the benefits of lazy conflict detection.

SendSets leverages the information in a transaction's read and write sets, as represented by the read mark bits and the write buffer. When a transaction in DPTM issues a cache request using *SendSets*, it includes with the request this information as a bit map.

The transaction receiving the request determines whether there could be a transactional conflict by using the read and write set information in the bit map. If it determines that there may be a conflict, it decides whether to deny the request or acknowledge it based on the transactions' priorities.

Even if the transaction acknowledges the request, it does not abort, anticipating that the request is due to a silent store, or that the sender of the invalidation request might be a doomed transaction. This allows the transaction to eagerly detect conflicts and prioritize requests accordingly, while benefiting from the laziness of value-based conflict detection.

Improving Prediction Accuracy by Sending Updates

Because DPTM speculates using stale cache line data, the accuracy of such predictions can increase if the values in the stale cache lines are kept up to date. When a processor requests exclusive permissions to write to a cache line, it could send the value it intends to write along with its request [Huh et al., 2004]. The receiver, upon receiving this request, would update the value of its now stale cache line.

Other alternatives might go even further; for example, processors could keep track of other processors they have invalidated, and broadcast any subsequent updates of the cache line to those processors [Huh et al., 2004].

7.3.5 DPTM Architecture

DPTM is compatible with existing best-effort hardware transactional memory proposals that associate transactional conflicts with coherence conflicts. DPTM is also compatible with any cache coherence protocol with states denoting cache lines that are not present, present but invalid, and valid (e.g., the protocol described in Section 2.2).

DPTM does not require modifications of the coherence protocol typically used in hardware transactional memory proposals, and adds only minor, processor-local, hardware. The additional hardware requirements are as follows.

DPTM requires additional bits per transactional cache line for the read mark bits, and assumes the ability of instantaneously *flash-clearing* all the read mark bits. Flash-clearing is desirable for performance, but not required for correctness. If these bits cannot be flash-cleared, they could be cleared sequentially, at the cost of either stalling while the bits are cleared, or potential false conflicts in future transactions for the locations whose bits are still set.

DPTM also requires the ability to validate cache lines that are part of a transaction's read set against incoming data. The incoming data could be buffered in a *miss status handling register* (MSHR) [Kroft, 1983] while the validation takes place. DPTM also requires logic to compare the values being validated, but could be designed to leverage existing logic in a processor.

As for some of the design alternatives, *SendSets* requires the ability to include the read and write sets as a payload with cache requests. Sending updates requires the ability to add the value being stored as an extra payload to exclusive cache requests and invalidations. Including additional payload with cache messages does not change the coherence protocol, at least not as far as the protocol's states or transitions are concerned [Bobba et al., 2008].

7.4 Evaluation

This section reports on the analysis of DPTM. It presents an evaluation of different design alternatives for DPTM and compares them with a best-effort hardware transactional memory proposal, focusing on false sharing.

7.4.1 Experiment Environments

I use the simulation framework described in Section 2.8. The simulation framework is based on Virtutech Simics 2.2.19, in conjunction with the University of Wisconsin GEMS 2.1 memory models. The simulator can model processors that have best-effort hardware support, using Sun's Rock-like ATMTP simulator.

The parameters used in the simulation are shown in Table 7.1.

ATMTP is used to simulate the baseline hardware transactional memory proposal. The simulator parameters are set so that most transactions commit successfully in hardware in the absence of contention.[2] ATMTP's conflict resolution policy is set to a timestamp-based priority policy, where the requester wins only if its transaction is older, instead of the default

[2]For example, the size of Rock's transactional write buffer is 32 entries, which is also the ATMTP default. This is sufficient for all transactions in all benchmarks, except for `labyrinth`, which requires a 256 entry write buffer.

Table 7.1: *Simulated machine configuration*

Item	Model
Processor	in-order, single-issue, single-threaded, multicore
Cache line size	64 bytes
L1 cache	128 KiB, 4-way set-associative, 1 cycle latency
L2 cache	2 MiB, 8-way set-associative, 20 cycle latency
Physical Memory	8 GiB, 450 cycle latency
Processor Network Topology	point to point
ATMTP Specific	
Transactional write buffer size	32 entries, 256 for `labyrinth`
Conflict resolution policy	timestamp
Function calls in transactions	allowed

requester-wins. This reduces the number of transactions that abort in the presence of conflicts. The goal of this evaluation is not to study how the limitations of a Rock-like system might affect transactions, but to see how a good best-effort hardware transactional memory, which can commit most of its transactions in hardware, behaves in the presence of false sharing.

To simulate DPTM, I extended ATMTP without modifying its cache coherence protocol. Because ATMTP models a best-effort hardware transactional memory, a single global lock is used as a fallback mechanism when unable to complete a transaction in hardware. Neither NZSTM nor any other software transactional memory are used as a fallback mechanism; the metadata such systems add to each object's header acts as cache line padding, thereby mitigating the effects of false sharing. Although this might be a welcome side effect when using NZSTM, the goal of this evaluation is to analyze how well DPTM itself mitigates the effects of false sharing in the absence of other mitigating factors.

7.4.2 Benchmarks

This evaluation uses STAMP, SPLASH-2, microbenchmarks, and my own *SharingPatterns* benchmarks.

I test the STAMP benchmarks presented in Table 7.2 using the parameters suggested by their authors [Minh et al., 2008].

Table 7.2: *STAMP parameters used in the evaluation*

Benchmark	Parameters
`genome`	`-g256 -s16 -n16384`
`intruder`	`-a10 -l4 -n2048 -s1`
`labyrinth`	`-i random-x32-y32-z3-n96`
`ssca2`	`-s13 -i1.0 -u1.0 -l3 -p3`
`vacation-high`	`-n4 -q60 -u90 -r16384 -t4096`
`vacation-low`	`-n2 -q90 -u98 -r16384 -t4096`
`kmeans-high`	`-m15 -n15 -t0.05 -i random-n2048-d16-c16`
`kmeans-low`	`-m40 -n40 -t0.05 -i random-n2048-d16-c16`

STAMP was written by transactional memory experts; its benchmarks exhibit no false sharing inside transactions. Although code written by non-experts is unlikely to be optimized for false sharing, I use these benchmarks, anyway, to investigate how DPTM behaves in the absence of false sharing.

The SPLASH-2 suite was not originally meant for evaluating transactional memory. However, Moore et al. [2006], who adapted some of the SPLASH-2 benchmarks for evaluating transactional memory, report that `raytrace` is particularly susceptible to false sharing; they showed that the padded version of `raytrace` is faster than the original version almost by a factor of two. Therefore, `raytrace` is included in this evaluation. A small image, *teapot*, is used as an input — the same input Moore et al. use.

The microbenchmarks, which are adapted from the Java-based DSTM, are the following. The `linkedlist` benchmark is a concurrent set implemented using a single sorted linked list. Each thread randomly chooses to insert, delete, or look up a value in the range of 0 to 255, with the *high* contention distribution of operations being 1:1:0 (insert:delete:lookup) and the *low* contention distribution of operations being 1:1:1. The `redblack` and `hashtable` benchmarks are also concurrent sets, implemented using a red-black tree and a chained hash-table.

Table 7.3 presents a qualitative summary, relative to the STAMP benchmarks, of each benchmark's runtime transactional characteristics: length of transactions (number of instructions), size of the transaction's read and write sets, and amount of contention.

Table 7.3: *Qualitative summary, relative to STAMP, of the benchmarks' runtime transactional characteristics [cf. Minh et al., 2008].*

Benchmark	Transaction Length	Read/Write Set Size	Contention
`genome`	medium	medium	low
`intruder`	short	medium	high
`labyrinth`	long	large	high
`ssca2`	short	small	low
`vacation`	medium	medium	low/medium
`kmeans`	short	small	low
`raytrace`	short	small	medium
`linkedlist`	medium	medium	high
`redblack`	short	medium	low
`hashtable`	very short	small	low

I have also created a group of benchmarks, *SharingPatterns*, to cover a range of sharing patterns. These benchmarks are not meant to represent realistic workloads, but to exaggerate these sharing patterns to better observe the behavior of DPTM in the presence of false and true sharing. The patterns *SharingPatterns* exhibits are the following.

Sharing followed by no sharing: All transactions start by incrementing a value residing on the *same* cache line, followed by incrementing 19 different lines. Because sharing occurs at the *beginning* of the transaction, eager conflict detection systems would detect

potential conflicts early, which increases the chances of triggering conflicts between transactions.

No sharing followed by sharing: All transactions start by incrementing values residing on 19 *different* cache lines, followed by incrementing the same line. Because sharing occurs at the *end* of the transaction, eager conflict detection systems would detect potential conflicts close to commit time, which reduces the chances of triggering conflicts between transactions.

Write sharing: All transactions increment values residing on the *same* 20 cache lines.

The sharing part of each of the three patterns above is tested with false sharing, where processors increment different values on the same cache line; and with true sharing, where processors increment the same value on the same cache line.

7.4.3 Experiments and Results

The benchmarks are compiled using GCC 3.4.6, with optimization set to level 3. The experiments in this section are performed by running 16 user threads on 16 cores. Because this evaluation is concerned with the effects of false sharing only in transactions, I measure the time spent in transactional workloads. The outcome of ten runs with pseudo-random perturbations is presented, where the physical memory latency is pseudo-randomly varied by up to four cycles [Alameldeen and Wood, 2003]. The error bars in the graphs represent the standard deviation.

The Effects of Different Sharing Patterns

Because DPTM is designed with the goal of mitigating the effects of false conflicts, the evaluation starts by looking at different patterns of false and true sharing. I expect DPTM to improve performance over the baseline in the presence of false sharing, and that its performance would be comparable in the case of true sharing. I also expect that as the amount of false sharing increases, the gap between DPTM's performance and the baseline would also increase.

These experiments use the *SharingPatterns* benchmarks, which present an exaggerated form of different sharing patterns. The benchmarks are run using the baseline hardware transactional memory, and using DPTM with conflict detection granularity levels, in bytes, of 4 (one word), 8, 16, 32, and 64 (one cache line, does not mitigate the effects of false sharing). In the presence of false sharing, I expect performance to improve the finer the granularity. Figure 7.2 presents the outcome of these experiments.

In the presence of false sharing, DPTM, with granularity of 4 to 32 bytes, always performs significantly better than the baseline (*Figure 7.2: a, c, e*). DPTM also performs at least as

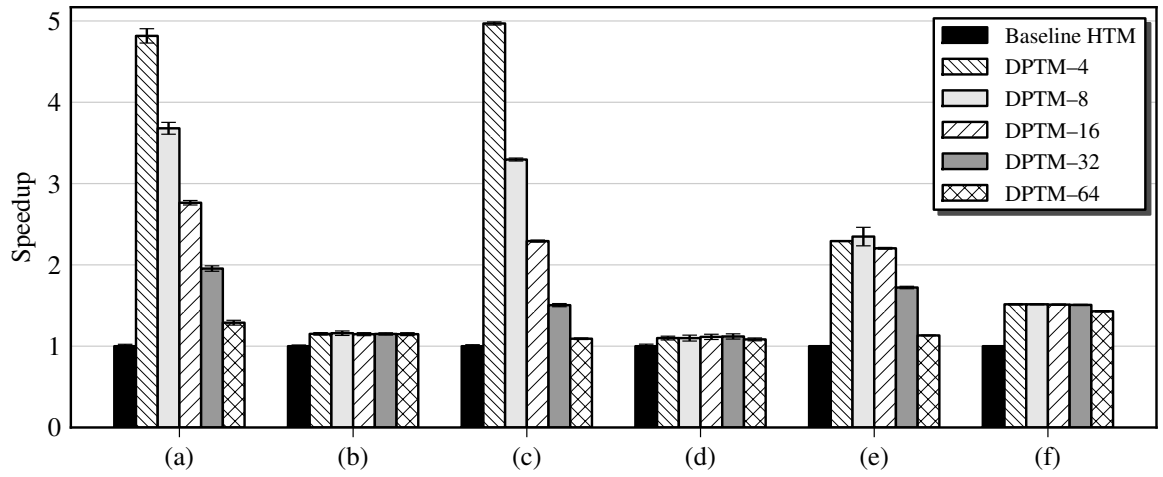

Figure 7.2: *The speedup of the* SharingPatterns *benchmarks running at granularities of 4 to 64 bytes, relative to the baseline. (a/b) false/true sharing followed by no sharing (c/d) no sharing followed by false/true sharing (e/f) false/true write sharing*

well, if not better, for true sharing (*b, d, f*). The improvement in the case of true sharing is an anomaly due to DPTM's *lazy* conflict detection, where its transactions are not aborted, as in the baseline, when a cache line is invalidated by a doomed transaction.

DPTM's gains are comparable whether false sharing occurs at the beginning of a transaction (*a*), or at the end (*c*). Because contention in these benchmarks is high, false sharing adversely affects performance in the baseline regardless of where it occurs inside a transaction. This shows that even one instance of false sharing can be detrimental to performance.

The improvement gained by finer granularity is more pronounced for the false sharing patterns that do not perform many shared stores (*a, c*), than where shared stores dominate (*e*). Moreover, where shared stores — and false conflicts — dominate, DPTM is less effective in mitigating their effects. This is due to DPTM's serialization of commit phases in the presence of false conflicts, which reduces the gains in such workloads.

The Effects of Restructuring

The previous experiments demonstrate that DPTM has potential for improving performance in the presence of false sharing. In the next set of experiments, I investigate how DPTM performs using a more diverse set of benchmarks. Figure 7.3 presents the outcome of these experiments.

I investigate the amount of false sharing present in these benchmarks. Where it is present, I try to mitigate it by padding, and where it is not, I try to instigate it by removing existing padding. This helps study the efficacy of using padding to mitigate false sharing, particularly when compared with using DPTM. I expect DPTM to be at least comparable to, if not better

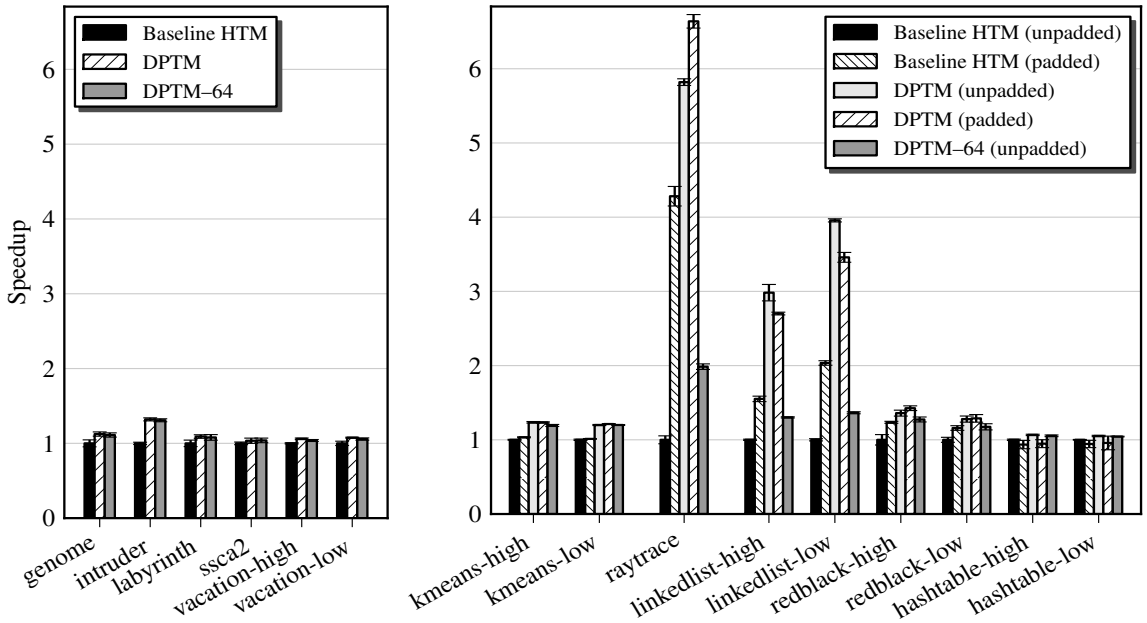

Figure 7.3: *The speedup of running the benchmarks, padded and unpadded where applicable, relative to the baseline. The benchmarks on the left do not exhibit false sharing inside transactions.*

than, padding in mitigating false sharing, because padding could have adverse effects on locality.

