Transactional Memory

Synthesis Lectures on Computer Architecture

Editor
Mark D. Hill, *University of Wisconsin, Madison*

Synthesis Lectures on Computer Architecture publishes 50- to 150 page publications on topics pertaining to the science and art of designing, analyzing, selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals.

Transactional Memory
James R. Larus and Ravi Rajwar
2007

Quantum Computing for Computer Architects
Tzvetan S. Metodi, Frederic T. Chong
2006

Transactional Memory

James R. Larus
Microsoft
larus@microsoft.com

Ravi Rajwar
Intel Corporation
ravi.rajwar@intel.com

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE #2

MORGAN & CLAYPOOL PUBLISHERS

ABSTRACT

The advent of multicore processors has renewed interest in the idea of incorporating transactions into the programming model used to write parallel programs. This approach, known as transactional memory, offers an alternative, and hopefully better, way to coordinate concurrent threads. The ACI (atomicity, consistency, isolation) properties of transactions provide a foundation to ensure that concurrent reads and writes of shared data do not produce inconsistent or incorrect results. At a higher level, a computation wrapped in a transaction executes atomically – either it completes successfully and commits its result in its entirety or it aborts. In addition, isolation ensures the transaction produces the same result as if no other transactions were executing concurrently.

Although transactions are not a parallel programming panacea, they shift much of the burden of synchronizing and coordinating parallel computations from a programmer to a compiler, runtime system, and hardware. The challenge for the system implementers is to build an efficient transactional memory infrastructure. This book presents an overview of the state of the art in the design and implementation of transactional memory systems, as of early summer 2006.

KEYWORDS

Contents

Acknowledgements

CHAPTER 1

Introduction

1.1   MOTIVATION

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. Fortunately, they were wrong. Parallel computers and parallel programming remained a source of frustration to the small group of programmers faced with enormous computations that outstripped conventional computers. The rest of the world programmed sequential computers. For most people and most applications, the 40–50% increase per year in sequential computer performance, made possible by semiconductor and computer architecture improvements, was more than sufficient.

As Bruce Spring stem sings, "good times got a way of coming to an end." Lost in the clamor of the Y2K nonevent and the .COM boom and bust, a less heralded but more significant milestone occurred. Around 2004, 50 years of exponential improvement in the performance of sequential computers ended [1]. Although the quantity of transistors on a chip continues to follow Moore's law (doubling roughly every two years), it has become increasingly difficult to continue to improve the performance of sequential processors. Increasing clock frequency to increase performance is not feasible any longer, due to power and cooling concerns. In the terminology of Intel's founder, Andrew Grove, this is an inflection point—a "time in the life of a business when its fundamentals are about to change" [2].

1.1.1   Single-Chip Parallel Computers

Industry's response to these changes was to introduce single-chip, parallel computers, variously known as "chip multiprocessors," "multicore," or "manycore" computers. The architecture of these computers puts two or more independent processors on a single chip and connects them through a shared memory. The architecture is similar to shared-memory multiprocessors.

This parallel architecture offers a potential solution to the problem of stalled performance growth. The number of processors that can be fabricated on a chip will continued to increase, at least for the next few generations, at the Moore's law rate of approximately 40% per year, and so will double roughly every two years. As the number of processors on a chip doubles, so does

the number of instructions executed per second—without increasing clock speed. This means that the performance of a well-formulated parallel program will also continue to improve at roughly Moore's law rate. Continued performance improvement permits a program's developers to increase its functionality by incorporating sophisticated, new features—the dynamic that has driven the software industry for a long time.

Unfortunately, despite more than 40 years' experience with parallel computers, few "well-formulated" parallel programs exist. Programming parallel computers has proven to be far more difficult than programming sequential computers (which pose many difficult challenges in producing robust, reliable, and secure code). Parallel algorithms are more difficult to formulate and prove correct than sequential algorithms. A parallel program is far more difficult to design, write, and debug than an equivalent sequential program. The nondeterministic bugs that occur in concurrent programs are notoriously difficult to find and remedy. Finally, to add insult to injury, parallel programs often perform poorly. Part of these difficulties may be attributable to the exotic nature of parallel programming, which was of interest to only a small community, was not widely investigated or taught by academics, and was ignored by most software vendors.

1.1.2   Difficulty of Parallel Programming

However, the authors believe that this explanation is insufficient. Parallel programming is fundamentally more difficult than sequential programming. At its core, people have a great deal of difficulty keeping track of concurrently occurring events. Psychologists call this phenomena "attention" and have been studying it for a century. A seminal experiment was Cherry's dichotic listening task, in which a person was asked to repeat a message heard in one ear, while ignoring a different message played to the other ear [3]. People are very good at filtering the competing message because they attend to a single channel at a time.

Concurrency and nondeterminacy greatly increase the number of items that a software developer must keep in mind. Consequently, few people are able to systematically reason about a parallel program's behavior. Consider an example. Professor Maurice Herlihy of Brown University has observed that implementing a queue data structure is a simple programming assignment in an introductory programming course. The parallel analogue, which allows concurrent enqueues and dequeues, is a publishable result because of the difficulty of coordinating concurrent access and handling the boundary conditions [4].

In addition, program analysis tools, which compensate for human failings by systematically identifying program defects, find parallel code to be provably more difficult to analyze than sequential code. For example, context-sensitive analysis is a fundamental technique for analyzing sequential programs. It improves the precision of analysis in defect-detection tools and compilers by analyzing a function with respect to a specific calling context, instead of the

union of all possible call sites. In a concurrent program, combining context-sensitive analysis with the synchronization analysis necessary to understand the communications between even two threads results in an undecidable problem [5]. Fortunately, some analyses, such as safety properties in threads that communicate only through mutual exclusion locks, are efficiently decidable [6].

Finally—and a primary motivation for the strong interest in transactional memory—the programming models, languages, and tools available to a parallel programmer have lagged far behind those available for sequential programs. Consider the two prevalent parallel programming models: data parallelism and task parallelism.

Data parallelism is an effective programming model that applies an operation simultaneously to an aggregate of individual items [7]. It is particularly appropriate for numeric computations, which use numeric matrices as their primary data structure. Programs often manipulate a matrix as an aggregate, for example, by adding it to another matrix. Scientific programming languages, such as High Performance Fortran (HPF) [8], directly support data parallel programming with a collection of operators on matrices and ways to combine these operations. Parallelism is implicit and abundant in data parallel programs. A compiler exploits the inherent concurrency of applying an operation to the elements of an aggregate by partitioning the work among the available processors. This approach shifts the burden of synchronization and load balancing from a programmer to a compiler and run-time system. Unfortunately, data parallelism is not a universal programming model. It is natural and convenient for some problems [7], but difficult to apply to most data structures and programming problems.

The other common programming model is *task parallelism*, which executes computations on concurrent threads that are coordinated with explicit synchronization such as locks, semaphores, queues, etc. This unstructured programming model imposes no restrictions on the code that each thread executes, when or how threads communicate, or how tasks are assigned to threads. The model is a general one, capable of expressing all forms of parallel computation. It, however, is very difficult to program correctly. In many ways, the model is at the same (low) level of abstraction as the underlying computer's hardware; in fact, processors directly implement many of the constructs used to write this type of program.