The STAMP suite was developed by transactional memory experts for evaluating transactional memory proposals, so I expect it would be optimized to mitigate false sharing. Analyzing its code shows that some of its structures are padded and aligned to cache line boundaries. The only padded data structures used inside transactions, as far as I could tell, are in `kmeans`. I created a modified version of `kmeans` with this padding removed. Even though I do not expect any significant performance gains, this evaluation still uses the remaining STAMP benchmarks to study other effects DPTM's changes might have in the absence of false conflicts.

Even though the SPLASH-2 suite was written for evaluating multiprocessors, `raytrace` is known to suffer from false sharing in the context of hardware transactional memory [Moore et al., 2006]. To compare against mitigating the effects of false sharing by restructuring, I created a modified version of `raytrace` in the manner Moore et al. [2006] describe.

The microbenchmarks were originally written in Java, and therefore are not padded. In porting them to C, I created two version: an unpadded version, and a version where each object is padded and aligned to the cache line boundary.

The performance of the padded versions is compared with the unpadded ones (where applicable), when running on the following: the baseline hardware transactional memory, DPTM with conflict detection granularity of 4 bytes (one word — *DPTM*), and DPTM with

conflict detection granularity of 64 bytes (one cache line — *DPTM–64)*. DPTM–64 is used to isolate the effects DPTM has on false sharing from other effects, such as DPTM's lazy conflict detection, because DPTM–64 detects conflicts at the granularity level of a whole cache line and therefore does not mitigate the effects of false sharing.

To better understand the impact of false sharing on the benchmarks' transactions, Table 7.4 presents an analysis of the impact false sharing has on the abort rate of transactions in DPTM–64. I present this analysis for DPTM–64 because it uses the same mechanisms as DPTM–4, yet detects conflict at the granularity level of a whole cache line. Therefore, DPTM–64 isolates the effects of false sharing in transactions from the effects of all the other changes DPTM makes to the baseline hardware transactional memory.

Table 7.4: *DPTM–64's abort rates, relative to the number of committed transactions, broken down by cause. The first column presents the percentage of aborting transactions that abort because of false sharing. The second column presents the percentage of transaction that abort because of true conflicts. These results are based on the unpadded version of the benchmarks (where applicable).*

Benchmark	False Sharing Aborts	True Conflict Aborts
`genome`	16%	18%
`intruder`	7%	104%
`labyrinth`	16%	300%
`ssca2`	1%	3%
`vacation-high`	19%	415%
`vacation-low`	15%	415%
`kmeans-high`	5%	18%
`kmeans-low`	5%	8%
`raytrace`	67%	28%
`linkedlist-high`	313%	251%
`linkedlist-low`	269%	156%
`redblack-high`	17%	9%
`redblack-low`	12%	6%
`hashtable-high`	2%	0%
`hashtable-low`	1%	0%

Table 7.4 shows that, to a certain extent, all benchmarks suffer from false sharing, even the ones where code inspection did not reveal any obvious cases of false sharing. In terms of effect on speedup, the two benchmarks that suffer the most from false sharing are the two that suffer the most in terms of abort rate: `linkedlist` and `raytrace`. More transactions in the `linkedlist` benchmark abort because of false sharing, but the `raytrace` benchmark is affected more in terms of speedup (Figure 7.3), because the relative increase in the number of aborted transactions is higher than in `linkedlist`.

The results of evaluating DPTM using STAMP show that DPTM is consistently faster than the baseline, with the exception of the `ssca2` benchmark, where DPTM and the baseline are comparable. This improvement is not due to the mitigation of false sharing or silent stores, but to DPTM's lazy conflict detection.

For the `kmeans` benchmark, padding improves performance only slightly. False sharing in `kmeans` does not occur often, because its transactions are small with low contention, and its transactional objects that suffer from false sharing are only slightly bigger than a cache line.

As for the SPLASH-2 benchmark `raytrace`, false sharing has a significant impact. Padding mitigates much of its effects. Furthermore, the cache miss statistics show that padding does not adversely affect locality in this case.

DPTM significantly speeds up `raytrace` compared with the baseline when running either the padded or the unpadded version. Most of this improvement is due to mitigating the effects of false sharing.

DPTM is faster when running the padded version compared with the unpadded version of `raytrace`. This shows that DPTM was able to mitigate most of the effects of false sharing, but not eliminate them altogether. DPTM serializes the commit phases of transactions with false conflicts, whereas the padded version has no false conflicts that cause commit phases to serialize.

For the microbenchmarks running on the baseline hardware transactional memory, the padded versions of the `linkedlist` and `redblack` benchmarks are significantly faster, because padding mitigates false sharing. As for `hashtable`, there is no significant difference because contention is low and its transactions are too short for false sharing to have much of an effect.

DPTM running the unpadded microbenchmarks is faster than the baseline running either the unpadded or the padded versions. Interestingly, in the `linkedlist` benchmark, DPTM performs better using the unpadded version whereas the opposite is true for the baseline: DPTM was able to mitigate the effects of false sharing as well as take advantage of the locality afforded by the use of the unpadded version.

An interesting observation regarding value prediction is that the results presented so far use a conservative DPTM, which speculates only on the values already part of a transaction's read set. An investigation of the results shows that most of the gains so far, even in the presence of false sharing, are due to the decoupling of transactional from coherence conflicts: value prediction accounts for less than 5% of the *additional* gains presented.

I also experimented with an aggressive approach, which *always* speculates on the values in a stale cache line, with the results (not shown) consistently slower. This is not surprising, because mis-prediction has the high cost of aborting the whole transaction, something that was also experienced in the Rock evaluation (Chapters 5 and 6). Others have also observed that for data speculation to be effective, it should be throttled to avoid the high cost of mis-prediction [Steffan, Colohan, Zhai, and Mowry, 2002].

The Effects of Design Alternatives

Section 7.3 discussed some of DPTM's design alternatives. This section evaluates these alternatives against the basic DPTM with conflict detection granularity of 4 bytes. Figure 7.4 presents the outcome of these experiments.

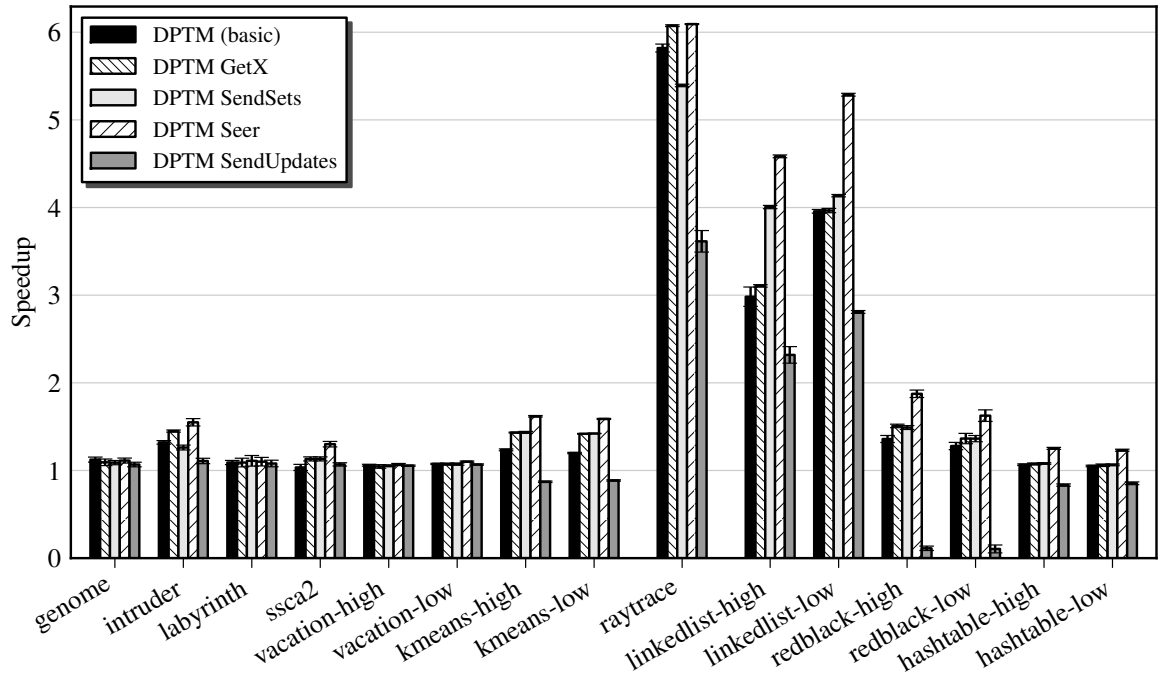

Figure 7.4: *The speedup of design alternatives for DPTM relative to the baseline (not shown)*

First, I evaluate issuing exclusive requests immediately on stores (*GetX*), rather than waiting until commit time. Issuing the requests immediately can hide the latency for the request. However, if contention is high, the transaction may lose exclusive permissions and need to reissue the request later, further increasing contention.

The evaluation shows no significant differences except in the `intruder`, `kmeans`, `raytrace`, and `redblack-high` benchmarks, where *GetX* is noticeably faster. These benchmarks have short transactions, so the window for losing exclusive permissions is small. On the other hand, `hashtable` is not benefiting much from *GetX*, because its transactions are very short, so there is little time to hide any request latency.

Because *GetX* performs as well as, if not better than, the basic DPTM, the remaining experiments also issue exclusive requests immediately on stores, and are compared against *GetX*.

Next, I evaluate *SendSets*, which makes DPTM more eager by taking advantage of a cache line's read and write set information in detecting transactional conflicts. As mentioned previously, eager and lazy conflict detection are superior under different conditions; therefore, I expect *SendSets*'s performance to be in line with more eager conflict detection schemes.

For the benchmarks used, *SendSets*'s performance is generally comparable with *GetX*, except for the `intruder`, `raytrace`, and `linkedlist-high` benchmarks. There is more improvement in `linkedlist-high`, a long benchmark with high contention, because eager conflict detection aids higher priority transactions (older transactions in this case). On the other hand, `intruder` and `raytrace` are slower, because requests are often denied by doomed transactions, leading to wasted work.

Finally, I investigate ways to improve the accuracy of value prediction in DPTM. I first model the instantaneous and free broadcasting of all cache line updates to all processors that have the cache line in a stale state (*Seer*). This *unrealistic* approach evaluates the effects of such a broadcasting mechanism regardless of its cost, in order to gain intuition about the benefits of sending updates in general. I then compare *Seer* against the more realistic approach of sending updates only with invalidation requests (*SendUpdates*).

When using either *Seer* or *SendUpdates*, DPTM *always* speculates on stale cache line data, unlike the conservative basic DPTM. Out of all experiments and all parameters, *Seer* performs the best.

The gains from value prediction when using *Seer* are substantially higher than in the basic DPTM. For example, in `ssca2`, value prediction now accounts for 50% of the additional gains over the baseline, 40% in `redblack`, and 30% in `kmeans` and `linklist`. This unrealistic approach demonstrates that broadcasting updates has potential for improving performance gains from value prediction.

On the other hand, *SendUpdates* overall performs poorly; transactions frequently abort because of mis-speculation, especially in `redblack`. This implies that although the sending of updates seems promising, *SendUpdates* is not sufficient to capture this potential.

Finally, to better appreciate the effect the different aspects the design of DPTM has on performance, Figure 7.5 presents a breakdown of the different speedup components in *Seer*. I present the breakdown for *Seer* because its component parameters perform the best across all workloads in these experiments. Therefore, a breakdown of its components is well suited to examine the relative benefits of the different design options investigated.

7.4.4 Discussion

For the workloads in this evaluation, DPTM consistently outperforms the baseline system. This is due to DPTM's mitigation of the effects of false sharing and to DPTM's lazy conflict detection.

Because most of the workloads in the evaluation were developed by experts in parallel programming, and therefore optimized to prevent false sharing, not many benefit from DPTM's mitigation of the effects of false sharing. Those that do, however, benefit immensely — even more than they benefit from restructuring by padding. The evaluation demonstrated that even a single instance of false sharing can be detrimental to performance. The evalua-

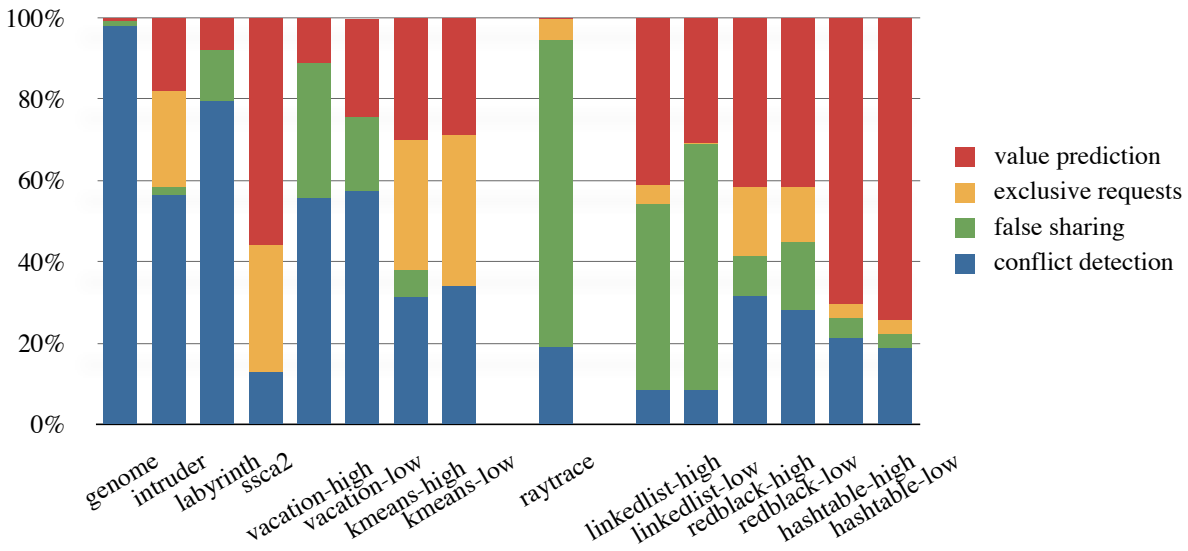

Figure 7.5: *A breakdown of the speedup components of* Seer. *The conflict detection component is due to the difference in DPTM's conflict detection from the baseline hardware. The false sharing component is due to the transactions not aborting because of false sharing. The exclusive requests component is due to issuing exclusive requests immediately on stores, rather than waiting until commit time. The value prediction component is due to the reduction in load latency when speculating on data values.*

tion also demonstrated that restructuring by padding, in addition to the software engineering challenges it poses, could also hurt performance by undermining data locality.

This evaluation assumes a 64 byte cache line, which is the cache line size for Sun's Rock processor. However, it seems that the current trend in cache line size favors larger cache lines.[3] Larger cache lines can fit more data and hold more distinct data objects; therefore, I expect the problem of false sharing to be exacerbated by larger cache lines, and in turn, that the benefits reaped from the techniques proposed in this chapter would be greater.

The other aspect of DPTM is its ability to reduce stalling by using value prediction. As others have observed, taking advantage of value prediction can be difficult [Steffan et al., 2002]. The difficulty lies in that mis-prediction is expensive because it leads to aborted transactions, which could negate the benefits gained from many instances of successful prediction. The unrealistic *Seer* design alternative demonstrated that value prediction can be very beneficial. However, further work is needed to find better heuristics for deciding when to predict, and for choosing the values to use for prediction.

[3]For example, Sun's first Niagara processor's cache line size was only 16 bytes [Kongetira, Aingaran, and Olukotun, 2005].

7.5 Related Work

Other hardware transactional memory proposals are also capable of fine-grained conflict detection, e.g., *TCC* [Hammond, Wong, Chen, Carlstrom, Davis, Hertzberg, Prabhu, Wijaya, Kozyrakis, and Olukotun, 2004; McDonald, Chung, Chafi, Minh, Carlstrom, Hammond, Kozyrakis, and Olukotun, 2005], *Bulk* [Ceze, Tuck, Torrellas, and Cascaval, 2006], and *RETCON* [Blundell, Raghavan, and Martin, 2010]. TCC associates fine-grained read bits for conflict detection with each cache line; however, TCC proposes an unconventional approach for memory consistency and cache coherence, where transactions are the basic unit of parallel work. Bulk hashes a transaction's access information, and uses this hash for conflict detection. RETCON, which is concurrent with this work, symbolically tracks modifications and constraints that a transaction applies to variables, and uses value-based conflict detection. In contrast to these proposals, I directly address the problem of false sharing using best-effort hardware with a conventional coherence protocol. I also propose using value prediction to reduce latencies incurred with false sharing, by speculating on the values of stale cache line data.