1.1.3   Parallel Programming Abstractions

A key shortcoming of task parallelism, however, is its lack of effective mechanisms for abstraction and composition—computer science's two fundamental tools for managing complexity. An *abstraction* is a simplified view of an entity, which captures the features that are essential to understand and manipulate it for a particular purpose. People use abstraction all the time. For example, consider an observer "Jim" and a dog "Sally" barking from the backyard across the

street. Sally is Jim's abstraction of the dog interrupting his writing of this book. In considering her barking, Jim need not remember that Sally is actually a one-year-old Golden Retriever and certainly, Jim does not think of her as a quadruped mammal. The latter specifics are true, but not germane to Jim's irritation at the barking. Abstraction hides irrelevant detail and complexity and allows humans (and computers) to focus on the aspects of a problem relevant to a specific task.

Composition is the ability to put together two entities to form a larger, more complex entity, which in turn is abstracted into a single, composite entity. Composition and abstraction are closely related, since details of the underlying entities can be suppressed when manipulating the composite product. Composition is also a common human activity. Consider an engineered artifact such as a car, constructed from components such as an engine, brakes, body, etc. For most purposes, the abstraction of a car subsumes these components and allows us to think about a car without considering the details explored in automobile enthusiast magazines.

Modern programming languages support powerful abstraction mechanisms, as well as rich libraries of abstractions for sequential programming. Procedures offer a way to encapsulate and name a sequence of operations. Abstract datatype and objects offer a way to encapsulate and name data structures as well. Libraries, frameworks, and design patterns collect and organize reusable abstractions that are the building blocks of software. Stepping up a level of abstraction, complex software systems, such as operating systems, databases, or middleware, provide the powerful, generally useful abstractions, such as virtual memory, file systems, or relational databases, used by most software. These abstraction mechanisms and abstractions are fundamental to modern software development, which increasingly builds and reuses software components, rather than writing them from scratch.

Parallel programming lacks comparable abstraction mechanisms. Low-level parallel programming models, such as threads and explicit synchronization, are unsuitable for constructing abstractions because explicit synchronization is not composable. A program that uses an abstraction containing explicit synchronization must be aware of its details, to avoid causing races or deadlocks.

Consider two examples from Tim Harris and Simon Peyton-Jones of Microsoft Research, a hash table that supports thread-safe `Insert` and `Delete` operations. In a sequential program, each of these operations can be an abstraction. One can fully specify their behavior without reference to the hash table's implementation. Now, suppose that in a parallel program, we want to construct a new operation, call it `Move`, which deletes an item from one hash table and insert it into another table. The intermediate state, in which neither table contains the item, must not be visible to other threads. Unless this requirement influences the implementation, there is no way to compose `Insert` and `Delete` operations to satisfy this requirement, since they

lock the table only for the duration of the individual operations. Fixing this problem requires new methods such as `LockTable` and `UnlockTable`, which break the hash-table abstraction by exposing an implementation detail. Moreover, these methods are error prone. A client that locks more than one table must be careful to lock them in a globally consistent order (and to unlock them!), to prevent deadlock.

The same phenomenon holds for other forms of parallel composition. Suppose a procedure `p1` waits for one of two input queues to produce data, using an internal call to `WaitAny`, and suppose another procedure `p2` does the same thing on two different queues. We cannot apply `WaitAny` to `p1` and `p2` to wait on any of the four queues, a fundamental loss of compositionality. Instead, programmers use awkward programming techniques, such as collecting queues used in lower level abstractions, performing a single top-level `WaitAny`, and then dispatching back to an appropriate handler. Again, two individually correct abstractions, `p1` and `p2`, cannot be composed into a larger one; instead, they must be ripped apart and awkwardly merged, in direct conflict with the goals of abstraction.

1.2   DATABASE SYSTEMS AND TRANSACTIONS

While parallelism has been a difficult problem for general programming, database systems have successfully exploited concurrency. Databases (DBs) achieve good performance on sequential and parallel computers by executing many queries simultaneously and by running queries on multiple processors when possible. Moreover, the database programming model ensures that the author of an individual query need not worry about this concurrency. Many have wondered if the programming model used by databases, with its relative simplicity and widespread success, could also function as a more general, parallel programming model.

At the heart of the programming model for databases is a *transaction*. A transaction specifies a program semantics in which a computation executes as if it was the only computation accessing the database. Other computations may execute simultaneously, but the model restricts the allowable interactions among the transactions, so each produces results indistinguishable from the situation in which the transactions run one after the other. As a consequence, a programmer who writes code for a transaction lives in the simpler, more familiar sequential programming world and only needs to reason about computations that start with the final results of other transactions. Transactions allow concurrent computations to access a common database and still produce predictable, reproducible results.

Transactions are implemented by an underlying database system or transaction processing monitor, both of which hide complex implementations behind a relatively simple interface [9–11]. These systems contain many sophisticated algorithms, but a programmer only sees a simple programming model that subsumes most aspects of concurrency and failure. Moreover, the

abstract specification of transaction behavior provides a great deal of implementation freedom and allows the construction of efficient database systems.

Transactions offer a proven abstraction mechanism in database systems for constructing reusable parallel computations. A computation executed in a transaction need not expose the data it accesses or the order in which these accesses occur. Composing transactions can be as simple as executing subtransactions in the scope of a surrounding transaction. Moreover, coordination mechanisms provide concise ways to constrain and order the execution of concurrent transactions.

The advent of multicore processors has renewed interest in an old idea, of incorporating transactions into the programming model used to write parallel programs. While programming language transactions bear some similarity to database transactions, the implementation and execution environments differ greatly, as databases typically store data on disks and programs store data in memory. This difference has given this new abstraction its name, *Transactional Memory* (TM).

1.2.1 What Is a Transaction?

A *transaction* is a sequence of actions that appears indivisible and instantaneous to an outside observer. A database transaction has four specific attributes: failure atomicity, consistency, isolation, and durability—collectively known as the ACID properties.

Atomicity requires that all constituent actions in a transaction complete successfully, or that none of these actions appear to start executing. It is not acceptable for a constituent action to fail and for the transaction to finish successfully. Nor is it acceptable for a failed action to leave behind evidence that it executed. A transaction that completes successfully *commits* and one that fails *aborts*. In this book, we will call this property *failure atomicity*, to distinguish it from a more expansive notion of *atomic execution*, which encompasses elements of other ACID properties.

The next property of a transaction is *consistency*. A transaction can modify the state of the world, that is, data in a database or memory. These changes should leave this state consistent, since subsequent transactions start executing from this modified state. Later transactions may have no knowledge of which transactions executed earlier, so it is unrealistic to expect a transaction to execute properly if an earlier transaction left the world in an arbitrary state. Transactions start with the assumption that the world is consistent, and they are required to leave it in a consistent state. The requirement is trivially satisfied if the transaction aborts, since it then does not perturb the initially consistent state.

The meaning of consistency is entirely application dependent. It typically consists of a collection of invariants on data structures. For example, `numCustomers` contains the number of items in the `Customer` table or that the `Customer` table does not contain duplicate entries.

The next property, called *isolation*, requires that each transaction produce a correct result, regardless of which other transactions are executing concurrently. We will explore the semantics of transactions in the next chapter. This property obviously makes transactions an attractive programming model for parallel computers.

The final property is *durability*, which requires that once a transaction commits, its result be permanent (i.e., stored on a durable media such as disk) and available to all subsequent transactions. This property is not important in transactional memory, since data in memory is usually transient.

1.3   TRANSACTIONAL MEMORY

In 1977, Lomet observed that an abstraction similar to a database transaction might make a good programming language mechanism to ensure the consistency of data shared among several processes [12]. The paper did not describe a practical implementation competitive with explicit synchronization, and so the idea lay fallow until Herlihy and Moss in 1993 [13] proposed hardware-supported transactional memory as a mechanism for building lock-free data structures. In the past few years, there has been a huge ground swell of interest in both hardware and software systems for implementing transactional memory.

The basic idea is very simple. The ACI properties of transactions provide a convenient abstraction for coordinating concurrent reads and writes of shared data in a multithreaded or multiprocess system. Accesses to shared data originate in computations executing on concurrent threads that run on one or more processors. Without a mechanism to coordinate these accesses, reads and writes from various computations can intermix in ways that produce inconsistent, incorrect, and nondeterministic results.

Today, this coordination is the responsibility of a programmer, who has only low-level mechanisms, such as locks, semaphores, mutexes, etc., to prevent two concurrent threads from interfering. Even modern languages such as Java and C# provide only a slightly higher level construct, a monitor, to prevent concurrent access to an object's internal data. As discussed previously, these low-level mechanisms are difficult to use correctly and are not composable.

Transactions provide an alternative approach to coordinating concurrent threads. A program can wrap a computation in a transaction. Failure atomicity ensures the computation completes successfully and commits its result in its entirety or it aborts. In addition, isolation ensures that the transaction produces the same result as it would if no other transactions were executing concurrently.

Although isolation appears to be the primary guarantee of transactional memory, the other properties, failure atomicity and consistency, are important. If a programmer's goal is a correct program, then consistency is important, since transactions may execute in unpredictable orders.

It would be difficult to write correct code without the assumption that a transaction starts executing in a consistent state. Failure atomicity is a key part of ensuring consistency. If a transaction fails, it could leave data in an unpredictable and inconsistent state that would cause subsequent transactions to fail. Moreover, a mechanism used to implement failure atomicity, reverting data to an earlier state, turns out to be very important for implementing certain types of concurrency control.

Transactions are not a panacea. It is still (all too) easy to write an incorrect concurrent program, even with transactional memory. For example, Flanagan and Qadeer developed atomicity analysis, which finds where a properly synchronized program releases a lock (i.e., end a transaction) too early [14]. Consider a slightly modified example from their paper:

```
class StringBuffer ... {
    ...
    private int count;

        public StringBuffer append(StringBuffer sb) {
            int len = sb.length();
            int newcount = count + len;
            if (newcount > value.length) expandCapacity(newcount);
            sb.getChars(0, len, value, count);
            count = newcount;
            return this;
    }

    public atomic int length() { return count; }

    public atomic void getChars(...) { ... }
}
```

The `atomic` keyword means that a method executes in a transaction (this is not the most felicitous keyword, since it suggests failure atomicity, rather than isolation, but it is terse and is used most by most researchers). The `append` method is missing a transaction. Even though the operations on the underlying representation (`length` and `getChars`) are transactional, the `append` method should also execute in a transaction, to ensure that the buffer's length does not change between the call on `length` and when the characters are copied into the new buffer.

1.3.1   Differences

Transactions in memory differ from transactions in databases, and consequently require new implementation techniques, a central topic of this book. The following differences are among the most important:

- Data in a database resides on a disk, rather than in memory. Disk accesses take 5–10 ms, or literally time enough to execute millions of instructions. Databases can freely trade computation against disk access. Transactional memory accesses main memory, which incurs a cost of at most several hundred instructions. A transaction cannot perform much computation at a memory access. Hardware support is more attractive for TM than for database systems.

- Transactional memory is not durable since data in memory does not survive program termination. This simplifies the implementation of TM, since the need to record data permanently on disk before a transaction commits considerably complicates a database system.

- Transactional memory is a retrofit into a rich, complex world full of existing programming languages, programming paradigms, libraries, programs, and operating systems. To be successful, transactional memory must coexist with existing infrastructure, even if a long-term goal may be to supplant portions of this world with transactions. Programmers will find it difficult to adopt transactional memory if it requires pervasive changes to programming languages, libraries, or operating systems or compels a closed world, like databases, where the only way to access data is through a transaction.

1.4   THIS BOOK

This book presents an overview of the state of the art in transactional memory, as of early summer 2006. After reading this book, a practitioner, graduate student, or researcher should be aware of the principal challenges and issues in implementing TM and the solutions investigated to date. The focus of this book is the design and implementation of transactional memory. Extensive research is underway both to develop effective programming models and to find efficient implementations, in software and in hardware.

The book is not a manual on programming with TM, since no one yet has the experience to write such a book. Nevertheless, Chapter 2 outlines the basic transactional programming constructs, sketches a programming model, and then presents a broad taxonomy of design choices for software and hardware TM systems. It also briefly discusses some of the key challenges in integrating TM into the existing software ecosystem, in particular, the issues about how TM coexists with existing programming abstractions and facilities.

Chapter 3 describes Software Transactional Memory (STM) implementation techniques and related research on integrating STM into programming languages. STM implements transactional memory on existing processors. Early systems cloned objects (or memory locations), so a transaction could manipulate its own copy of the memory. When the transaction attempted to commit, the STM system looked for conflicts with other transactions, and if none occurred, the system atomically updates the shared state with the transaction's modified values. Recent STM systems have explored other implementation techniques, such as locking objects and updating them in place. These STM systems typically have been implemented in a language run-time system, such as a Java Virtual Machine or .NET CLR, and tightly integrated with a programming language and compiler.

Chapter 4 describes Hardware Transactional Memory (HTM) implementation techniques. HTM research investigates changes to a computer system and instruction set architecture to support transactions. Early HTM systems kept a transaction's modified state in a cache and used the cache coherence protocol to detect conflicts with other transactions. Recent HTM systems have explored spilling transactional data into lower levels of the memory hierarchy or into software-managed memory, and investigate ways to integrate more software with HTM mechanisms. Other research directions include hybrid systems, which integrate HTMs and STMs, and hardware-accelerated STM systems, which use HTM mechanisms to accelerate an STM implementation.

HTM systems typically provide primitive mechanisms that underlie the user-visible languages, compilers, and run-time systems. Software bridges the gap between programmers and hardware, which makes much of the discussion of STM systems, languages, and compilers relevant to HTM systems as well.

This book describes most published TM systems. It sometimes mentions systems described only in technical reports, but rarely explores them in depth, under the assumption that they will appear in the refereed literature. The discussion focuses on a paper's new ideas and implementation techniques. Further details about a system can be found in the original paper or its follow-on papers. When warranted by the complexity or importance of a system, the book explores its implementation in detail.

We omit detailed discussion of performance for three reasons. First, the benchmarks used to evaluate TM systems are almost exclusively small data structures, such as a hash table or red-black tree. The performance of these computational kernels offers little insight into how a full application or system would execute with TM, since code that only performs data structure accesses in tight loops is likely atypical. Guerraoui, Herlihy, and Pochon showed that the behavior of contention management policies is easily and radically changed by inserting a short delay—corresponding to using data—in the popular red-black tree benchmark [15].