Concurrently with and independently of this work, Pant and Byrd [2009] proposed using value prediction in hardware transactional memory, in a manner very different from mine both in design and in purpose. Their proposal does not address false sharing in transactions, but uses value prediction to reduce load latencies by predicting future updates. Their proposal also requires extensive modifications to the underlying hardware: the value predictor is located at the memory level, near the memory or directory controller; the directory must be able to observe all stores, which could be a bottleneck; the number of transactions concurrently involved in value prediction is limited; additional hardware modifications to the directory are needed to keep prediction history and make further predictions; and a specific type of cache coherence protocol is required (a *nacking* protocol, modeled on the one Moore et al. [2006] use), as well as further changes to that protocol.

Outside the context of transactional memory, Huh, Chang, Burger, and Sohi [2004] decouple the use of a cache line's data from obtaining permissions for that line to mitigate the effects of false sharing. They propose speculating on the values of stale cache lines, as well as mechanisms of sending write updates and forwarding modified data. In their work, latency tolerance for the coherence permissions was low, effectively limited to the parallelism in the existing reorder buffer. Hardware transactional memory naturally provides a larger scope, and so is better suited for coherence decoupling. Moreover, the additions required for DPTM can also be used to mitigate the effects of false sharing *outside* transactions, in the same manner Huh et al. propose.

Value prediction has also been explored in the context of thread-level speculation in works by Knight [1986], Akkary and Driscoll [1998], Martin, Sorin, Cain, Hill, and Lipasti [2001], Cintra and Torrellas [2002], and Steffan, Colohan, Zhai, and Mowry [2002], among others.

EazyHTM [Tomić, Perfumo, Kulkarni, Armejach, Cristal, Unsal, Harris, and Valero, 2009] is a hardware transactional memory that separates conflict detection from conflict resolution, allowing it to detect conflicts eagerly, but act on them lazily. The work on *SendSets*, which was concurrent with EazyHTM, serves a similar purpose using a different technique.

On the software side, Torrellas, Lam, and Hennessy [1994] propose some solutions to the false sharing problem using compiler modifications that optimize the layout of shared data in cache lines to mitigate its effects. Olszewski, Cutler, and Steffan [2007] propose a software transactional memory that uses value-based conflict detection, and is capable of improving performance in the presence of silent stores.

7.6 Concluding Remarks

This chapter demonstrated how data speculation in hardware transactional memory, by example of DPTM, has the potential for improving performance, particularly in the presence of false conflicts. Benchmarks that exhibit false sharing show dramatic gains, whereas ones that do not exhibit false conflicts are not harmed by the alternative designs — and some even benefit from lazy conflict detection.

The evaluation showed that DPTM can mitigate the effects of false sharing inside transactions. The modifications DPTM needs could also be applied to mitigate the effects of false sharing *outside* transactions, in the same manner proposed by Huh et al. [2004], thus improving performance over a wider range of workloads.

Although DPTM can significantly mitigate the effects of false sharing, this mitigation is not perfect, because it serializes the commit phases of transactions with false conflicts.

DPTM can also improve performance in the presence of silent stores. Others have noted that silent stores are common in certain workloads [Steffan et al., 2002; Cintra and Torrellas, 2002]; however, they were not common in the workloads tested in this chapter. This might be because most of the benchmarks used were developed by experts for evaluating transactional memory, and who are aware that silent stores could adversely impact certain transactional memory proposals.

On some of the more recent multicore processors, cache coherence costs are low as long as the coherence traffic remains within the same chip [Dice and Shavit, 2010]. Moreover, with the more recent and faster processor interconnects, such as Intel's QPI [Maddox, Singh, Safranek, and Colwell, 2009], coherence costs even between processors are decreasing. Decreasing coherence costs means that the detrimental effects of false sharing and silent stores are also reduced. However, even if the coherence cost of false conflicts is practically zero, transactional memory that associates transactional conflicts with coherence conflicts still runs the risk of serializing or aborting whole transactions in the presence of false conflicts.

Chapter 8

Conclusion

This thesis presented a case for hybrid transactional memory, arguing that if transactional memory, as a programming model, is going to become a mainstream model, then it would require both hardware and software support. This is because the alternatives, whether they are purely in hardware or purely in software, do not appear to be viable in the near future. A hardware transactional memory solution that can truly stand alone is too complex for processor manufacturers to consider, whereas software transactional memory's performance is lacking. This leads to an impasse where programmers are not using the available (software) transactional memory because it is too slow, and hardware manufacturers do not add hardware support because they do not believe that transactional memory is a popular model.

To support the case for hybrid transactional memory, this thesis presented the results of my research and contributions to this field. These contributions cover different areas of transactional memory, from software to hardware to a hybrid combination of the two. This thesis also presented a study of using transactional memory in real-world applications.

My research introduced the first nonblocking zero introduction software transactional memory (NZSTM). NZSTM demonstrated that nonblocking algorithms are not inherently slower than their blocking counterpart. With little transactional hardware support, nonblocking algorithms can be almost as simple as blocking ones — true to the spirit of transactional memory as originally envisioned by Herlihy and Moss.

My research also introduced a hybrid system, NZTM, which is based on NZSTM. NZTM takes advantage of the lack of indirection in NZSTM to access the data directly in the common case. This thesis presented an evaluation of NZTM using a simulator and using Rock, one of the few processors that supports the hardware primitives that NZTM requires. Even with the restrictions Rock imposes on its hardware transactions, the evaluation shows that some workloads benefit significantly from hybrid support.

Because CPython uses a single global lock to protects its data structures, I thought it would make an interesting case study on the capabilities of Rock's transactional support for improving the performance of a real-world platform. A preliminary evaluation showed that

Rock's hardware support has the potential to make CPython scale with the number of cores, with only a few simple changes to its code.

The final contribution presented in this thesis relates to the hardware level of support for transactional memory. False sharing, at the cache line level, in hardware transactions can undermine the performance of the transaction abstraction. By leveraging the speculation inherent in hardware transactions, my proposed system can mitigate most of the effects of false sharing.

* * *

Transactional memory was first proposed in 1993, yet it is still not a mainstream programming model. Many use this as an argument against the transactional model. However, I do not believe that this undermines the transactional model: transactional memory presents a paradigm shift in the way programmers think about their programs. In the past, other new models have also faced resistance before being accepted.

For example, virtual memory, which is ubiquitous today, initially encountered some resistance. Even though Sayre demonstrated that virtual memory's performance is superior to manual memory management back in the 1960s, many popular operating systems, such as Microsoft DOS in the 1980s and the 1990s, still did not support virtual memory. Another model met with resistance is object oriented programming, also first proposed in the 1960s. Even the Turing Award winner, Edsger Dijkstra [1994], decried object oriented programming as "snake oil" with a "most impressive name". Object oriented programming did not become mainstream until the 1990s, with the rising popularity of languages such as C++, Java, and Python.

The field of software engineering is more mature now than it was in the last century. Therefore, it is not surprising that a new, and arguably unproven, programming model would encounter much resistance.

Even when, or if, transactional memory is accepted into the mainstream, I do not believe it will be the last word on parallel programming models. It is also unlikely that it will supplant the existing models — the almost forty year old C programming language has not been supplanted by some of the newer, and arguably better, programming languages: C is *still* the most popular programming language[1].

In light of this, the contributions presented in this thesis might ease the adoption of this new programming model. In particular, NZTM, which closes some of the performance gap between software and hardware, might appeal to programmers who need the fault tolerance and reliability of nonblocking algorithms. Furthermore, demonstrating the value of decoupling transactional conflicts from coherence conflict, and using data speculating inside transactions, contributes to strengthening the transaction abstraction, thereby making it more appealing for software engineers.

* * *

[1] According to the TIOBE Programming Community Index for May 2010 [TIO, 2010].

In my opinion, one of the biggest challenges the transactional memory community faces today is not a technical one, but a battle for the hearts and minds of programmers. Our community needs to demonstrate that transactional memory can help improve the design process and performance of real programs, and not just an arbitrary set of benchmarks. Showing that Rock's best-effort transactional memory can benefit CPython is a step in that direction; however, that effort is limited to one specific platform.

I believe that the challenge is to improve the performance of software transactional memory systems well enough so programmers would consider the performance impact to be a worthwhile tradeoff with the ease of programming transactional memory promises. Once programmers appreciate the value of the transactional model, then hardware manufacturers might add support for transactions the same way they added support for machine virtualization after it had gained popularity (Section 3.2.3).

This thesis has argued for hybrid transactional in general, but my research was focused on an approach that uses best-effort hardware transactions, when available, and falls back on software transactions when the underlying hardware is incapable of committing the transaction. I do not believe that this approach to hybrid transactional memory is necessarily the best approach, mainly because once a transaction aborts in hardware and falls back on software, then the hardware transactional support is not being leveraged any longer (unless the hardware is also used to simplify the software algorithm).

As explained in Chapter 3, there are other approaches to hybrid transactional memory. Others have suggested using both hardware and software techniques simultaneously to support transactions. It is not clear, however, which one of these myriad of approaches might be best, or if there might even be a better way of adding hardware support that has not been proposed yet. Researching and evaluating different hardware primitives that could support transactional memory is, in my opinion, one of the key remaining issues in hybrid transactional memory.

> *Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.*
>
> *— Winston Churchill*

Appendix A

Dynamic Software Transactional Memory

This appendix describes the algorithm for Dynamic Software Transactional Memory (DSTM) [Herlihy, Luchangco, Moir, and Scherer, 2003b]. NZSTM inflates objects owned by unresponsive transactions into DSTM-like objects, and uses the DSTM algorithm to acquire these inflated objects (Section 4.1.4). The DSTM algorithm described is the one used in my implementation of NZSTM, and is based on the description given by Herlihy et al. [2003b].

DSTM is a nonblocking software transactional memory proposal, and is the first *dynamic* software transactional memory proposal. Earlier software proposals [e.g., Shavit and Touitou, 1995] required transactions to *statically* specify the data they are going to access *before* they can run. This requirement limits the usability of such proposals for algorithms that cannot determine this information in advance. For example, a non-dynamic transaction would be restricted when attempting to traverse a tree structure, because it cannot determine the sequence of nodes its going to access before the transaction. DSTM, on the other hand, makes no such restrictions.

DSTM was initially implemented in the Java programming language. The DSTM code I had access to, which was used as a reference when developing NZSTM, is the Java DSTM code.

A.1 DSTM Data Structures

The basic data structures DSTM uses are shown in Figure A.1.

The `TMObject` data structure encapsulates a program object that DSTM transactions can access, and serves as a container for its data and metadata. It contains a `start` pointer, which points to a `Locator` object, and must be modified only atomically.

Each `Locator` object is composed of three fields. First, the `Owner` pointer, which points to the transaction that created this `Locator`. The `new data` and the `old data` pointers

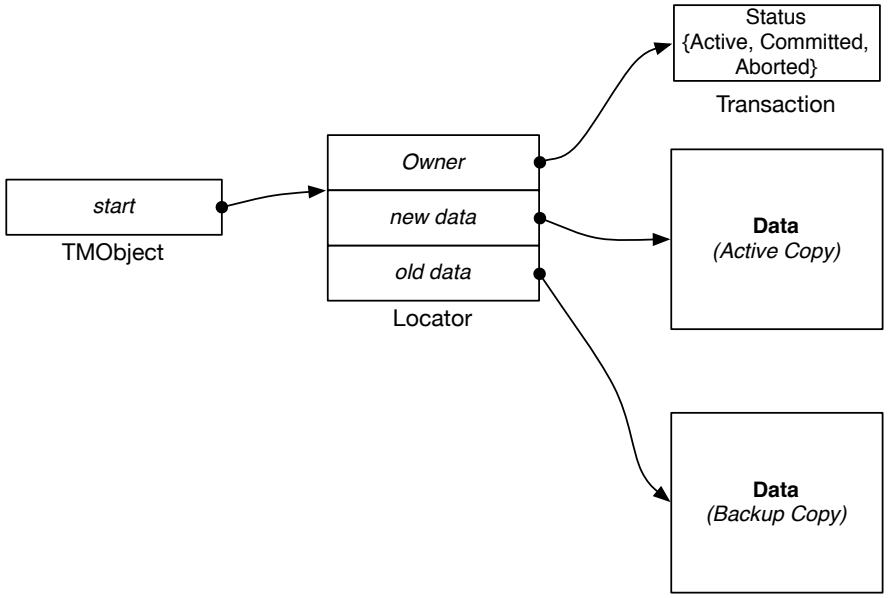

Figure A.1: *The structure of DSTM's main transactional object*

point to the actual object data. The `old data` pointer points to the backup copy of the object data, which is interpreted as the object's current data for aborted transactions. Initially, `new data` points to a *copy* of the old data; then the current active transaction accesses and modifies the data that `new data` points to. The data that `new data` points to is interpreted as the object's current data for committed transactions.

DSTM adds a level of indirection to access the `Locator` because, as will be explained later, transactions that acquire objects must modify all three fields of an object's `Locator` atomically. Most modern hardware allows only one word to be modified atomically. Therefore, by making the `Locator` accessible only through this (one word) `start` pointer, the algorithm can atomically change the `Locator` just by atomically swapping the `start` pointer, e.g., using *Compare&Swap*.

DSTM's `Transaction` object is the model used for NZSTM's `Transaction`. DSTM creates a new `Transaction` object for every new transaction and does not reuse these objects regardless of whether a transaction commits or aborts. Even when DSTM attempts an aborted transaction again, it creates a new `Transaction` object for the new attempt.

Each `Transaction` object contains the transaction's `Status` field, which must be modified only atomically, and can be in one of the following states: `Active`, which indicates that the transaction is currently running; `Committed`, which indicates that the transaction has committed successfully; and `Aborted`, which indicates that the transaction has been aborted.

A.2 DSTM Algorithm

Inflated NZSTM objects are accessed exclusively, i.e., no read sharing is allowed. Therefore, this appendix describes only the exclusive DSTM algorithm.

In DSTM, a thread begins a transaction by creating a new `Transaction` object with its status set to `Active`. The thread then executes the transaction, acquiring each object it accesses. When the thread completes the execution of the transaction, it attempts to atomically change its transaction's status from `Active` to `Committed`.

To acquire an object, a DSTM transaction, T, first determines if it has already acquired the object by following the object's `start` pointer to its `Locator`, and examining it `Owner` field. If the `Owner` points to the current `Transaction`, then it has already acquired the object within the same transaction.

If T has not already acquired the object, then T must ensure that there are no conflicts with other transactions before acquiring ownership of the object. If the `Owner` field points to a committed or an aborted transaction, there is no conflict. Consequently, T creates a new `Locator`. T sets the `Owner` of the new `Locator` to point to its transaction, sets the `old data` field to point to the object's current data, which is the `new data` of the current `Locator` if the last owner is a committed transaction, or the `old data` of the current locator if the last owner is an aborted transaction. T then attempts to atomically swap the `start` pointer to point to its newly created `Locator` (Figure A.2). If the swap is successful, it means it has successfully acquired the object. If not, it means that another transaction acquired the object instead, and T tries again. Finally, T checks whether it has been aborted by checking its own `Transaction` status. If it has been aborted, then it restarts the transaction. If T is still active, then it has successfully acquired the object.

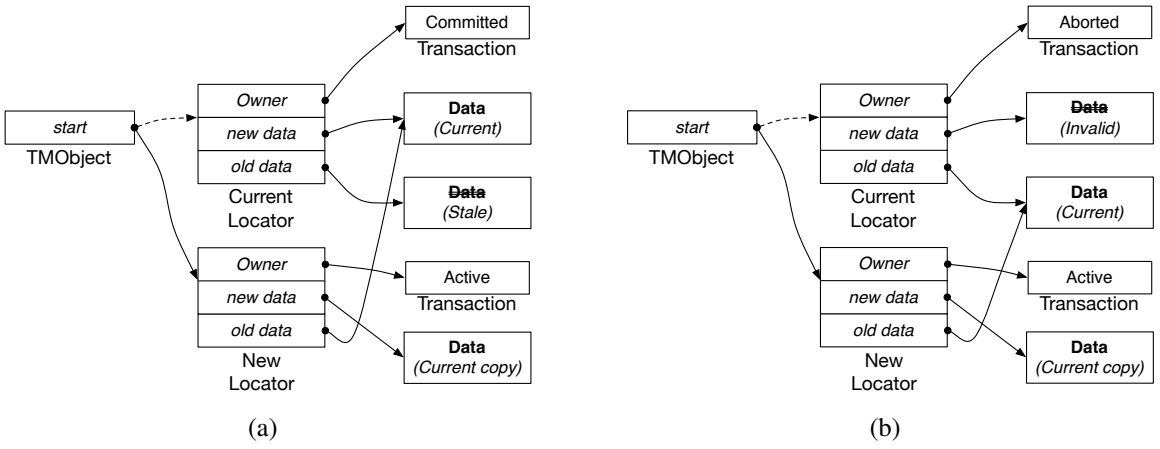

Figure A.2: *A DSTM transaction acquiring an object most recently owned by (a) a committed transaction, or (b) an aborted transaction.*

However, if the `Owner` field points to an active transaction, there is a conflict. The transaction, T, consults an out-of-band contention manager, which may apply different policies. Depending on the decision of the contention manager, T either waits and then tries to acquire the object again, or it aborts.

Should a DSTM transaction decide to abort another transaction, it does so by atomically attempting to change its `Transaction` status from `Active` to `Aborted`. In the case of the original Java implementation, as well as the implementation used in NZSTM, this is done using a *Compare&Swap* instruction. The code listing below demonstrates how DSTM aborts a transaction.

```
void abortTransaction(Transaction *enemyTransaction)
{
   atomic {
      /* can be performed using Compare&Swap */
      if (enemyTransaction->status == Active) {
         enemyTransaction->status = Aborted;
      }
   }

   /*
    * If the swap fails, it means that the transaction has either committed or aborted.
    * In either case, no further action is necessary.
    */
}
```

To commit a transaction, a thread attempts to atomically change its `Transaction` status from `Active` to `Committed`. If it fails, it means it has been aborted, and so it restarts the transaction by creating a new `Transaction` object and runs the code again.

Appendix B

NZSTM Promela Model

This appendix contains the source code for the Promela model used for model checking NZSTM (Section 4.2.1). A concise Promela primer can be found at the Spin website [Gerth, 1997]; alternatively, refer to Holzmann [2003] for a more complete reference.

I present NZSTM's Promela source code instead of the code for the C implementation. The Promela model is a high-level description, which abstracts away much of the architectural-dependent details and makes it easier to read than the C code.