Performance evaluation requires better benchmarks, but few programmers will write substantial TM programs until TM systems' performance reaches acceptable levels. This chicken and egg problem seems likely to resolve itself gradually, as STM systems become robust and fast enough for early adopters. Another approach was taken by Chung et al. [16], who mechanically converted 35 sizeable, multithreaded applications to use transactions. First, they transformed synchronized blocks in Java and lock-based critical sections in OpenMP and Pthreads to transactions. Second, they executed the code between a lock release and a subsequent lock acquire in a transaction, to model TCC's transactions-all-the-time execution model (Section 4.5.1). Third, they used transactions to speculatively parallelize loops in OpenMP programs. The underlying operation strongly influenced the characteristics of a transaction. Transactions used for synchronization were of short duration and accessed a small number of memory locations. The few large transactions either were in the JVM itself or replaced a lock held during a long duration operation. Transactions used for speculative parallelization executed a large number of memory operations. It is unclear if programs originally written with transactions will share these characteristics.

Second, few papers meaningfully compare a system's performance against prior work, so it is difficult to evaluate a paper's contribution on a quantitative basis. Differences in the underlying computer hardware, TM implementation, benchmarks, and workloads frustrate direct comparison of the performance of two TM systems or implementation techniques.

Finally, TM is a promising programming model for future multicore systems, which will have low interprocessor communication latencies. Software implementations on today's multiprocessor systems execute with much higher interprocessor communication latencies, which may favor computationally expensive approaches that incur less synchronization or cache traffic. Future systems may favor other tradeoffs.

This book does not contain the answers to many questions. At this point in the evolution of the field, we do not have enough experience building and using transactional memory systems to prefer one approach definitively to another. Instead, our goal in writing this book is to raise the questions and provide an overview of the answers that others have proposed. We hope that this background will help consolidate and advance research in this area and accelerate the search for answers.

ON-LINE BIBLIOGRAPHY

One of the authors maintains an on-line bibliography of materials related to transactional memory, which includes all works cited in this book. The bibliography is available at: http://www.cs.wisc.edu/trans-memory/biblio.

REFERENCES

[1]   K. Olukotun and L. Hammond, "The Future of Microprocessors," *ACM Queue*, Vol. 3(7), pp. 26–29, 2005 doi:10.1145/1095408.1095418

[2]   A.S. Grove, *Only the Paranoid Survive*. New York: Currency, 1996.

[3]   E.C. Cherry, "Some experiments on the recognition of speech, with one and with two ears," *J. Acoust. Soc. Am.*, Vol. 25, pp. 975–979, 1953.

[4]   M.M. Michael and M.L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms," In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, 1996, pp. 267–275 doi:10.1145/248052.248106

[5]   G. Ramalingam, "Context-sensitive synchronization-sensitive analysis is undecidable," *ACM Trans. Program. Lang. Syst.*, Vol. 22(2), pp. 416–430, 2000 doi:10.1145/349214.349241

[6]   V. Kahlon, F. Ivancic and A. Gupta, "Reasoning about threads communicating via locks," In *Proc. 17th Int. Conf. on Computer Aided Verification (CAV 2005)*, Edinburgh, Scotland, 2005, pp. 505–518 doi:10.1007/11513988-49

[7]   W.D. Hillis and G.L. Steele, Jr., "Data parallel algorithms," *Commun. ACM*, Vol. 29(12), pp. 1170–1183, 1986 doi:10.1145/7902.7903

[8]   D.B. Loveman, "High performance Fortran," *IEEE Parallel Distrib. Technol.*, Vol. 1(1), pp. 25–42, 1993 doi:10.1109/88.219857

[9]   P.A. Bernstein, "Transaction processing monitors," *Commun. ACM*, Vol. 33(11), pp. 75–86, 1990 doi:10.1145/92755.92767

[10] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. San Francisco, CA: Morgan Kaufmann, 1992.

[11] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. New York: McGraw-Hill, 2000.

[12] D.B. Lomet, "Process structuring, synchronization, and recovery using atomic actions," In *Proc. ACM Conf. on Language Design for Reliable Software*, Raleigh, NC, 1977, pp. 128–137 doi:10.1145/800022.808319

[13] M. Herlihy and J.E.B. Moss, "Transactional memory: architectural support for lock-free data structures," In *Proc. 20th Annu. Int. Symp. on Computer Architecture (ISCA '93)*, 1993, pp. 289–300 doi:10.1145/165123.165164

[14] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," In *Proc. ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI 03)*, San Diego, CA, 2003, pp. 338–349 doi:10.1145/781131.781169

[15] R. Guerraoui, M. Herlihy and B. Pochon, "Polymorphic contention management," In *Proc. 19th Int. Symp. on Distributed Computing (DISC)*. Krakow: Springer, 2005,

pp. 303–323 (http://lpdwww.epfl.ch/upload/documents/publications/neg−1700857499-main.pdf).

[16] J. Chung, et al., "The common case transactional behavior of multithreaded programs," In *Proc. 12th Int. Symp. on High–Performance Computer Architecture (HPCA)*, 2006 (http://www.hpcaconf.org/hpca12/37_session10_paper1.pdf).

CHAPTER 2

Programming Transactional Memory

This chapter presents transactional memory from a programmer's perspective. Although early hardware transactional memory (HTM) and software transactional memory (STM) research presented transactional memory (TM) as processor instructions and a software library, respectively TM is a programming abstraction. This chapter presents TM from that perspective. It describes the basic programming language support for transactions. It also discusses the behavior of transactional constructs, in either libraries or languages, and their implications for programming. The chapter also outlines many open questions about the semantics of TM and its integration with existing language features. These questions can only be resolved by using TM to write real software—a process that is just beginning.

For the most part, this chapter avoids discussing implementation details, since Chapter 3 on Software Transactional Memory and Chapter 4 on Hardware Transactional Memory describe many proposed systems in detail.

An alternative view of this chapter is that it lays out the requirements for a transactional memory system, whether implemented in hardware or in software. Unfortunately, these requirements are still incomplete and underspecified, as we lack sufficient experience to choose among the design alternatives and we can only speculate about the implementation complexity or programming expressiveness of proposed or incompletely implemented features.

Nevertheless, it is valuable to approach TM from a programmer's perspective. Subsequent chapters on STM and HTM present a bottom-up view of TM. The danger in a bottom-up perspective is that a profusion of details can obscure the larger picture of what TM should accomplish for its users, the programmers. This chapter attempts to remedy that problem.

2.1 BASIC TRANSACTIONAL CONSTRUCTS

For simplicity, we will describe only basic extensions to a Java or C#-like language to support transactional memory. These extensions are the primary constructs proposed by researchers [1–3], not a complete proposal for a TM language extension. Unsafe languages, such as C or C++, may require different implementations, but language features are likely to be similar.

Most research on transactional memory focused on using it as a parallel programming construct, so this discussion will focus on that aspect, as opposed to using transactions for error recovery, real-time programming, or multitasking. In a parallel program, more than one thread executes concurrently, which requires concurrency control to prevent threads from simultaneously accessing a shared resource and to coordinate the actions of threads. Transactional memory provides mechanisms to control both aspects of concurrency.

Mutual exclusion is a mechanism that prevents several threads from accessing a shared resource (e.g., a variable, array, object, or data structure) simultaneously. We can ignore the benign case of read sharing, in which threads only read the resource, since concurrent access only causes problems when one thread modifies a shared resource. When this happens, other threads can read inconsistent values or see intermediate stages in a thread's computation. If several threads modify the resource simultaneously, then any thread's result can be overwritten and the final state of the resource might not correspond to any thread's computation. Mutual exclusion must be conservative. Even if operations are mostly reads, and writes are infrequent, a program must bear the cost of synchronization.

Coordination is the other reason for concurrency control. A program may require the execution of two independent threads to be coordinated, so, for example, thread 2 executes after thread 1 performed an action. Without coordination, threads run independently, and thread 1 may produce its result at the same time, or even after, thread 2 executes.

Transactions are not the only way to control parallel computation, but much of the recent interest in transactional memory is due to a widespread belief that transactions offer a higher–level and less error-prone parallel programming model than better-known alternatives such as locks, semaphores, mutexes, monitors, etc.

2.1.1   Atomic Block

The atomic statement delimits a block of code that should execute in a transaction:

```
atomic {
  if (x != null) x.foo();
  y = true;
}
```

The sementics of this construct are discussed below (Sec. 2.1.2), but for now assume that the block executes with the failure atomicity and isolation properties of a transaction. The resulting transaction is dynamically scoped—it encompasses all code executed while control is in the atomic block, regardless of whether the code itself is lexicographically enclosed by the block. So, for example, the code in function foo also executes transactionally.

A key advantage of transactions is that an atomic block does not name shared resources, either data or synchronization mechanisms. This feature distinguishes it from earlier programming constructs, such as monitors [4], in which a programmer explicitly names the data protected by a critical section. It also distinguishes atomic blocks from lock-based synchronization, in which a programmer explicitly names the synchronization protecting data. Naming a resource used in an abstraction exposes implementation details, and so violates the abstraction's boundary.

Transactions, by contrast, specify the desired execution outcome (failure atomicity and isolation) and rely on a TM system to implement it. As a result, an `atomic` block enables abstractions to hide their implementation and be composable with respect to these properties. *Composition* is the process of creating software from components—abstractions whose internal details are hidden. Programming is far more difficult and error-prone if a programmer cannot depend on the specified interface of an object and instead must understand its implementation.

Using locks in application code to achieve mutual exclusion exposes this low level of detail. If a library routine accesses a data structure protected by locks A and B, then all code that calls this routine must be cognizant of these locks, to avoid running concurrently with a thread that might acquire the locks in the opposite order. If a new version of the library also acquires lock C, this change may ripple throughout the entire program that uses the library.

An atomic block achieves the same end, but hides the mechanism. The library routine can safely access the data structure in a transaction without concern about how the transactional properties are maintained. Code calling the library need not know about the transaction or its implementation, nor be concerned about isolation from concurrently executing threads. The routine is composable.

The atomic block is not a parallel programming panacea. It is still regrettably easy to write incorrect code. Consider this published example [5]:

```
bool flagA = false; bool flagB = false;
```

Thread 1:   *Thread 2:*

```
atomic {                   atomic {
  while (!flagA);            flagA = true;
  flagB = true;             while (!flagB);


}                          }
```

The code in the atomic blocks is incorrect since the loops will not terminate unless the respective flag is true when the block starts executing (in which case, the loop is pointless). This

example was published to illustrate the difficulty of mechanically converting code with explicit synchronization to use transactions, but it also illustrates that atomic blocks by themselves do not guarantee program termination or correct results.

Composing two atomic blocks, as in this example, does not guarantee program termination or correctness. Other mechanisms or logics are necessary to reason about these properties. Atomic blocks make this reasoning easier, since pre- and postconditions apply to the entire transactional sequence of operations, even when the block executes concurrently.

Some languages also define atomic functions or methods, whose body executes in an implicit atomic statement:

```
atomic void foo {                ⇔      void foo {
    if (x != null) x.foo();                  atomic {
    y = true;                                    if (x != null) x.foo();
}                                                y = true;
                                             }
                                         }
```

From the programmer's perspective, an atomic block has three possible outcomes. If the transaction *commits*, its results become part of the program's state visible to code executed outside the atomic block. If the transaction *aborts*, it leaves the program's state unchanged. If the transaction does not terminate, it is undefined.

The TM run-time system or the program can abort a transaction. The former is an implementation mechanism invoked if the transaction's access to a resource conflicts with another transaction or if the transaction deadlocks waiting for a resource. In either case, the system aborts a transaction and reexecutes it, in hope that the problem will not reoccur. In general, this process of reexecution is not visible to a program, except perhaps as lower performance.

These aborts differ from a program-induced abort—which may arise through an explicit abort statement or an exception (Section 2.2.3). When the system aborts a transaction, it reexecutes the transaction to give it another opportunity to complete. When the program aborts a transaction, control passes to the statement after the atomic block or to an exception handler.

What is the result of a transaction? In language proposals to date, a transaction is a statement, so it either modifies the program's state or transfers control. In general, program *state* includes not only variables and data structures in the process running the transaction, but also its communications with entities outside of the process. The latter includes actions such as creating and modifying disk files, interprocess communication over pipes, or even TCP/IP

traffic—all of which affect the larger environment in which a computation executes. Many TM systems avoid the complexity of this communication (Section 2.4.1) by prohibiting IO operations in transactions.

2.1.2   Semantics

A programming abstraction with a simple, clean semantics helps programmers understand the programming construct, increases their chances of writing correct code, and facilitates detecting errors with programming tools. The semantics of transactional memory has not yet been formally specified, although some attempts have been made [6]. Most papers assume the correctness criteria from database transactions (serializability [7]) or concurrent data structures (linearizability [8]). Both criteria specify some aspects of an atomic block, but neither specifies the semantics of nested transactions, language features such as `retry` or `orElse` (Sections 2.1.4 and 2.1.5), or the interaction between code inside atomic blocks and the non-transactional code outside atomic blocks.

Database Correctness Criteria

Since transactions have their roots in databases, it seems natural to adopt the semantic model used by database to specify the behavior of transactional memory. However, the model is not fully applicable because of differences between database-manipulating and multi-threaded programs. To understand this better, let us revisit the three database properties of atomicity, consistency, and isolation.

Atomicity (specifically, failure atomicity) requires that a transaction execute to completion or, in case of failure, to appear not to have executed at all. An aborted transaction should have no side effects.

Consistency requires that a transaction transform the database from one consistent state to another consistent state. Consistency is a property of a specific data structure, application, or database. It cannot be specified independently of the semantics of a particular system. Enforcing consistency requires a programmer to specify data invariants, typically in the form of predicates.

Isolation requires that execution of a transaction not affect the result of concurrently executing transactions. In database systems, the basic correctness condition for concurrent transactions is serializability. It states that the result of executing concurrent transactions on a database must be identical to a result in which these transactions executed serially. Serializability allows a programmer to write a transaction in isolation, as if no other transactions were executing in the database system. The system is free to reorder or interleave transactions, but it must ensure the result of their execution remains serializable.