The model presented covers the nonblocking and the blocking versions of NZSTM, using both the exclusive algorithm and the read sharing algorithm. The model also covers the DSTM2 Shadow Factory algorithm used in Chapter 4. Some of the techniques used in this model were inspired by Ananian [2007].

```
/**
 * @file nzstm.pml
 * Promela model for NZSTM
 *
 * @author Fuad Tabba (fuad at cs.auckland.ac.nz)
 *
 * Some ideas taken from C.S. Ananian's Promela model:
 *
 * C. S. Ananian. Architectural and compiler support for strongly atomic
 * transactional memory. Ph.D. thesis, Massachusetts Institute of Technology,
 * 2007.
 * http://flex-compiler.csail.mit.edu/Harpoon/swx.pml
 *
 * The interface of this TM and the structure of the locator is based on DSTM:
 *
 * M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software
 * transactional memory for dynamic-sized data structures. In PODC '03:
 * Proceedings of the twenty-second annual symposium on Principles of
 * distributed computing. ACM, 2003b.
 *
 * The DSTM2 Shadow Factory model is based on the description in:
 *
 * M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing
 * software transactional memory. In OOPSLA '06: Proceedings of the 21st annual
 * ACM SIGPLAN conference on Object-oriented programming systems, languages, and
 * applications. ACM, 2006.
 *
 * Goals:
 *     - Only use atomic statements (atomic, d_step) in scenarios where the
 *       operation can be made to look atomic in C using nothing more than CAS
 *       (except when modeling SCSS)
 *     - Only one process can modify an object at a time
 *     - Multiple processes can read an object at a time
 *     - Consistency must be preserved
 *     - All threads must reach the end of the code (no deadlocks)
 *     - Livelocks within the algorithm itself are not allowed. Meaning each
 *       transaction must eventually either commit or abort. Livelocks in the
 *       larger sense are possible though (i.e. everyone aborting), and that is
 *       prevented using good contention management.
 *
 * Current Issues and Bugs:
 *      - none known
 */


/*
 * ================================================================================
 * Global Constants
 * ================================================================================
 */

/*
 * If defined uses the DSTM2 Shadow factory, otherwise its NZSTM
 */
/* #define SHADOW */

/*
 * If defined performs the increment/decrement same object test.
 *
 * This nondeterministically open two different objects and either:-
 *  - increments and decrements them, with the sanity check being that after
 *   incrementing the value must be 1 greater than the initial value
   (indicating that only one thread was successful in incrementing the object).
 *
 *  - reads the value of the object to check that it is the initial value. Since
 *   all writers increment then decrement the object, its value must stay at
 * the initial value.
 */
/*#define INCDEC_TEST */

/*
 * If defined performs the balancing test.
 *
```

```
 * This test divides the threads into two groups, one group subtracts from one
 * object and adds the same amount to the other object. The other group
 * reads both objects and checks that their combined value is preserved.
 */
/* #define BALANCE_TEST */

/*
 * If defined performs the add around test.
 *
 * This test has three objects. It reads the first value and the second value,
 * Adds this to the third value, then adds this total to the first again. All
 * the while comparing the values with what it expects them to be.
 *
 * The idea here is to go back and fourth between values, alternating between
 * opening for read and opening for write. Create some room for concurrency
 * and unlike the other tests, have invariants that change the state of the
 * system.
 *
 * NOTE: This test needs one more process to be defined since there's an init
 * process to setup the system. Not so good in terms of state-space efficiency.
 */
/* #define ADD_AROUND_TEST */


/* If defined then visible reads are used, otherwise it's exclusive reads */
#define READ_VISIBLE


/* If defined then the blocking model is used */
/*#define BLOCKING*/


/*
 * If defined WITH blocking, then the blocking algorithm with SCSS is used, which models
 * nonblocking by hardware support
 */
/* #define SCSS */

/**
 * Number of processes running in the system.
 * Remember the initializing process if any (e.g., for AddAround).
 */
#define PROCESSES    3

/**
 * Maximum number of transactions the system can create
 *
 * Must have one extra transaction since a txn identifier of 0 is like
 * a NULL pointer. In practice, 0 points to a committed transaction that
 * all newly created objects point to.
 */
#define TOTAL_TXN    10

/**
 * Number of objects to preallocate.
 */
#define TOTAL_OBJECTS 3

/**
 * Maximum number of transactions per thread
 */
#define TXN_PER_PROC ((TOTAL_TXN - 1)/PROCESSES)

/*
 * Stuff to help with debugging and validating
 */

/**
 * A number that represents an invalid state. i.e. uninitialized memory
 *
 * To ensure it's never used, must be bigger than all arrays. That way, any
 * errors would be detected by spin's bound checking (indexing error).
 */
```

```
#define INVALID 77

/**
 * Represents NULL pointers. Just to make it clearer that the value we're
 * assigning NULL to represents a pointer rather than a proper value.
 */
#define NULL 0

/**
 * The value the data in the objects is initialized to.
 */
#define INITV 13

/*
 * ============================================================================
 * Structure Definitions
 * ============================================================================
 */

/**
 * This data structure represents the pointer to the oldData.
 */
typedef OldData
{
    /**
     * ID Number for this version of the backup.
     *
     * This information is stored implicitly in NZTM since the pointer itself
     * acts as an ID number. To ensure uniqueness, and without making the system
     * state space explode, always set its value to the current txid
     *
     * If it's set to NULL it means there's no backup copy.
     */
    byte id = NULL;

    /**
     * The backup copy (if any)
     */
    byte backup = INVALID;
};

/**
 * Used to keep track of the data after the object has been inflated.
 *
 * This is the mechanism that enabled NZTM to be nonblocking.
 */
typedef Locator
{
    /**
     * Points to the transaction that owns this locator
     */
    byte txn = INVALID;

    /**
     * Points to the unresponsive transaction that caused the object to be
     * inflated.
     * A NULL (0) value indicates that the object was inflated because of a
     * unresponsive reader.
     */
    byte abortedTxn = INVALID;

    /**
     * Points to the actual value of object, if transaction is aborted; and a
     * backup copy of the value when the transaction is active
     */
    byte old = INVALID;

    /**
     * Points to the actual of object if transaction is committed;
     * Tentative value, if transaction is active.
     */
    byte new = INVALID;
};
```

```
/**
 * The main data structure used in NZTM.
 *
 * This maintains the data and the metadata required for proper access
 */
typedef NZObject
{
    /**
     * A "pointer" to the writer (or exclusive) transaction (if the object is
     * not inflated), OR a pointer to the locator that contains the object (if
     * the object is inflated)
     *
     * A value of zero indicates that no one is pointing to it (assuming it's
     * not inflated)
     * Must be changed atomically.
     */
    byte txn = NULL;

    /**
     * Indicates whether this object has been inflated, thereby the data is
     * reachable through the locator.
     *
     * If this value is true, then the txn field points to the locator.
     *
     * In C, this value is actually represented by the least significant bit of
     * the transaction field. Therefore it can be manipulated atomically with it
     */
    bool isInflated = false;

    /**
     * The actual data in place
     */
    byte data = INITV;

    /**
     * A "pointer" to the oldata
     */
    OldData old;

    /**
     * Visible list of readers reading this object.
     *
     * In NZTM it's a different structure, but since it's allocated at the same
     * time as the object we'll consider it as part of it.
     */
    byte readers[PROCESSES] = NULL;

    /**
     * Preallocated Locators.
     *
     * A locator is preallocated for every process. It's done this way since
     * dynamic memory allocation is not supported in the spin models. Moreover,
     * the maximum number of locators possible is 1 per process per transaction.
     */
    Locator locators[PROCESSES];
};

/**
 * The main data structure used in DSTM2 - Shadow.
 *
 * This maintains the data and the metadata required for proper access
 */
typedef SObject
{
    /**
     * A "pointer" to the writer (or exclusive) transaction
     *
     * A value of zero indicates that no one is pointing to it (assuming it's
     * not inflated)
     */
    byte txn = NULL;
```

```
    /**
     * The actual data in place
     */
    byte data = INITV;

    /**
     * The shadow backup field
     */
    byte backup = INVALID;

    /**
     * The object lock.
     *
     * This lock must be held before any modification is made to the object;
     * except for the readers list
     */
    bool lock = false;

    /**
     * Visible list of readers reading this object.
     *
     * It should be a different structure, but since it's allocated at the same
     * time as the object we'll consider it as part of this one.
     */
    byte readers[PROCESSES] = NULL;
};

/**
 * The different states a transaction can be in.
 *
 * ACTIVE: The transaction is running and hasn't been asked to abort. If it
 *   tries to commit it would succeed, and its status would be COMMITTED.
 * COMMITTED: The transaction has finished successfully, it's changes are
 *   permanent.
 * ABORTED: The transaction has finished but failed. Its changes are rolled
 *   back.
 * ABORT_NOW: The transaction is still running but it has been asked to abort.
 *   If it tries to commit it would fail, and its status would be ABORTED.
 */
mtype = {ACTIVE, COMMITTED, ABORTED, ABORT_NOW};

/**
 * Definition of a transaction.
 *
 * There must be one unique location per transaction, do not recycle.
 */
typedef Transaction
{
    mtype status = COMMITTED;
};



/*
 * ================================================================================
 * Function Definitions
 * ================================================================================
 */

/*
 * Transactional functions
 */

/**
 * Determines whether a transaction is active
 */
#define TXACTIVE(x) (transactions[x].status == ACTIVE)

/**
 * Determines whether a transaction finished successfully
 */
#define TXCOMMITTED(x) (transactions[x].status == COMMITTED)
```

```
/**
 * Determines whether a transaction is failed to commit (and knows that)
 */
#define TXABORTED(x) (transactions[x].status == ABORTED)

/**
 * Determines whether a transaction has been asked to abort
 */
#define TXABORTING(x) (transactions[x].status == ABORT_NOW)

/**
 * Transaction definitions: defined local variables that are needed by all
 * processes, to be defined per process.
 *
 * txn: Points to my transaction
 * enemyid: Points to the current enemy transaction
 * oldData: scratch variable used to compare values of backup data
 * lid: keep track of the current locator's id number
 * isInflated: true if the last object opened is inflated
 * success: resembles a return value of whether the function was successful
 */
#define TXN_DEFS()                                                              \
    byte txid = NULL;                                                           \
    byte enemyid = NULL;                                                        \
    OldData oldData;                                                            \
    byte lid = INVALID;                                                         \
    bool isInflated = false;                                                    \
    byte dummy;

/**
 * Begins a new transaction.
 *
 * Each process can allocate transactions from predefined slots in a particular
 * order. This is a modeling restriction to reduce the state-space size.
 *
 * txid is the variable that maintains the value of the pointer to the
 * transaction. Must always be the same variable for every process. Initially
 * must be 0.
 */
inline beginTransaction()
{
    /* Atomic since setting pointers in C appear that way */
    d_step {
        if
        :: (txid == NULL) -> txid = _pid * TXN_PER_PROC + 1;
        :: else -> txid++;
        fi;

        /* Ensure that we're in the correct range */
        assert(txid <= _pid * TXN_PER_PROC + TXN_PER_PROC);

        /* Redundant: Stress that txid cannot be 0, since it's reserved */
        assert(txid != 0);

        /* Ensure that we have enough to handle all processes */
        assert(_pid < PROCESSES);

        /* Ensure we're allocating from a fresh transaction*/
        assert(TXCOMMITTED(txid));

        transactions[txid].status = ACTIVE;
    }
}

/**
 * Attempts to commit a transaction.
 *
 */
inline commitTransaction()
{
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));
```

```
        if
        :: TXACTIVE(txid) -> transactions[txid].status = COMMITTED;
        :: else -> transactions[txid].status = ABORTED;
        fi;
    }
}

/**
 * Validates my current transaction, ensures that I'm still active.
 *
 * Aborts if I've been asked to abort.
 */
inline validateTransaction()
{
    atomic {
        /*
         * Doesn't need to be atomic. Done that way to ensure that the assertion
         * takes place at the correct instance. Other than the assertion, being
         * atomic does not affect behavior.
         */
        if
        :: TXABORTING(txid) -> goto aborted;
        :: else -> assert(TXACTIVE(txid));
        fi;
    }

    /*
     * NOTE: This might trigger some dead code notices. That's ok if the dead
     * code is related to validating reopening for read/write in tests that do
     * not open for read/write the same object again.
     */
}

/**
 * Aborts an enemy transaction atomically.
 *
 * Atomically changes the state of the transaction from ACTIVE to ABORT_NOW.
 * Do not use to abort self.
 */
inline abortEnemy(enemyid)
{
    d_step {
        if
        :: TXACTIVE(enemyid) -> transactions[enemyid].status = ABORT_NOW;
        :: else;
        fi;

        assert(enemyid != txid);
        assert(!TXACTIVE(enemyid));
    }
}

/**
 * Blocks until the enemy acknowledges an abortion or that I have been aborted
 * myself.
 *
 * This is the only blocking portion in the blocking NZSTM
 */
inline blockAborted(enemyid)
{
    (!TXABORTING(enemyid) || !TXACTIVE(txid));
}

/**
 * Handles a conflict with another transaction.
 *
 * @param eid The id of the transaction there's a conflict with
 */
inline resolveConflict(eid)
{
    /*
     * Contention managers will eventually abort any enemy.
     * The way spin operates, this will encompass all possibilities, from waiting
```

```
     * for the enemy to finish by itself, to actually aborting it.
     */
    abortEnemy(eid);
}

/**
 * Loops through every reader in the readers list, and performs an action if the
 * reader isn't me on ALL the readers.
 *
 * Done in this awkward manner to reduce the state-space size
 */
#define LOOP_READERS(action)                                                    \
    if                                                                          \
    :: (PROCESSES > 4 && _pid != 4) ->                                          \
        action(objects[oid].readers[4]);                                        \
    :: else;                                                                    \
    fi;                                                                         \
                                                                                \
    if                                                                          \
    :: (PROCESSES > 3 && _pid != 3) ->                                          \
        action(objects[oid].readers[3]);                                        \
    :: else;                                                                    \
    fi;                                                                         \
                                                                                \
    if                                                                          \
    :: (PROCESSES > 2 && _pid != 2) ->                                          \
        action(objects[oid].readers[2]);                                        \
    :: else;                                                                    \
    fi;                                                                         \
                                                                                \
    if                                                                          \
    :: (PROCESSES > 1 && _pid != 1) ->                                          \
        action(objects[oid].readers[1]);                                        \
    :: else;                                                                    \
    fi;                                                                         \
                                                                                \
    if                                                                          \
    :: (PROCESSES > 0 && _pid != 0) ->                                          \
        action(objects[oid].readers[0]);                                        \
    :: else;                                                                    \
    fi;

/**
 * Handles a conflict with the readers list (visible reads).
 *
 * @param oid The 'pointer' to the object with the readers list
 */
inline resolveReadersConflicts(oid)
{
    /*
     * Resolve conflicts with all readers
     */
    LOOP_READERS(resolveConflict);
}

/**
 * Models waiting for the whole readers list or being impatient and
 * deciding to inflate it.
 */
inline waitOrInflateReaders(oid)
{
#ifdef BLOCKING
    LOOP_READERS(blockAborted);
#else /* nonblocking */
    if
    :: (PROCESSES > 4 && _pid != 4 && TXABORTING(objects[oid].readers[4])) ->
        inflateObjectReader(oid);

    :: (PROCESSES > 3 && _pid != 3 && TXABORTING(objects[oid].readers[3])) ->
        inflateObjectReader(oid);

    :: (PROCESSES > 2 && _pid != 2 && TXABORTING(objects[oid].readers[2])) ->
        inflateObjectReader(oid);
```

```
    :: (PROCESSES > 1 && _pid != 1 && TXABORTING(objects[oid].readers[1])) ->
        inflateObjectReader(oid);

    :: (PROCESSES > 0 && _pid != 0 && TXABORTING(objects[oid].readers[0])) ->
        inflateObjectReader(oid);

    :: else; /* No inflation */
    fi;
#endif /* BLOCKING */
}

/**
 * Breaks out of an outer loop if there are unresponsive readers.
 *
 * Used when trying to deflate an object.
 */
inline breakNonresponsiveReaders(oid)
{
    if
    :: (PROCESSES > 4 && _pid != 4 && TXABORTING(objects[oid].readers[4])) ->
        break;

    :: (PROCESSES > 3 && _pid != 3 && TXABORTING(objects[oid].readers[3])) ->
        break;

    :: (PROCESSES > 2 && _pid != 2 && TXABORTING(objects[oid].readers[2])) ->
        break;

    :: (PROCESSES > 1 && _pid != 1 && TXABORTING(objects[oid].readers[1])) ->
        break;

    :: (PROCESSES > 0 && _pid != 0 && TXABORTING(objects[oid].readers[0])) ->
        break;

    :: else;
    fi;
}

/*
 * OldData Functions
 */

/**
 * Tests whether there's a backup copy of the data
 */
#define IS_OLD(oid) (objects[oid].old.id)

/*
 * Inflated locator functions
 */

/**
 * De-references the locator associated with this particular id
 *
 * @param lid The locator id that we want dereferenced
 */
#define LOCATOR(oid, lid) objects[oid].locators[lid]

/**
 * Sets lid to the id of the locator if the object is inflated. If the object
 * isn't inflated, sets the locator id to an invalid value.
 *
 * @param oid The id of the possibly inflated object
 * @param lid Where to store the id of the locator
 */
inline findLocator(oid, lid)
{
    /* Atomic since the two fields are overloaded into one field in NZTM */
    d_step {
        if
        :: (objects[oid].isInflated) -> lid = objects[oid].txn;
        :: else -> lid = INVALID;
```

```
        fi;
    }
}

/**
 * Inflated an object owned by an unresponsive transaction.
 *
 * Only to be used by a writer to an unresponsive reader that wants to inflate.
 *
 * @param oid The 'pointer' to the NZObject I want to inflate
 */
inline inflateObject(oid)
{
    validateTransaction();

    /* Find the possibly unresponsive transaction */
    d_step {
        enemyid = objects[oid].txn;
        isInflated = objects[oid].isInflated;

        /* This is used for hung writers, so the enemy cannot be me */
        assert(isInflated || enemyid != txid);
    }

    /* Remember the value of the old data pointer */
    d_step {
        oldData.backup = objects[oid].old.backup;
        oldData.id = objects[oid].old.id;
    };

    /* If the old data pointer doesn't exist, copy the in-place data */
    if
    :: (oldData.id == NULL) -> oldData.backup = objects[oid].data;
    :: else;
    fi;

    /* Create and initialize the locator */
    d_step {
        LOCATOR(oid, _pid).old = oldData.backup;
        LOCATOR(oid, _pid).new = oldData.backup;
        LOCATOR(oid, _pid).txn = txid;
        LOCATOR(oid, _pid).abortedTxn = enemyid;
    }

    /*
     * Check that all the assumption I've made still hold:-
     * - There is an unresponsive transaction
     * - The object is not inflated
     * - The value of the old pointer is the same as it was before
     * - I am still active
     */
    if
    :: (!TXABORTING(enemyid) || isInflated ||
        oldData.id != objects[oid].old.id || !TXACTIVE(txid)) ->
        enemyid = INVALID; /* (return fail) Read note at end of function */

    :: else ->
        /* Attempt to install the newData locator to inflate the object */
        d_step {
            if
            :: (objects[oid].txn == enemyid && !objects[oid].isInflated) ->
                objects[oid].isInflated = true;
                objects[oid].txn = _pid;
                isInflated = true;

            :: else;
            fi;
        }

        /* Check if I was successful in inflating the object */
        if
        :: (objects[oid].txn == _pid && objects[oid].isInflated) ->
#ifdef READ_VISIBLE
```

```
            /* Before proceeding, resolve all conflicts with readers */
            resolveReadersConflicts(oid);
#endif /* READ_VISIBLE */
            break; /* Breaks out of the open loop (i.e. return success) */

        :: else -> enemyid = INVALID; /* (i.e. return fail) */
        fi;

        /*
         * NOTE: I am overloading the enemyid field to also indicate a failed
         * inflation. This is done to keep the state space small. In C, it
         * would return either TRUE or FALSE to indicate success or failure.
         */
    fi;
} /* inflateObject(oid) */

/**
 * Inflated an object being visibly read by an unresponsive transaction.
 * Assumes that I have already acquired the object exclusively.
 *
 * I can only leave this if I got aborted.
 * Only used with Visible Reads.
 *
 * @param oid The 'pointer' to the NZObject I want to inflate.
 */
inline inflateObjectReader(oid)
{
    validateTransaction();

    oldData.backup = objects[oid].data; /* A direct copy of the data*/

    /* Create and initialize the locator */
    d_step {
        /* Sanity Checks */
        assert(!IS_OLD(oid));
        assert(objects[oid].txn == txid || objects[oid].isInflated);

        LOCATOR(oid, _pid).old = oldData.backup;
        LOCATOR(oid, _pid).new = oldData.backup;
        LOCATOR(oid, _pid).txn = txid;
        LOCATOR(oid, _pid).abortedTxn = NULL; /* Hung on a reader */
    }

    /* Attempt to install the newData locator to inflate the object */
    atomic {
        if
        :: (objects[oid].txn == txid && !objects[oid].isInflated) ->
            objects[oid].isInflated = true;
            objects[oid].txn = _pid;
            isInflated = true;

        :: else -> assert(TXABORTING(txid));
        fi;
    }

    /* Check if I was successful in inflating the object */
    if
    :: (objects[oid].txn == _pid && objects[oid].isInflated) ->
        /* Before proceeding, resolve all conflicts with readers */
        resolveReadersConflicts(oid);
        break; /* Breaks out of the open loop */

    :: else -> /* I must have been aborted */
        assert(TXABORTING(txid));
        goto aborted;
    fi;

} /* inflateObjectReader(oid) */

/**
 * Opens an inflated object exclusively (used for both read and write)
 *
 * @param oid The 'pointer' to the object I want to open
```

```
 */
inline inflatedOpenExclusive(oid)
{
#ifdef READ_VISIBLE
    validateTransaction();

    /*
     * Before proceeding, resolve all conflicts with readers. This is important
     * since the transaction that started the inflation process might have not
     * gotten around to doing this.
     */
    resolveReadersConflicts(oid);
#endif /* READ_VISIBLE */

    do
    :: TXABORTING(txid) -> goto aborted; /* been asked to abort */

    :: else ->
        findLocator(oid, lid);

        /* Check if the object is inflated */
        if
        :: (lid == INVALID) -> break;
        :: else -> enemyid = LOCATOR(oid, lid).txn;
        fi;

        /* See if it's already open by me */
        if
        :: (enemyid == txid) -> break;
        :: else;
        fi;

        resolveConflict(enemyid);
        validateTransaction();

        /* Create a new locator */
        d_step {
            LOCATOR(oid, _pid).txn = txid;
            LOCATOR(oid, _pid).abortedTxn = LOCATOR(oid, lid).abortedTxn;

            if
            :: TXCOMMITTED(objects[oid].locators[lid].txn) ->
                LOCATOR(oid, _pid).old = LOCATOR(oid, lid).new;
            :: else ->
                LOCATOR(oid, _pid).old = LOCATOR(oid, lid).old;
            fi;

            LOCATOR(oid, _pid).new = LOCATOR(oid, _pid).old;
        };

        /* Try to install the new locator */
        d_step {
            if
            :: (objects[oid].txn == lid && objects[oid].isInflated) ->
                objects[oid].txn = _pid;
                isInflated = true;

            :: else;
            fi;
        }

        /* If I acquired the object, try to deflate it (step 1 below) */
        if
        :: (objects[oid].txn == _pid && objects[oid].isInflated) ->
            deflateObject(oid);
            break;

        :: else;
        fi;
    od;
} /* inflatedOpenExclusive(oid) */

/**
```

```
 * Attempts to deflate an object that has been inflated
 *
 * 1. Acquire the object the DSTM way.
 *
 * 2. Read and remember the value of the NZObject's oldData pointer.
 *
 * 3. Check if the transaction we're waiting on to abort has finally aborted, if
 * not then just proceed with the normal DSTM acquisition stuff. Make sure to
 * check for readers as well.
 *
 * (If Visible Reads) also check that all the transactions in the read list are
 * responsive, if not then just proceed with the normal DSTM acquisition stuff.
 *
 * 4. Check that the current transaction is still Active (meaning that it still
 * has the object acquired). If not, then proceed with the normal I'm aborted
 * stuff.
 *
 * 5. CAS the NZObject's oldData pointer to point to the real current data. If
 * it fails abort.
 *
 * 6. CAS the NZObject's transaction pointer to point to my transaction.
 * If the CAS was successful, then copy the old data back to the data.
 */
inline deflateObject(oid)
{
    do
    :: true -> /* Infinite Loop */

        enemyid = LOCATOR(oid, _pid).abortedTxn;

        /* 2 */

        /* Atomic since in C it's one pointer field */
        d_step {
            oldData.id = objects[oid].old.id;
            oldData.backup = objects[oid].old.backup;
        }

        /* 3 */
#ifdef READ_VISIBLE
        if
        :: TXABORTING(enemyid) -> break;
        :: else -> breakNonresponsiveReaders(oid);
        fi;
#else /* Exclusive Reads*/
        if
        :: TXABORTING(enemyid) -> break;
        :: else;
        fi;
#endif /* READ_VISIBLE */

        /* 4 */
        validateTransaction();

        /* 5 */
        atomic {
            assert(enemyid == NULL || TXABORTED(enemyid));

            if
            :: (objects[oid].old.id == oldData.id) ->
                objects[oid].old.id = txid;
                objects[oid].old.backup = LOCATOR(oid, _pid).new;

            :: else ->
                /*
                 * An unresponsive transaction that's responsive again might
                 * wake up and decide to take a backup copy. Doesn't
                 * necessarily mean I was aborted.
                 */
               break;
            fi;
        }
```

```
        /* 6 */
        atomic {
            if
            :: (objects[oid].txn == _pid && objects[oid].isInflated) ->
                objects[oid].txn = txid;
                objects[oid].isInflated = false;
                isInflated = false;

            :: else ->
                assert(TXABORTING(txid));
                goto aborted;
            fi;
        }

        objects[oid].data = objects[oid].old.backup;
        break;
    od;
} /* deflateObject(oid) */

/*
 * NZObject functions
 */

/* Defined opening macros for different kind of reads */

/* Shadow DSTM2 */
#ifdef SHADOW

#define OPEN_WRITE(oid) openWriteShadow(oid)
#define OPEN_READ(oid) openReadShadow(oid)

#else /* NZSTM */
#ifdef READ_VISIBLE
#define OPEN_WRITE(oid) openWriteVisible(oid)
#define OPEN_READ(oid) openReadVisible(oid)
#else /* Exclusive Reads */
#define OPEN_WRITE(oid) openWriteExclusive(oid)
#define OPEN_READ(oid) openReadExclusive(oid)
#endif /* READ_VISIBLE */
#endif /* SHADOW */
/**
 * Creates a backup copy of the data and points the old pointer to that copy.
 */
inline cloneData(oid)
{
    /* Done atomically since setting a pointer in C appears to be atomic */

    d_step {
        objects[oid].old.id = txid;
        objects[oid].old.backup = objects[oid].data;
    }
}

/**
 * Restores the data pointed to by the old pointer. (as it was taken for backup)
 */
inline restoreOldData(oid)
{
    /* Should not be atomic since it doesn't appear that way in C */

    objects[oid].data = objects[oid].old.backup;
    objects[oid].old.id = NULL;
}

/**
 * Atomically changes the transaction pointer to point to my transaction, if
 * the object is acquired by who I think it is.
 */
inline casTransaction(oid, eid)
{
    d_step {
        if
        :: (objects[oid].txn == eid && !objects[oid].isInflated) ->
```

```
            objects[oid].txn = txid;
            isInflated = false;

        :: else;
        fi
    }
}

/**
 * Acquires the object exclusively (as if using CAS)
 *
 * Clears unnecessary old data pointers as well...
 */
inline acquireExclusive(oid, eid)
{
    /*
     * Nonblocking
     * Problem: I acquire the object, which belonged to a committed txn,
     * and old points to stale data. Before I clear the old pointer, I
     * get aborted. The txn that aborted me now thinks that the old pointer
     * actually points to the real data rather than the state one.
     *
     * Possible solution: before acquiring the object, clear the old data
     * pointer using CAS.
     *
     * Doesn't acquire the object if clearing the old pointer fails.