Serializability is useful for understanding the behavior of transactions in a transactional memory system. The earlier example involving `flagA` and `flagB` is not serializable, since neither transaction will terminate when executed serially. A non-terminating execution does not produce a result, so there is no standard of the correctness for the concurrent execution.

The ACI model is only partly applicable to transactional memory. Although it specifies legal interactions of atomic blocks, it says nothing about the interaction of atomic blocks with code outside of a transaction. Databases, in general, do not face this difficulty since they control access to shared data and require all accesses to occur in transactions. By contrast, transactional memory programs directly access data and allow code to execute outside of a transaction. If transactional and non-transactional code access shared data, their interaction needs to be specified to describe the semantics of transactional memory fully.

Operational Semantics

An intuitive, operational semantics for specifying the interaction of atomic blocks in transactional memory is **single-lock atomicity**, in which a program executes *as if* all atomic blocks were protected by a single, program-wide mutual exclusion lock, so at most one block is in execution at a time. This model is not an implementation technique, as it precludes concurrency, but it specifies the isolation expected from transactions. It, however, does not capture the failure atomicity of a transaction. This aspect of an atomic block's semantics is probably best described with the conventional techniques used to specify programming language constructs.

The single-lock atomicity model does not preclude aggressive transactional memory implementations. For example, if transactions T_1 and T_2 access disjoint data, an implementation that executes and commits them concurrently will satisfy the single-lock atomicity model. In this case, the result of concurrently executing T_1 and T_2 would be the same as a serialized execution in which either transaction executed before the other one.

Non-Transactional Accesses

So far, single-lock atomicity is nothing more than a concrete embodiment of the database serializability condition. However, because this semantics defines a transaction in terms of a conventional lock, it can also discuss what happens when code outside a transaction accesses shared data, which we will call **non-transactional accesses**. The semantic is similar to code executing outside of a critical section in a program written with conventional locking.

Programs of this type can contain a data race [9–11]. A **conflict** occurs when concurrently executing threads access the same shared data and at least one thread modifies the data. When the conflicting accesses are not synchronized, a **data race** occurs. The behavior of code containing data races is undefined and depends on the implementation of a computer's memory consistency model.

Programs that exhibit data races often do so because of programming errors in which a programmer incorrectly identified a critical section or because of performance-enhancing tricks used to avoid the cost of locking and synchronization. With data races, the result of the execution depends on the implementation of a processor's memory consistency model. The precise behavior of the program in the presence of data races is implementation-specific and may differ on other hardware systems. Because data races are often symptomatic of programming errors, they are discouraged.

In a transaction, the single lock provides synchronization (at least conceptually). Non-transactional code does not acquire this lock. Therefore, conflicting data accesses among transactional and non-transactional code can result in data races and undefined behavior.

Data races can be prevented either by using locking and other forms of synchronization to prevent concurrent access or by arranging a program's control flow to prevent these accesses. Consider the following example:

Thread 1:   *Thread 2:*

```
lock_acquire (lock);
   obj.x = 1;                                obj.x = 2;
   if (obj.x != 1) fireMissiles();
lock_release (lock);
```

Thread 1 operates on `obj.x` inside a critical section protected by `lock`. Thread 2's unprotected updates to `obj.x` compete with thread 1's access, resulting in a data race on `obj.x`. The result of the execution is not deterministic as Thread 2's execution may change the value of `obj.x` at any time. A data race is often symptomatic of an incorrect program [10]. Preventing the data race requires Thread 2 to acquire `lock` before modifying `obj.x` or Thread 2 not to execute concurrently with Thread 1.

Suppose that `lock` was the single lock for transactions, so the example becomes:

Thread 1:   *Thread 2:*

```
atomic {
   obj.x = 1;                                obj.x = 2;
   if (obj.x != 1) fireMissiles();
}
```

This behavior of this transactional program is also undefined because of the same data race. The remedies are the same as above. Thread 2 can execute in a transaction (semantically using `lock` to ensure mutual exclusion) or can ensure that it does not access `obj` concurrently with Thread 1.

As an example of the latter behavior, consider the example:

```
Thread T₁:                        Thread T₂:
ListNode res;                     atomic {
atomic {                             ListNode n = lHead;
   res = lHead;                      while (n != null) {
   if (lHead != null)                   n.val ++;
      lhead = lhead.next;               n = n.next
}                                    }
use res;                          }
```

The first thread removes an element from a list. The second thread modifies all elements in the list. The atomic blocks ensure these two operations do not execute concurrently, but why can Thread 1 access the item it removed from the list (`res`) outside the block? The answer is that Thread 1 and Thread 2 cannot access this item concurrently. Suppose Thread 1 executes first. When Thread 1 leaves the atomic block, it will have removed the first item from the shared list (an operation known as **privatization**), so Thread 2 cannot subsequently modify it. If Thread 2 executes first, Thread 1 will not execute its transaction or subsequent statement until Thread 2 finishes, so there is no conflict. This example also demonstrates that not all non-transactional operations constitute a data race. Thread 1's access of `res` after the transaction commits is not a data race.

If this program was not properly synchronized – for example, Thread 2 did not execute in a transaction – the list modification might not execute strictly before or strictly after the Thread 1. For example:

- Thread 2 starts executing and stores `lHead` in a local variable.
- Thread 1 executes its atomic block and references `res` outside the block.
- Thread 2 iterates over the list, from its original head, and modifies each node.
- Thread 1 references `res` again and sees the modified value.

The behavior of a program that uses transactions is well defined if the program does not exhibit data races when the transactions are expressed using a single lock. This semantic provides clear guidelines to programmers using transactional memory.

Data races in a transactional memory program can expose details of the transactional memory implementation and so may produce different results on different systems. For example, in a deferred-update TM system, a transaction typically reads and writes a copy of a memory location, which overwrites the program-visible location when the transaction commits. Code outside of a transaction is only aware of the program-visible location and does not respect the carefully crafted protocols used to update multiple locations atomically. In addition, the order

in which these locations are physically updated depends on the TM implementation. It may not match the underlying memory consistency model behavior. Similarly, in a direct-update TM system, non-transactional code may not respect synchronization protocols and consequently read the contents of a memory location that rolls back when a transaction subsequently aborts. In general, non-transactional code that conflicts with transactions can read inconsistent state, produce results that are overwritten and lost, or disrupt the atomicity of a transaction.

Transactions that abort should not leave shared data in an inconsistent state. In the absence of data races, aborted transactions do not leave any side effects. Therefore, they do not form part of the execution history used to detect data races. However, this is not true in the presence of data races. Conflicting code may read a value from an aborted transaction. Since this code is non-transactional, it does not abort and its read operation constitutes a side effect of the aborted transaction.

Single-lock atomicity covers only the basic transactional construct. Simple nested transactions may require recursive locks. Specifying the behavior of more complex nesting models is an open question.

Linearizability

An alternative to serializability is linearizability. Herlihy and Wing [8] proposed linearizability as a correctness condition for operations on shared concurrent objects. This criterion raises the level of abstraction from a low-level systems description, such as hardware accesses to memory locations, to a higher-level description, such as method invocations on an abstract data type. Linearizability assumes that the effect of a method invocation occurs at an instantaneous point somewhere between the invocation and completion of the method.

Linearizability defines the correctness of an abstract datatype in terms of its history of method invocation and responses. An execution is linearizable if all invocations and responses form a legal sequential history. A datatype is linearizable if all executions are linearizable.

Linearizability can be applied as a correctness criterion for transactional memory by defining transaction method operation (begin transaction, read, write, end transaction) on a logical object representing shared memory (or comprising the multiple objects accessed within the transaction). This model, however, does not distinguish memory accesses inside and outside a transaction. Therefore, as currently specified, it does not define the semantics of transactions executing in multithreaded applications in which shared data is accessed both inside and outside transactions.

2.1.3   Declarations

Depending on implementation details, a programmer may need to annotate functions invoked by a transaction, to ensure that they are compiled appropriately. In the example above, if function `foo` executes inside and outside of a transaction, the system may need two versions of its

body: one accessing data through transactional mechanisms and the other directly referencing the data. STM systems usually impose this distinction. Most HTMs eliminate the need to recompile code because hardware can dynamically change the semantics of an instruction depending on whether it executes inside a transaction. Other HTMs require explicit identification of transactional references and may require different instructions to access data in a transaction.

Declaring that a function executes in a transaction can be cumbersome, particularly if a large fraction of a system runs in transactions. An alternative is to use interprocedural program analysis to identify methods invoked by transactions. Another alternative is to defer compilation until run-time, when the compiler is aware of context in which a method is running.

In addition, programmers can declare data that is shared among transactions, both to improve the performance of a TM system and to document the program [11]. Knowing which data is not shared permits optimizations to improve the performance of both STM and HTM systems, since references to private data cannot conflict with other transactions and consequently need not be tracked by the TM system [12, 13].

Shared data declarations, however, shift the burden of correctness from the TM system back to a programmer. Accidentally omitting a declaration can cause a data race, which is a problem that transactions should eliminate. Again, program analysis can alleviate the burden. For example, a compiler can use escape analysis [14] or a type system [15] to conservatively identify data that cannot be shared with another thread.

2.1.4   Retry

The discussion of semantics says nothing about the order in which two concurrently executing transactions should modify program state: either could commit before the other. In database systems, transactions are generally independent operations on a database and indeterminacy of this sort is often acceptable. Within a single process, transactions are more tightly coupled; it is often necessary to coordinate which transactions execute and when. A common example is one transaction produces a value used by another.

Harris et al. [2] introduced the `retry` statement to coordinate transactions. A transaction that executes a `retry` statement aborts and then reexecutes. This is the programmer-controlled analogue of the actions that occur when most TM systems detect a conflict. `retry` is a general mechanism that allows a transaction to abandon its current computation for an arbitrary reason and reexecute in hope of producing a different result. Unlike the explicit signaling, with conditional variables and `signal/wait` operations, or the implicit signaling, with a `waituntil` predicate statement, used in monitors [16], `retry` does not name either the transaction being coordinated with or the shared locations.

Harris suggested delaying reexecution until the system detects changes in one or more of the values that the transaction read in its previous execution, so its outcome may be different.

This makes `retry` into a potentially efficient mechanism for coordinating transactions. For example, the following transaction waits until a buffer contains an item before executing its computation:

```
atomic {
   if (buffer.isEmpty()) retry;
   Object x = buffer.getElement();
   ...
}
```

This example illustrates the advantage of combining `atomic` and `retry` statements. Since the predicate examining the buffer executes in the `atomic` block, if it finds an item in the buffer, this value will be present when the next statement in the block goes to get it. The transaction's isolation property ensures that there is no "window of vulnerability" through which another thread can come in and removes an item.

Practical systems may need a mechanism to limit the number of retries, since a transaction itself cannot keep track of the number of times it retries (each abort rolls back the counter). The Atomos language provides such a limit. Open nested transactions (Section 2.2.2) provide another approach to limiting the number of retries by permitting a transaction to exclude data from the rollback process.

Atomos also allows a `retry` statement to specify the locations (*watch set*) that trigger reexecution of an atomic block [3]. A watch set is potentially smaller than the locations read by an aborted transaction. Reducing the size of this set can improve performance, particularly for hardware, which may only be able to track a bounded number of changes in hardware.

2.1.5   OrElse

Harris et al. [2] also introduced the `orElse` operation; another mechanism to coordinate the execution of two transactions. If T_1 and T_2 are transactions, then T_1 `orElse` T_2 starts by executing transaction T_1:

1. If T_1 commits or explicitly aborts (with an explicit `abort` statement or uncaught exception), the `orElse` statement terminates without executing T_2.
2. If T_1 executes a `retry` statement, the `orElse` operation starts executing T_2.
3. If T_2 commits or explicitly aborts, the `orElse` statement finishes execution.
4. If T_2 executes a `retry` statement, the `orElse` statement waits until a location read by either transaction changes, and then reexecutes itself in the same manner.

The `orElse` statement must execute in an atomic block, because the composition of two transactions is itself a transaction that can commit, abort, or retry.

For example, suppose we want to read a value from one of two transactional queues. `getElement` is a blocking operation will retry if a queue is empty, to wait until there is a value to return. The following code checks two queues and returns a value from `Q1` if possible, otherwise it attempts to dequeue an item from queue `Q2` and only blocks if both are empty:

```
atomic {
    { x = Q1.getElement(); }
    orElse
    { x = Q2.getElement(); }
}
```

Harris and Peyton-Jones deliberately specified left-to-right evaluation of the transactions, as opposed to concurrent or nondeterministic evaluation, to enable use of the `orElse` construct to produce values from blocking operations that uses `retry` to wait until a condition is satisfied. For example, the following code returns `null` if a blocking `getElement` operation waits on an empty queue:

```
Object getElementOrNull ()
    atomic {
        { return this.getElement(); }
        orElse
        { return null; }
    }
}
```

2.2   TRANSACTION DESIGN SPACE

Even for the simple programming constructs above, there are many alternatives in the semantics of a transaction and many issues in the transaction's interaction with other programming and system abstractions.

2.2.1   Weak and Strong Isolation

Blundell et al. [5] introduced the terms weak atomicity and strong atomicity. Weak atomicity guarantees transactional semantics only among transactions. Strong atomicity guarantees transactional semantics between transactions and non-transactional code, in addition to transactions. The terms "weak atomicity" and "strong atomicity" are misnomers, since the property

they specify is isolation, not atomicity. The more appropriate names we will use in this book are **weak isolation** and **strong isolation**.

With **weak isolation**, a conflicting memory reference executed outside of a transaction may not fellow the protocols of the TM system. Consequently, the reference may return an inconsistent value or disrupt the correct execution of the transaction. The exact consequences are implementation-specific. With our single-lock atomicity semantic model (Section 2.1.2), an access outside of an atomic block that conflicts with a reference in a transaction introduces the possibility of a data race and makes the system's behavior undefined.

Weak isolation is sometimes understood to necessitate that all access to shared data occur inside a transaction. However, this requirement is unnecessarily strong. A program only needs to ensure that non-transactional accesses do not conflict with transactional accesses, either by making the accesses not overlap in time or in memory location.