     */

    /* Atomic since in C it's one pointer field */
    d_step {
        oldData.id = objects[oid].old.id;
        oldData.backup = objects[oid].old.backup;
    }

    if
    :: (oldData.id && objects[oid].txn == enemyid && TXCOMMITTED(enemyid)
            && !objects[oid].isInflated) ->
        atomic {
            if
            :: (objects[oid].old.id == oldData.id) ->
                objects[oid].old.id = NULL;
                objects[oid].old.backup = INVALID;

            :: else;
            fi;
        }

        if
        :: (!objects[oid].old.id) -> casTransaction(oid, eid);
        :: else;
        fi;

    :: else -> casTransaction(oid, eid);
    fi;
}

/**
 * Releases the object from my transaction (atomically, if I have it)
 */
inline releaseObject(oid)
{
    d_step {
        if
        :: (objects[oid].txn == txid && !objects[oid].isInflated ) ->
            objects[oid].txn = NULL;
        :: else;
        fi
    }
}

/**
 * Models waiting for an enemy then inflating the object if it's not responsive.
 *
```

```
 * @param eid The enemy transaction I'm waiting to abort.
 * @param oid The 'pointer' to the object that might inflate
 */
inline waitOrInflate(eid, oid)
{
#ifdef BLOCKING
    /*
     * Wait until the enemy has finished (or I have been aborted)
     */
   blockAborted(eid);
#else /* Nonblocking */
    /*
     * If the enemy is not responsive then inflate. This models both waiting for
     * the object to abort and being impatient since spin tries all possible
     * interleaving of instructions.
     */
    if
    :: TXABORTING(eid) -> inflateObject(oid);
    :: else;
    fi;
#endif /* BLOCKING */
}

/**
 * Opens the object exclusively for writing. Goes to Label aborted if it fails.
 *
 * Doesn't check for readers in the readers list
 *
 * @param oid The 'pointer' to the object I want to open
 */
inline openWriteExclusive(oid)
{
    if
    :: (objects[oid].txn == txid && !objects[oid].isInflated) ->
        validateTransaction();
        isInflated = false;

        /*
         * Check where there's a backup copy of the data or not,
         * which would be the case if I had the object open for read.
         */
        if
        :: (!IS_OLD(oid)) -> cloneData(oid); /* was open for read - upgraded */
        :: else; /* Already open for write */
        fi;

    :: else->
        do
        :: TXABORTING(txid) -> goto aborted; /* been asked to abort */

        :: else ->
                d_step {
                    enemyid = objects[oid].txn;
                    isInflated = objects[oid].isInflated;
                }

                if
#ifndef BLOCKING /* nonblocking */
                :: isInflated ->
                    inflatedOpenExclusive(oid);

                    /* See if inflation or deflation were successful */
                    if
                    :: (objects[oid].txn == _pid && objects[oid].isInflated) ->
                        break;

                    :: (objects[oid].txn == txid && !objects[oid].isInflated) ->
                        break;

                    :: else;
                    fi;
#endif /* BLOCKING */
                :: else -> /* Not inflated */
```

```
                    resolveConflict(enemyid);
                    waitOrInflate(enemyid, oid);
                    validateTransaction();

                    /* Now try to acquire the object */
                    acquireExclusive(oid, enemyid);

                    /* Check if I successfully acquired the object */
                    if
                    :: (objects[oid].txn == txid && !objects[oid].isInflated) ->
                        /* Cleanup if necessary */
                        if
                        :: (TXABORTED(enemyid) && IS_OLD(oid)) ->
                            restoreOldData(oid);

                        :: else;
                        fi;

                        /* Take a backup copy of the data */
                        cloneData(oid);

                        /* Object acquired (return TRUE) */
                        break;

                    :: else; /* Failed to acquire object... */
                    fi;
                fi;
        od;
    fi;
} /* openWriteExclusive */

/**
 * Opens the object exclusively for reading. Goes to Label aborted if it fails.
 *
 * Doesn't backup any of the old data
 *
 * @param oid The 'pointer' to the object I want to open
 */
inline openReadExclusive(oid)
{
    if
    :: (objects[oid].txn == txid && !objects[oid].isInflated) ->
        validateTransaction();
        isInflated = false;

    :: else ->
        do
        :: TXABORTING(txid) -> goto aborted;

        :: else ->
                d_step {
                    enemyid = objects[oid].txn;
                    isInflated = objects[oid].isInflated;
                }

                if
#ifndef BLOCKING /* nonblocking */
                :: isInflated ->
                    inflatedOpenExclusive(oid);

                    /* See if inflation or deflation were successful */
                    if
                    :: (objects[oid].txn == _pid && objects[oid].isInflated) ->
                        break;

                    :: (objects[oid].txn == txid && !objects[oid].isInflated) ->
                        break;

                    :: else;
                    fi;
#endif /* BLOCKING */
                :: else -> /* Not inflated */
                    resolveConflict(enemyid);
```

```
                    waitOrInflate(enemyid, oid);
                    validateTransaction();

                    /* Now try to acquire the object */
                    acquireExclusive(oid, enemyid);

                    /* Check if I acquired the object */
                    if
                    :: (objects[oid].txn == txid && !objects[oid].isInflated) ->
                        /* Cleanup if necessary */
                        if
                        :: (TXABORTED(enemyid) && IS_OLD(oid)) ->
                            restoreOldData(oid);

                        :: else;
                        fi;

                        /* Object acquired (return TRUE) */
                        break;

                    :: else; /* Failed to acquire object... */
                    fi;
                fi;
        od;
    fi;
} /* openReadExclusive */

/**
 * Opens the object exclusively for writing. Goes to Label aborted if it fails.
 *
 * Checks for visible reads.
 *
 * @param oid The 'pointer' to the object I want to open
 */
inline openWriteVisible(oid)
{
    if
    /* See if I already have the object open for writing */
    :: (objects[oid].txn == txid && !objects[oid].isInflated) ->
        validateTransaction();
        isInflated = false;

    :: else ->
        do
        :: TXABORTING(txid) -> goto aborted; /* been asked to abort */

        :: else ->
                d_step {
                    enemyid = objects[oid].txn;
                    isInflated = objects[oid].isInflated;
                }

                if
#ifndef BLOCKING /* nonblocking */
                :: isInflated ->
                    inflatedOpenExclusive(oid);

                    /* See if inflation or deflation were successful */
                    if
                    :: (objects[oid].txn == _pid && objects[oid].isInflated) ->
                        break;

                    :: (objects[oid].txn == txid && !objects[oid].isInflated) ->
                        break;

                    :: else;
                    fi;
#endif /* BLOCKING */
                :: else -> /* Not inflated */
                    resolveConflict(enemyid);
                    waitOrInflate(enemyid, oid);
                    validateTransaction();
```

```
                    /* Now try to acquire the object */
                    acquireExclusive(oid, enemyid);

                    /* Check if I successfully acquired the object */
                    if
                    :: (objects[oid].txn == txid && !objects[oid].isInflated) ->

                        /* Cleanup if necessary (safe before checking reads) */
                        if
                        :: (TXABORTED(enemyid) && IS_OLD(oid)) ->
                            restoreOldData(oid);

                        :: else;
                        fi;

                        resolveReadersConflicts(oid);
                        waitOrInflateReaders(oid);
                        validateTransaction();

                        /* Take a backup copy of the data */
                        cloneData(oid);

                        /* Object acquired */
                        break;

                    :: else; /* Failed to acquire object, try again... */
                    fi;
                fi;
        od;
    fi;
} /* openWriteVisible */

/**
 * Opens the object for reading visibly. Goes to Label aborted if it fails.
 *
 * Uses visible reads with predefined slots per read.
 *
 * @param oid The 'pointer' to the object I want to open
 */
inline openReadVisible(oid)
{
    if
    /* Check if the object is already open for write or read */
    :: ((objects[oid].txn == txid  || objects[oid].readers[_pid] == txid) &&
        !objects[oid].isInflated) ->
        validateTransaction();
        isInflated = false;

    :: else ->
        /*
         * Add this transaction to the readers list.
         * Must be done before checking writers to ensure that readers
         * are visible
         */
        objects[oid].readers[_pid] = txid;

        do
        :: TXABORTING(txid) -> goto aborted; /* been asked to abort */

        :: else ->
                d_step {
                    enemyid = objects[oid].txn;
                    isInflated = objects[oid].isInflated;
                }

                if
#ifndef BLOCKING /* nonblocking */
                :: isInflated ->
                    inflatedOpenExclusive(oid);

                    /* See if inflation or deflation were successful */
                    if
                    :: (objects[oid].txn == _pid && objects[oid].isInflated) ->
```

```
                        break;

                    :: (objects[oid].txn == txid && !objects[oid].isInflated) ->
                        break;

                    :: else;
                    fi;
#endif /* BLOCKING */
                :: else -> /* Not inflated */
                    resolveConflict(enemyid);
                    waitOrInflate(enemyid, oid);
                    validateTransaction();

                    /* Cleanup if necessary */

                    if
                    :: enemyid == INVALID; /* Inflation failed, try the loop again */

                    /* If I need to cleanup, acquire, clean then release to do so */
                    :: (enemyid != INVALID && TXABORTED(enemyid) && IS_OLD(oid)) ->
                        casTransaction(oid, enemyid);

                        if
                        :: (objects[oid].txn == txid && !objects[oid].isInflated) ->
                            restoreOldData(oid);
                            releaseObject(oid);
                            break; /* Object acquired for read */

                        :: else; /* Failed to acquire, redo loop */
                        fi;

                    :: else -> break; /* Object acquired for read */
                    fi;
                fi;
        od;
    fi;
} /* openReadVisible */


/**
 * Acquire the shadow object's lock.
 *
 * Fails to lock if I get aborted before acquiring the lock
 *
 * @param oid The 'pointer' to the object I want to lock
 */
inline lockShadow(oid)
{

    do
    :: TXABORTING(txid) -> goto aborted; /* been aborted */

    :: else ->
        (!objects[oid].lock || !TXACTIVE(txid));

        atomic {
            if
            :: (!objects[oid].lock && TXACTIVE(txid)) ->
                objects[oid].lock = true;
                break;

            :: else;
            fi;
        }
    od;
}

/**
 * Release the shadow object's lock.
 *
 * @param oid The 'pointer' to the object I want to lock
 */
inline unlockShadow(oid)
```

```
{
    d_step {
        assert(objects[oid].lock);
        objects[oid].lock = false;
    }
}

/**
 * Takes a backup copy of the data in-place to the shadow field
 */
inline backupShadow(oid)
{
    d_step {
        assert(objects[oid].lock);
        objects[oid].backup = objects[oid].data;
    }
}

/**
 * Restores backup copy of the data in-place from the shadow field
 */
inline restoreShadow(oid)
{
    d_step {
        assert(objects[oid].lock);
        objects[oid].data = objects[oid].backup;
    }
}

/**
 * Opens the object for writing using DSTM2 Shadow.
 *
 * @param oid The 'pointer' to the object I want to open
 */
inline openWriteShadow(oid)
{
    if
    /* See if I already have the object open for writing */
    :: (objects[oid].txn == txid) ->
        validateTransaction();

    :: else ->
        do
        :: TXABORTING(txid) -> goto aborted; /* been aborted */

        :: else ->
            enemyid = objects[oid].txn;
            resolveConflict(enemyid);

            validateTransaction();

            /* Now try to acquire the object */
            lockShadow(oid);

            /* Check that the enemy is still who I think it is */
            if
            :: (objects[oid].txn == enemyid) ->
                /* Claim ownership of the object */
                objects[oid].txn = txid;

                /* See if we need to cleanup, or just backup */
                if
                :: !TXCOMMITTED(enemyid) ->
                    restoreShadow(oid);

                :: else ->
                    backupShadow(oid);
                fi;

            :: else;
            fi;

            /* Release the object lock */
```

```
            unlockShadow(oid);

            /*
             * If I acquired the object for write successfully, deal with potential
             * readers then break out of the loop.
             *
             * Otherwise, redo the loop!
             */
            if
            :: objects[oid].txn == txid ->
                resolveReadersConflicts(oid);
                break;
            :: else; /* Someone else acquired it; redo */
            fi;

        od;
    fi;
} /* openWriteShadow */

/**
 * Opens the object for reading using DSTM2 Shadow.
 *
 * @param oid The 'pointer' to the object I want to open
 */
inline openReadShadow(oid)
{
    if
    /* Check if the object is already open for write or read */
    :: (objects[oid].txn == txid  || objects[oid].readers[_pid] == txid) ->
        validateTransaction();

    :: else ->
        /*
         * Add this transaction to the readers list.
         * Must be done before checking writers to ensure that readers
         * are visible
         */
        objects[oid].readers[_pid] = txid;

        do
        :: TXABORTING(txid) -> goto aborted; /* been asked to abort */

        :: else ->
            enemyid = objects[oid].txn;
            resolveConflict(enemyid);

            validateTransaction();

            /* See if we need to cleanup after an aborted enemy */
            if
            :: !TXCOMMITTED(enemyid) ->
                lockShadow(oid);

                if
                :: (objects[oid].txn == enemyid) ->
                    assert(!TXCOMMITTED(enemyid));
                    restoreShadow(oid);
                    objects[oid].txn = NULL;
                    unlockShadow(oid);
                    break;

                :: else ->
                    unlockShadow(oid);
                fi;

            :: else ->
                break;
            fi;
        od;
    fi;
} /*openReadShadow*/

/**
```

```
 * Finds the current data value of the object and returns it in value.
 *
 * Assumes the object is open for read (or write)! Doesn't validate that.
 */
inline getValue(oid, value)
{
    /* DSTM2 - Shadow */
#ifdef SHADOW
    value = objects[oid].data;
    validateTransaction();
#endif /* SHADOW */

    /* NZSTM */
#ifndef SHADOW
#ifdef BLOCKING
    value = objects[oid].data;
#else /* nonblocking */
    if
    :: isInflated ->
        findLocator(oid, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        value = LOCATOR(oid, _pid).new;

    :: else->
        value = objects[oid].data;
    fi;

    validateTransaction();
#endif /* BLOCKING */
#endif /* not SHADOW */
}

/**
 * Sets the current data value of the object to value.
 *
 * Assumes the object is open for write! Doesn't validate that.
 */
inline setValue(oid, value)
{
    /* DSTM2 - Shadow */
#ifdef SHADOW
    lockShadow(oid);
    assert(objects[oid].data == objects[oid].backup);
    objects[oid].data = value;
    unlockShadow(oid);
#endif /* SHADOW */

    /* NZSTM */
#ifndef SHADOW
#if defined(BLOCKING) && !defined(SCSS)
    objects[oid].data = value;
#elif defined(BLOCKING) && defined(SCSS)
    atomic {
        if
        :: TXACTIVE(txid) -> objects[oid].data = value;
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;
    }
#else /* nonblocking */
    if
    :: isInflated ->
        findLocator(oid, lid);

        if
```

```
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        LOCATOR(oid, _pid).new = value;

    :: else->
        objects[oid].data = value;
    fi;
#endif /* BLOCKING */
#endif /* not SHADOW */
}

/*
 * ========================================================================
 * Sanity Checks and Assertions
 * ========================================================================
 */

/**
 * Performs sanity checks related to the object being open for write.
 *
 * Run before the data is modified.
 */
inline assertOpenWrite(oid)
{
#ifdef BLOCKING
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));
        assert(objects[oid].txn == txid);
        assert(objects[oid].isInflated == false);
        assert(objects[oid].old.id == txid);
        assert(objects[oid].data == objects[oid].old.backup);
    }
#else /* nonblocking */
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));

        if
        /* Not inflated and I know it */
        :: (!isInflated && !objects[oid].isInflated) ->
            assert(objects[oid].txn == txid);
            assert(objects[oid].old.id == txid);
            assert(objects[oid].data == objects[oid].old.backup);

        /* I think it's not inflated but it is inflated => I was aborted */
        :: (!isInflated && objects[oid].isInflated) ->
            assert(TXABORTING(txid));
            assert(objects[oid].old.id == txid);
            assert(objects[oid].data == objects[oid].old.backup);

        /* I think it's inflated but it's not => I was aborted */
        :: (isInflated && !objects[oid].isInflated) ->
            assert(TXABORTING(txid));
            assert(LOCATOR(oid, _pid).txn == txid);
            assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);

        /* I think it's inflated and it is */
        :: (isInflated && objects[oid].isInflated) ->
            assert(LOCATOR(oid, _pid).txn == txid);
            assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);
        fi;
    }
#endif /* BLOCKING */
}

/**
 * Performs sanity checks related to the object being open for read.
 */
inline assertOpenRead(oid)
{
```

```
#ifdef READ_VISIBLE
#ifdef BLOCKING
    /* visible blocking */
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));

        if
        /* Actually open for write */
        :: (objects[oid].txn == txid) ->
            assert(objects[oid].isInflated == false);
            assert(objects[oid].old.id == txid);
            assert(objects[oid].data == objects[oid].old.backup);

        :: else ->
            assert(objects[oid].readers[_pid] == txid);
        fi;
    }
#else /* nonblocking */
    /* visible nonblocking */
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));

        if
        /* Not inflated and I know it */
        :: (!isInflated && !objects[oid].isInflated) ->
            if
            /* Actually open for write */
            :: (objects[oid].txn == txid) ->
                assert(objects[oid].old.id == txid);
                assert(objects[oid].data == objects[oid].old.backup);

            :: else ->
                assert(objects[oid].readers[_pid] == txid);
            fi;

        /* I think it's not inflated but it is inflated */
        :: (!isInflated && objects[oid].isInflated) ->
            skip; /* Many different things could have happened */

        /* I think it's inflated but it's not => I was aborted */
        :: (isInflated && !objects[oid].isInflated) ->
            assert(TXABORTING(txid));
            assert(LOCATOR(oid, _pid).txn == txid);
            assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);

        /* I think it's inflated and it is */
        :: (isInflated && objects[oid].isInflated) ->
            assert(LOCATOR(oid, _pid).txn == txid);
            assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);
        fi;
    }
#endif /* BLOCKING */

#else /* exclusive reads */

#ifdef BLOCKING
    /* exclusive blocking */
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));
        assert(objects[oid].txn == txid);
        assert(objects[oid].isInflated == false);
    }
#else /* nonblocking */
    /* exclusive nonblocking */
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));

        if
        /* Not inflated and I know it */
        :: (!isInflated && !objects[oid].isInflated) ->
            assert(objects[oid].txn == txid);

        /* I think it's not inflated but it is inflated => I was aborted */
```

```
        :: (!isInflated && objects[oid].isInflated) ->
            assert(TXABORTING(txid));

        /* I think it's inflated but it's not => I was aborted */
        :: (isInflated && !objects[oid].isInflated) ->
            assert(TXABORTING(txid));
            assert(LOCATOR(oid, _pid).txn == txid);
            assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);

        /* I think it's inflated and it is */
        :: (isInflated && objects[oid].isInflated) ->
            assert(LOCATOR(oid, _pid).txn == txid);
            assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);
        fi;
    }
#endif /* BLOCKING */

#endif /* READ_VISIBLE */
}

/*
 * ========================================================================
 * Tests
 * ========================================================================
 */

/*
 * I my C code I would be using functions/macros similar to getValue/setValue
 * as defined above. However, to keep the state space as small as possible I
 * have to resort to the ugly code below.
 *
 * An example of proper getValue and setValue usage can be found below though in
 * addAround
 */

/**
 * Increments and decrements the same object while performing sanity checks.
 */
inline incDecObject(oid)
{
    OPEN_WRITE(oid);
    assertOpenWrite(oid);

#if defined(BLOCKING) && !defined(SCSS)
    objects[oid].data++;

    assert(objects[oid].data == 1 + INITV);

    objects[oid].data--;
#elif defined(BLOCKING) && defined(SCSS)
    atomic {
        if
        :: TXACTIVE(txid) -> objects[oid].data++;
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;
    }

    assert(objects[oid].data == 1 + INITV);

    atomic {
        if
        :: TXACTIVE(txid) -> objects[oid].data--;
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;
    }
#else /* nonblocking */
    if
    :: isInflated ->
        findLocator(oid, lid);
```

```
        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        LOCATOR(oid, _pid).new++;
        assert(LOCATOR(oid, _pid).new == 1 + INITV);
        LOCATOR(oid, _pid).new--;

    :: else->
        objects[oid].data++;
        assert(objects[oid].data == 1 + INITV);
        objects[oid].data--;
    fi;
#endif /* BLOCKING */
}

/**
 * Reads the data of an object that is expected to be 0.
 */
inline readZeroObject(oid)
{
    /*
     * assert(objects[oid].data == INITV);
     */

    OPEN_READ(oid);
    assertOpenRead(oid);

#ifdef BLOCKING
    assert(objects[oid].data == INITV);
#else /* nonblocking */
    if
    :: isInflated ->
        findLocator(oid, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            atomic {
                assert(TXABORTING(txid));
                goto aborted;
            }
        fi;

        assert(LOCATOR(oid, _pid).new == INITV);

    :: else->
        assert(objects[oid].data == INITV);
    fi;
#endif /* BLOCKING */
}

/**
 * Takes from o1 and deposits into o2 the amount in val
 */
inline transferAmount(o1, o2, val)
{
    /*
     * objects[o1].data -= val;
     * objects[o2].data += val;
     * assert(objects[o1].data + objects[o2].data == 2 * INITV);
     */

#ifdef defined(BLOCKING) && !defined(SCSS)
    OPEN_WRITE(o1);
    OPEN_WRITE(o2);

    objects[o1].data = objects[o1].data - val;
    objects[o2].data = objects[o2].data + val;
```

```
    assert(objects[o1].data + objects[o2].data == 2 * INITV);

#elif defined(BLOCKING) && defined(SCSS)
    OPEN_WRITE(o1);
    OPEN_WRITE(o2);

    atomic {
        if
        :: TXACTIVE(txid) -> objects[o1].data = objects[o1].data - val;
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;
    }

    atomic {
        if
        :: TXACTIVE(txid) -> objects[o2].data = objects[o2].data + val;
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;
    }


    assert(objects[o1].data + objects[o2].data == 2 * INITV);

#else /* nonblocking */
    dummy = 0;

    OPEN_WRITE(o1);
    assertOpenWrite(o1);

    if
    :: isInflated ->
        findLocator(o1, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        LOCATOR(o1, _pid).new = LOCATOR(o1, _pid).new - val;
        dummy = dummy + LOCATOR(o1, _pid).new;

    :: else->
        objects[o1].data = objects[o1].data - val;
        dummy = dummy + objects[o1].data;
    fi;

    OPEN_WRITE(o2);
    assertOpenWrite(o2);

    if
    :: isInflated ->
        findLocator(o2, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        LOCATOR(o2, _pid).new = LOCATOR(o2, _pid).new + val;
        dummy = dummy + LOCATOR(o2, _pid).new;

    :: else->
        objects[o2].data = objects[o2].data + val;
        dummy = dummy + objects[o2].data;
```

```
    fi;

    assert(dummy == 2 * INITV);
#endif /* BLOCKING */
}

/**
 * Checks that o1 and o2 balance, the sum of their values is twice of their
 * initial value
 */
inline checkBalance(o1, o2)
{
    /*
     * assert(objects[o1].data + objects[o2].data == 2 * INITV);
     */

#ifdef BLOCKING
    OPEN_READ(o1);
    OPEN_READ(o2);
    assert(objects[o1].data + objects[o2].data == 2 * INITV);
#else /* nonblocking */
    dummy = 0;

    OPEN_READ(o1);
    assertOpenRead(o1);

    if
    :: isInflated ->
        findLocator(o1, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        dummy = dummy + LOCATOR(o1, _pid).new;

    :: else->
        dummy = dummy + objects[o1].data;
    fi;

    OPEN_READ(o2);
    assertOpenRead(o2);

    if
    :: isInflated ->
        findLocator(o2, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        dummy = dummy + LOCATOR(o2, _pid).new;

    :: else->
        dummy = dummy + objects[o2].data;
    fi;

    assert(dummy == 2 * INITV);
#endif /* BLOCKING */
}

/**
 * Performs the add around test which reads the first value, then the second
 * value, adding their sum to the third. Then adding the third back to the
 * first.
 */
inline addAround(o1, o2, o3)
```

```
{
    /*
     * objects[o3].data += objects[o1].data + objects[o2].data;
     * objects[o1].data += objects[o3].data;
     * Asserting all the while that the variables match what's expected, OR
     * that I have been aborted.
     */

    byte a, b, c;

    OPEN_READ(o1);
    getValue(o1, a);
    assert(a == expected[o1] || TXABORTING(txid));

    OPEN_READ(o2);
    getValue(o2, b);
    assert(b == expected[o2] || TXABORTING(txid));

    OPEN_WRITE(o3);
    getValue(o3, c);
    assert(c == expected[o3] || TXABORTING(txid));
    c = a + b + c;
    setValue(o3, c);

    OPEN_WRITE(o1);
    a = a + c;
    setValue(o1, a);
}

/*
 * ========================================================================
 * Preallocating Memory
 * ========================================================================
 */

/**
 * Preallocated transactions.
 *
 * (remember that txn #0 is an unusable committed txn)
 */
Transaction transactions[TOTAL_TXN];

#ifdef SHADOW
/**
 * Preallocated objects for DSTM2 Shadow
 */
SObject objects[TOTAL_OBJECTS];
#else /* NZSTM */
/**
 * Preallocated objects for NZSTM
 */
NZObject objects[TOTAL_OBJECTS];
#endif /* SHADOW */

/**
 * Expected Values
 *
 * Used for some tests to store what the expected values of objects are
 */
byte expected[TOTAL_OBJECTS] = INITV;


/*
 * ========================================================================
 * Running Processes
 * ========================================================================
 */

#ifdef INCDEC_TEST
/**
 * The main body of the program representing the increment/decrement test.
 */
active [PROCESSES] proctype IncDecTest()
```

```
{
    TXN_DEFS();

    beginTransaction();

    /* Do a write test and a read test nondeterministically */
    if
    :: true ->
        incDecObject(0);

    :: true ->
        readZeroObject(1);
    fi;

    if
    :: true ->
        readZeroObject(0);

    :: true ->
        incDecObject(1);
    fi;

/* All transaction must either commit or abort */
progress:

aborted:
    commitTransaction();
}
#endif /* INCDEC_TEST */

#ifdef BALANCE_TEST
/**
 * The main body of the program representing the balance test.
 */

active proctype BalanceTest0()
{
    TXN_DEFS();

    beginTransaction();

    transferAmount(0, 1, 2);

progress0:
aborted:
    commitTransaction();
}

active proctype BalanceTest1()
{
    TXN_DEFS();

    beginTransaction();

    checkBalance(0, 1);

progress1:
aborted:
    commitTransaction();
}

active proctype BalanceTest2()
{
    TXN_DEFS();

    beginTransaction();

    transferAmount(1, 0, 3);

progress2:
aborted:
    commitTransaction();
}
```

```
/*
active proctype BalanceTest3()
{
    TXN_DEFS();

    beginTransaction();

    checkBalance(1, 0);

progress3:
aborted:
    commitTransaction();
}
*/

#endif /* BALANCE_TEST */

#ifdef ADD_AROUND_TEST

/**
 * A process that attempts to perform the following transaction, while updating
 * the expected state for future processes.
 *
 * Read x
 * Read y
 * z += x + y
 * x += z
 */
proctype addrProc(byte x, y, z)
{
    TXN_DEFS();

    beginTransaction();

    addAround(x, y, z);

progress:
aborted:
    /* If I commit, atomically change the system's expected state */
    atomic {
        commitTransaction();

        if
        :: TXCOMMITTED(txid) ->
            expected[z] = expected[x] + expected[y] + expected[z];
            expected[x] = expected[x] + expected[z];
        :: else;
        fi;
    }
}

/**
 * Set the initial values of the objects and start the processes
 */
init {
#ifdef SHADOW
    d_step {
        objects[0].data = 3;
        expected[0] = objects[0].data;

        objects[1].data = 5;
        expected[1] = objects[1].data;

        objects[2].data = 7;
        expected[2] = objects[2].data;
    }
#else /* NZSTM */
    d_step {
        objects[0].data = 3;
        expected[0] = objects[0].data;

        objects[1].data = 5;
```

```
        expected[1] = objects[1].data;

        objects[2].data = 7;
        expected[2] = objects[2].data;
    }
#endif /* SHADOW */

    atomic {
        run addrProc(0, 1, 2);
        run addrProc(1, 2, 0);
        run addrProc(2, 0, 1);
        /*run addrProc(2, 1, 0);*/
    }
}

#endif /* ADD_AROUND_TEST */

/* End of File (EOF) */
```

Bibliography

OpenMP Application Program Interface, Version 3.0. OpenMP Architecture Review Board, 2008.

Python 2.6.4 release. Python Software Foundation, 2009. http://www.python.org/download/releases/2.6.4/.

.NET framework 4 beta 1 enabled to use software transactional memory (STM.NET version 1.0): Programmers' guide. Microsoft, 2009. http://download.microsoft.com/download/9/5/6/9560741A-EEFC-4C02-822C-BB0AFE860E31/STM_User_Guide.pdf.

TIOBE programming community index for May. TIOBE Software BV, 2010. http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

Transactional memory in GCC. GNU Project, 2010. http://gcc.gnu.org/wiki/TransactionalMemory.

Grand Central Dispatch (GCD) Reference. Apple Inc., 2010.

The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.10.4. The GHC Team, 2010. http://www.haskell.org/ghc/docs/6.10-latest/html/users_guide/index.html.

Intel C++ STM compiler, prototype edition 3.0. Intel Corporation, 2010a. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/.

Intel 64 and IA-32 Architectures Software Developer's Manual, volume 1: Basic Architecture. Intel Corporation, 2010b.

A. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2006a.

A. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2006b.

H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *MICRO 31: Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society Press, 1998.

A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *HPCA '03: Proceedings of the 9th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2003.

G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967.

C. S. Ananian. *Architectural and compiler support for strongly atomic transactional memory*. Ph.D. thesis, Massachusetts Institute of Technology, 2007.

C. S. Ananian and M. Rinard. Efficient object-based software transactions. In *SCOOL '05: Synchronization and Concurrency in Object-Oriented Languages*, 2005.

C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2005.

L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 2008.

D. Beazley. Inside the Python GIL (slides). Presented at the Python Concurrency Workshop, 2009.
http://www.dabeaz.com/python/GIL.pdf.

R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 1995.

C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. ACM, 2007.

C. Blundell, A. Raghavan, and M. M. K. Martin. RETCON: Transactional repair without replay. In *ISCA '10: Proceedings of the 37th annual international symposium on Computer architecture*. ACM, 2010.

J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. ACM, 2007.

J. Bobba, N. Goyal, M. D. Hill, M. M. Swift, and D. A. Wood. TokenTM: Efficient execution of large transactions with hardware transactional memory. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 2008.

H. J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *HOTPAR '09: 1st USENIX Workshop on Hot Topics in Parallelism*. USENIX, 2009.

H. J. Boehm, A. Demers, and M. Weiser. A garbage collector for C and C++. 2010. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *Sedms'93: USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems*. USENIX Association, 1993.

F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, 2nd edition, 1995.

C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Queue*. 2008.

L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*. 1978.

L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*. IEEE Computer Society, 2006.

S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance SPARC CMT processor. *IEEE Micro*. IEEE Computer Society, 2009a.

S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Simultaneous speculative threading: a novel pipeline architecture implemented in Sun's Rock processor. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. ACM, 2009b.

T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: a performance view. *IBM Journal of Research and Development*. IBM Corporation, 2007.

S.-E. Choi and E. C. Lewis. A study of common pitfalls in simple multi-threaded programs. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*. ACM, 2000.

W. Chuang, S. Narayanasamy, G. Venkatesh, J. Sampson, M. Van Biesbrouck, G. Pokam, B. Calder, and O. Colavin. Unbounded page-based transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM, 2006.

J. Chung, H. Chafi, A. McDonald, C. C. Minh, B. D. Carlstrom, C. Kozyrakis, , and K. Olukotun. The common case transactional behavior of multithreaded programs. In *HPCA '06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, 2006a.

J. Chung, C. C. Minh, A. McDonald, T. Skare, H. Chafi, B. D. Carlstrom, C. Kozyrakis, and K. Olukotun. Tradeoffs in transactional memory virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM, 2006b.

M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2002.

C. Click. And now some hardware transactional memory comments. Azul Systems, 2009a. http://blogs.azulsystems.com/cliff/2009/02/and-now-some-hardware-transactional-memory-comments.html.

C. Click. Azul's experiences with hardware transactional memory (slides). Azul Systems, Presented at the Bay Area Workshop on Transactional Memory, 2009b. https://vsci.hpl.hp.com/marcs/tm_workshop.html.

C. Click. Java on 1000 cores: Tales of hardware/software co-design. In *Genoa: Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*. Springer-Verlag, 2009c.

E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*. ACM, 1970.

E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*. ACM, 1971.

H. Curnow and B. Wichman. A synthetic benchmark. *Computer Journal*. Oxford University Press, 1976.

L. Dalessandro, V. Marathe, M. Spear, and M. L. Scott. Capabilities and limitations of library-based software transactional memory in C++. In *TRANSACT '07: 2nd ACM SIGPLAN Workshop on Transactional Computing*, 2007.

L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: streamlining STM by abolishing ownership records. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2010.

P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. ACM, 2006.

D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *SPAA '10: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2010.

D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC '06: Proceedings of the 20th International Symposium on Distributed Computing*. Springer-Verlag, 2006.

D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. ACM, 2009a.

D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical Report TR-2009-180, Sun Microsystems Laboratories, 2009b. http://labs.oracle.com/techrep/2009/smli_tr-2009-180.pdf.

D. Dice, Y. Lev, V. Marathe, M. Moir, M. Olszewski, and D. Nussbaum. Simplifying concurrent algorithms by exploiting hardware TM. In *SPAA '10: Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*. ACM, 2010.

E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD 123, Technological University, Eindhoven, 1965.
http://userweb.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html.

E. W. Dijkstra. The strengths of the academic enterprise. Technical Report EWD 1175, The University of Texas at Austin, 1994.
http://userweb.cs.utexas.edu/users/EWD/transcriptions/EWD11xx/EWD1175.html.

C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*. ACM, 2007.

F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: scalable nonzero indicators. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM, 2007.

R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Corporation, 2006.
http://berkeley.intel-research.net/rennals/pubs/052RobEnnals.pdf.

K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*. ACM, 1976.

W. Feng. Making a case for efficient supercomputing. *Queue*. ACM, 2003.

S. Frank and A. Inselberg. Synapse tightly coupled multiprocessors: a new approach to solve old problems. In *AFIPS '84: Proceedings of the July 9-12, 1984, national computer conference and exposition*. ACM, 1984.

K. Fraser. *Practical Lock-Freedom*. Ph.D. thesis, Cambridge University, 2004.

V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. QuakeTM: parallelizing a complex sequential application using transactional memory. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009.

D. Geer. Chip makers turn to multicore processors. *IEEE Computer*. 2005.

R. Gerth. Concise Promela reference. 1997.
http://spinroot.com/spin/Man/Quick.html.

A. Ghuloum. Unwelcome advice. Intel Corporation, 2008.
http://blogs.intel.com/research/2008/06/unwelcome_advice.php.

J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '83: Proceedings of the 10th annual international symposium on Computer architecture*. ACM, 1983.

J. R. Goodman and P. J. Woest. The Wisconsin Multicube: a new large-scale cache-coherent multiprocessor. In *ISCA '88: Proceedings of the 15th Annual International Symposium on Computer architecture*. IEEE Computer Society Press, 1988.

J. R. Goodman, M. Moir, F. Tabba, and C. Wang. System and method for implementing nonblocking zero-indirection transactional memory. *United States Patent Application 20090171962*. 2009a.

J. R. Goodman, M. Moir, F. Tabba, and C. Wang. System and method for implementing hybrid single-compare-single-store operations. *United States Patent Application 20090172299*. 2009b.

J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison Wesley, 3rd edition, 2005.

J. E. Gottschlich, M. Vachharajani, and J. G. Siek. An efficient software transactional memory using commit-time invalidation. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010.

J. Gray. The transaction concept: virtues and limitations (invited paper). In *VLDB '81: Proceedings of the seventh international conference on Very Large Data Bases*. VLDB Endowment, 1981.

J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, 1993.

W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. The MIT Press, 2nd edition edition, 1999.

D. Grossman. The transactional memory / garbage collection analogy. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*. ACM, 2007.

R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. ACM, 2008.

T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*. ACM, 1983.

L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*. IEEE Computer Society, 2004.

D. Harmanci, V. Gramoli, P. Felber, and C. Fetzer. Extensible transactional memory testbed. *Journal of Parallel and Distributed Computing*. Elsevier, 2010.

T. Harris, K. Fraser, and I. Pratt. A practical multi-word compare-and-swap operation. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*. Springer-Verlag, 2002.

T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2005.

T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2nd edition, 2010.

J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2006.

M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*. ACM, 1991.

M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*. ACM, 1993.

M. Herlihy and J. E. B. Moss. System for achieving atomic non-sequential multi-word operations in shared memory. *US Patent 5,428,761*. 1995.

M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*. IEEE Computer Society, 2003a.

M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*. ACM, 2003b.

M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. ACM, 2006.

M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*. IEEE Computer Society Press, 2008.

C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*. ACM, 1969.

C. A. R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*. ACM, 1974.

C. A. R. Hoare. Assertions: a personal perspective. Springer-Verlag New York, Inc., 2002.

O. S. Hofmann, D. E. Porter, C. J. Rossbach, H. E. Ramadan, and E. Witchel. Solving difficult htm problems without difficult hardware. In *TRANSACT '07: 2nd ACM SIGPLAN Workshop on Transactional Computing*, 2007.

O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM (slides). University of Texas at Austin, Presented at ASPLOS '09: The 14th international conference on Architectural support for programming languages and operating systems, 2009a.
http://www.cs.utexas.edu/users/witchel/pubs/asplos09_slides.pptx.

O. S. Hofmann, C. J. Rossbach, and E. Witchel. Maximum benefit from a minimal HTM. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*. ACM, 2009b.

G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley Professional, 2003.

R. L. Hudson, B. Saha, A. Adl-Tabatabai, and B. C. Hertzberg. McRT-Malloc: a scalable transactional memory allocator. In *ISMM '06: Proceedings of the 2006 international symposium on Memory management*, 2006.

J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: making use of incoherence. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*. ACM, 2004.

S. A. R. Jafri, M. Thottethodi, and T. N. Vijaykumar. LiteTM: Reducing transactional state overhead. In *HPCA '10: Proceedings of the 16th International Symposium on High-Performance Computer Architecture*, 2010.

E. H. Jensen, G. W. Hagensen, and J. M. Broughton. A new approch to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Lab, 1987.

T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 1995.

W. N. Joy. Reduced instruction set computers (RISC): Academic/industrial interplay drives computer performance forward. Sun Microsystems, Inc., 1995.
http://www.cs.washington.edu/homes/lazowska/cra/risc.html.

J. Juneau, J. Baker, F. Wierzbicki, L. S. Munoz, and V. Ng. *The Definitive Guide to Jython: Python for the Java Platform*. Apress, 2010.

M. Kadiyala and L. N. Bhuyan. A dynamic cache sub-block design to reduce false sharing. In *ICCD '95: Proceedings of the 1995 International Conference on Computer Design*. IEEE Computer Society, 1995.

B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 2nd edition, 1988.

T. Knight. An architecture for mostly functional languages. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*. ACM, 1986.

G. Koch. Discovering multi-core: Extending the benefits of Moore's law. *Technology@Intel*. 2005.
http://www-us-east.intel.com/technology/magazine/computing/multi-core-0705.pdf.

P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*. IEEE Computer Society Press, 2005.

D. Kroft. Cache memory organization utilizing miss information holding registers to prevent lockup from cache misses. *US Patent 4,370,710*. 1983.

S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2006.

L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*. ACM, 1974.

L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*. ACM, 1978.

J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 1st edition, 2007.

K. M. Lepak and M. H. Lipasti. Silent stores for free. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. ACM, 2000.

K. M. Lepak and M. H. Lipasti. Temporally silent stores. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ACM, 2002.

Y. Lev and M. Moir. Fast read sharing mechanism for software transactional memory (poster paper). In *PODC '04: Proceedings of the 23rd annual symposium on Principles of distributed computing*. ACM, 2004.

Y. Lev, M. Moir, and D. Nussbaum. PhTM: Phased transactional memory. In *TRANSACT '07: 2nd ACM SIGPLAN Workshop on Transactional Computing*, 2007.

Y. Lev, V. Luchangco, V. Marathe, and M. Moir. Anatomy of a scalable software transactional memory. In *TRANSACT '09: The 4th annual SIGPLAN Workshop on Transactional Memory*, 2009.

S. Lie. *Hardware Support for Unbounded Transactional Memory*. Ph.D. thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, May 2004.

S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. ACM, 2008.

V. Luchangco. Against lock-based semantics for transactional memory. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. ACM, 2008.

R. Maddox, G. Singh, R. Safranek, and R. Colwell. *Weaving High Performance Multiprocessor Fabric: Architectural Insights to the Intel QuickPath Interconnect*. Intel Press, 2009.

P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*. 2002.

V. Marathe and M. Moir. Efficient nonblocking software transactional memory (poster paper). In *PPoPP '07: Proceedings of the ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2007.

V. Marathe, W. N. Scherer, III, and M. L. Scott. Adaptive software transactional memory. In *DISC '05: Proceedings of the 19th International Symposium on Distributed Computing*. Springer-Verlag, 2005.

V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer, III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. In *TRANSACT '06: 1st ACM SIGPLAN Workshop on Transactional Computing*, 2006.

M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Computer Architecture News (CAN)*. ACM, 2005.

M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 2001.

M. Marty, B. Beckmann, L. Yen, A. Alameldeen, M. Xu, and K. Moore. Multifacet gems: ISCA tutorial (slides). Univeristy of Wisconsin, Tutorial at ISCA '05: The 32nd annual international symposium on Computer Architecture, 2005. http://www.cs.wisc.edu/gems/isca_tutorial.ppt.

J. D. McCullough, K. H. Speierman, and F. W. Zurcher. A design for a multiple user multiprocessing system. In *AFIPS '65 (Fall, part I): Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*. ACM, 1965.

A. McDonald, J. Chung, H. Chafi, C. C. Minh, B. D. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on chip-multiprocessors. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2005.

J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*. ACM, 1991.

M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 1996.

C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. ACM, 2007.

C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: IEEE International Symposium on Workload Characterization*, 2008.

M. Moir. Hybrid transactional memory. Sun Microsystems Laboratories, 2005. http://web.archive.org/web/20060315094623/http://research.sun.com/scalable/pubs/Moir-Hybrid-2005.pdf.

M. Moir, K. E. Moore, and D. Nussbaum. The adaptive transactional memory test platform: A tool for experimenting with transactional code for Rock. In *TRANSACT '08: The 3rd annual SIGPLAN Workshop on Transactional Memory*, 2008.

G. E. Moore. Cramming more components onto integrated circuits. *Electronics Magazine*. 1965.

K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: log-based transactional memory. In *HPCA '06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, 2006.

D. Neto. Java objects memory structure. 2008. http://www.codeinstructions.com/2008/12/java-objects-memory-structure.html.

J. O'Leary, B. Saha, and M. R. Tuttle. Model checking transactional memory with Spin. In *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society, 2009.

M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, 2007.

V. Pankratius, A. Adl-Tabatabai, and F. Otto. Does transactional memory keep its promises? results from an empirical study. Technical Report 2009-12, University of Karlsruhe, Germany, 2009. http://www.rz.uni-karlsruhe.de/~kb95/papers/pankratius-TMStudy.pdf.

S. M. Pant and G. T. Byrd. Extending concurrency of transactional memory programs by using value prediction. In *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*. ACM, 2009.

M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA '84: Proceedings of the 11th annual international symposium on Computer architecture*. ACM, 1984.

C. Perfumo, N. Sönmez, S. Stipic, O. Unsal, A. Cristal, T. Harris, and M. Valero. The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment. In *CF '08: Proceedings of the 5th conference on Computing frontiers*. ACM, 2008.

G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*. 1981.

R. Rajwar and J. R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *MICRO 34: Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*. IEEE Computer Society, 2001.

R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*. ACM, 2002.

R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd annual international symposium on Computer Architecture*. IEEE Computer Society, 2005.

H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: transactional memory for an operating system. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. ACM, 2007.

H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *MICRO 41: Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2008.

H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2009.

A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006.

N. Riley and C. Zilles. Hardware tansactional memory support for lightweight dynamic language evolution. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM, 2006.

C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *WDDD '09: Proceedings of the 8th Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2009.

C. J. Rossbach, O. S. Hofmann, and E. Witchel. Is transactional programming actually easier? In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2010.

B. Saha, A. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2006.

D. Sayre. Is automatic "folding" of programs efficient enough to displace manual? *Communications of the ACM*. ACM, 1969.

W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*. ACM, 2005.

N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. ACM, 1995.

A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer, III, and M. F. Spear. Hardware acceleration of software transactional memory. In *TRANSACT '06: 1st ACM SIGPLAN Workshop on Transactional Computing*, 2006.

A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott. An integrated hardware-software approach to flexible transactional memory. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*. ACM, 2007.

A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible decoupled transactional memory support. In *ISCA '08: Proceedings of the 35th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 2008.

J. Siracusa. Mac OS X 10.6 Snow Leopard: the Ars Technica review. *Ars Technica*. 2009. http://arstechnica.com/apple/reviews/2009/08/mac-os-x-10-6.ars/12.

S. L. Smith. 32nm Westmere family of processors. Intel Corporation, 2009. http://download.intel.com/pressroom/kits/32nm/westmere/32nm_WSM_Press.pdf.

G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*. ACM, 1995.

M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking transactions without indirection using alert-on-update. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2007.

M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2009.

J. Spolsky. *Joel on Software: And on Diverse and Occasionally Related Matters That Will Prove of Interest to Software Developers, Designers, and Managers, and to Those Who, Whether by Good Fortune or Ill Luck, Work with Them in Some Capacity*. Apress, 2004.

R. M. Stallman. *Using The GNU Compiler Collection: A GNU Manual for GCC Version 3.4.6*. GNU Press, 2004.

J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *HPCA '02: Proceedings of the 8th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, 2002.

G. Stein. Free threading. Python Software Foundation, 2001. http://mail.python.org/pipermail/python-dev/2001-August/017099.html.

J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *IEEE Parallel Distrib. Technol.* IEEE Computer Society Press, 1993.

B. Stroustrup. *The C++ Programming Language: Special Edition*. Addison-Wesley Professional, 2000.

H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*. 2005.
http://www.gotw.ca/publications/concurrency-ddj.htm.

P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE Futurebus. In *ISCA '86: Proceedings of the 13th annual international symposium on Computer architecture*. IEEE Computer Society Press, 1986.

F. Tabba. Adding concurrency in Python using a commercial processor's hardware transactional memory support. *To appear in SIGARCH Computer Architecture News (CAN)*. ACM, 2011.

F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: nonblocking zero-indirection transactional memory. In *TRANSACT '07: 2nd ACM SIGPLAN Workshop on Transactional Computing*. ACM, 2007.

F. Tabba, A. W. Hay, and J. R. Goodman. Transactional value prediction. In *TRANSACT '09: The 4th annual SIGPLAN Workshop on Transactional Memory*. ACM, 2009a.

F. Tabba, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. NZTM: nonblocking zero-indirection transactional memory. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 2009b.

F. Tabba, A. W. Hay, and J. R. Goodman. Transactional conflict decoupling and value prediction. Under submission, 2011.

C. K. Tang. Cache system design in the tightly coupled multiprocessor system. In *AFIPS '76: Proceedings of the June 7-10, 1976, national computer conference and exposition*. ACM, 1976.

C. P. Thacker and L. C. Stewart. Firefly: a multiprocessor workstation. In *ASPLOS-II: Proceedings of the second international conference on Architectual support for programming languages and operating systems*. IEEE Computer Society Press, 1987.

S. Tomić, C. Perfumo, C. Kulkarni, A. Armejach, A. Cristal, O. Unsal, T. Harris, and M. Valero. EazyHTM: eager-lazy hardware transactional memory. In *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2009.

J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*. 1994.

M. Tremblay and S. Chaudhry. A third-generation 65nm 16-core 32-thread plus 32-scout-thread CMT SPARC processor. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, 2008.

A. B. Tucker. *Computer Science Handbook, Second Edition*. Chapman & Hall/CRC, 2004.

E. Vallejo, T. Harris, A. Cristal, O. S. Unsal, and M. Valero. Hybrid transactional memory to accelerate safe lock-based transactions. In *TRANSACT '08: The 3rd annual SIGPLAN Workshop on Transactional Memory*, 2008.

G. van Rossum. [Python-3000] the future of the GIL. Python Software Foundation, 2007. http://mail.python.org/pipermail/python-3000/2007-May/007414.html.

G. van Rossum. *The Python Language Reference: Release 2.6.4*. Python Software Foundation, 2009a.

G. van Rossum. *The Python/C API: Release 2.6.4*. Python Software Foundation, 2009b.

G. van Rossum. *Extending and Embedding Python: Release 2.6.4*. Python Software Foundation, 2009c.

G. van Rossum. *The Python Standard Library: Release 2.6.4*. Python Software Foundation, 2009d.

A. Vance. Sun is said to cancel big chip project. The New York Times (Bits Blog), 2009. http://bits.blogs.nytimes.com/2009/06/15/sun-is-said-to-cancel-big-chip-project/.

D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual, Version 9*. PTR Prentice Hall, 2000.

R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*. ACM, 1984.

S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*. ACM, 1995.

W. A. Wulf. Compilers and computer architecture. *IEEE Computer*. IEEE Computer Society, 1981.

L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*. IEEE Computer Society, 2007.

R. M. Yoo, Y. Ni, A. Welc, B. Saha, A. Adl-Tabatabai, and H.-H. S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*. ACM, 2008.