Data types are a blunt tool for identifying mistakes in sharing data. For example, code executed in a transaction could be restricted to access data whose type is transactional. It would be a detectable error to access data of this type outside a transaction. This is the approach taken by the Haskell STM [2], where mutable variables are accessible only within a transaction.

In Haskell, however, most data is read-only and is accessible both inside and outside a transaction. In non-functional languages, most data is mutable and must be partitioned into transactional and non-transactional data. This distinction effectively divides a program into two worlds, which communicate only through immutable values. This division complicates the architecture of a program, where data may originate in the non-transactional world, be processed in transactions, and then return to the non-transactional world for further computation or IO.

Lev and Maessen propose tracking the sharing of objects at runtime, to detect when a shared object is accessed outside a transaction [18]. The tracking conservatively assumes that an object is not shared only if it is visible to only one thread. Objects start local and transition to shared when a global variable or a shared object points to them. The cost of tracking is unclear.

Strong isolation automatically converts all operations outside an atomic block into individual transactional operations, thus replicating the database model in which all accesses to shared state execute in transactions.

For a program without data races, weak and strong isolation is equivalent with respect to the semantics of single-lock atomicity.

Although strong isolation appears to specify precise behavior even in the presence of data races, it by itself does not necessarily lead to correct and safe concurrent execution. A programmer may have incorrectly marked transaction boundaries, so that a (transactional) statement executes between two operations that should have executed atomically. For example, the debit and credit operations that occur when money is transferred between two bank accounts should appear atomic, or another thread can see an inconsistency. If a programmer incorrectly delimits a

transaction, the program's concurrent execution can be incorrect, even under strong isolation, although the incorrect execution itself would be well defined.

Strong isolation, nevertheless, has some advantages. The mechanisms used to implement strong isolation can help detect conflicting data accesses, if a programmer's goal is to eliminate data races. Strong isolation also specifies program behavior in the presence of data races, which may aid in debugging programs.

2.2.2   Nested Transactions

A *nested transaction* is a transaction whose execution is properly contained in the dynamic extent of another transaction. For now, we assume that there is only one thread of control for the transactions, so the outer one passes control to the inner one. The inner transaction sees modifications to program state made by the outer transaction. The behavior of the two transactions can be linked in several ways.

If the transactions are *flattened*, aborting the inner transaction causes the outer transaction to abort, but committing the inner transaction has no effect until the outer transaction commits, at which point the inner transaction's changes become visible to other threads. The outer transaction sees modifications to program state made by the inner transaction. If the inner transaction is flattened in the following example, when the outer transaction terminates, the variable x has value 1:

```
int x = 1;

atomic {
   x = 2;
   atomic flatten {
      x = 3;
      abort;
   }
}
```

Flattened transactions are easy to implement, since there is only a single transaction in execution. However, they are a poor programming abstraction that subverts program composition, since an abort in a library routine terminates all surrounding transactions.

Transactions that are not flattened have two alternative semantics. A *closed transaction* aborts without terminating its parent transaction. When a closed inner transaction commits or aborts, control passes to its surrounding transaction. If the inner transaction commits, its modifications become visible to the surrounding transaction. However, nonsurrounding transactions see these changes only when the outermost surrounding transaction commits. In the

following example, variable x is left with the value 2 because the inner transaction's assignment is undone by the abort. Closed nesting can have higher overheads than flattened transactions. For executions that commit successfully, the behavior of flattening and closed nesting is equivalent. If commits are common, a TM system can take advantage of flattening transactions as a performance optimization and switch to closed nesting if aborts occur:

```
int x = 1;

atomic {
   x = 2;
   atomic closed {
      x = 3;
      abort;
   }
}
```

By contrast, when an *open transaction* commits, its changes become visible to all other transactions in the system, even if the surrounding transaction is still executing. Moreover, even if the parent transaction aborts, the results of the nested, open transactions will remain committed. In the following example, even after the outer transaction aborts, variable x is left with the value 3:

```
int x = 1;

atomic {
   x = 2;
   atomic open {
      x = 3;
   }
   abort;
}
```

Open transactions permit a transaction to make permanent modifications to a program's state, which are not rolled back if a surrounding transaction aborts. In addition, they permit unrelated code, such as a garbage collector, to execute in the middle of a transaction and make permanent changes that are unaffected by the surrounding transactions. These uses of open transactions can improve program performance.

For example, a common programming paradigm is to use a counter to generate unique tokens. If this counter is incremented in a closed transaction, then every transaction that uses the counter conflicts, and the counter effectively serializes their execution. If the transaction that obtains value N from the counter aborts, then every transaction that obtains a later value must also abort. Open transactions provide a mechanism to increment the counter atomically without serializing execution:

```
int counter = 1;

atomic {
   ...
   atomic open {
      counter += 1;
   }
   ...
}
```

On the other hand, open transactions can subvert the ACI properties of transactions. The correctness of an optimization may depend on a subtle understanding of the semantics of a program. Consider the example above. In general, tokens only have to be unique (not contiguous), so it does not matter if an aborted transaction generated a value that was discarded. However, if this counter were recording the number of operations successfully completed, an aborted transaction must decrement the counter when it fails, an action known as compensation.

Moss and Hosking describe a reference model for nested transactions that provides a precise definition of open and closed nesting [19].

Zilles and Baugh described an operation similar to open nested transactions which they called "pause" [20]. This operation suspends a transaction and executes nontransactionally until the transaction explicitly resumes. Unlike open nesting, operations in a pause do not have transactional semantics and must use explicit synchronization.

Appropriate semantics for nested transactions become murkier if multiple nested transactions execute concurrently, say because a transaction forked off threads to execute the nested transactions. To preserve isolation, the inner transactions should not see modifications made by the concurrent outer transaction, nor should the outer transaction see the results committed by an inner transaction. In essence, the situation is closer to independent, nonnested transactions.

2.2.3   Exceptions

An exception thrown from within a transaction that leaves the scope of the transaction can either terminate or abort the transaction. An exception that *terminates* a transaction is similar to a nonlocal goto statement that terminates and exits the transaction. Before control leaves the transaction, the system attempts to commit the transaction. In the following example, the value that is printed is 2:

```
int x = 1;

try {
    atomic {
        x = 2;
        throw new AtomicTerminatingException();
    }
} catch(Exception) { print(x); };
```

An exception that *aborts* a transaction causes the transaction to abort as control leaves its scope. In the following example, the value that is printed is 1:

```
int x = 1;

try {
    atomic {
        x = 2;
        throw new AtomicAbortingException();
    }
} catch(Exception) { print(x); };
```

Another design decision is whether an exception of this type (or an `abort` statement) should cause the transaction to reexecute or whether it should terminate it. The latter alternative is more flexible, since a programmer can reexecute the atomic block by catching the exception and looping.

A TM system, with appropriate language and compiler support, can implement both models by providing different classes of exceptions for each alternative, different variants of 'throw' statements, or different forms of 'try' or 'atomic' blocks.

An exception caught within a transaction does not terminate the transaction, so the following code leaves x with value 4 and does not print the value:

```
int x = 1;

try {
    atomic {
        x = 2;
        try {
            throw new AtomicAbortingException();
        } catch {
            x = 4;
        }
    }
} catch(Exception) { print(x); };
```

2.3   TRANSACTIONAL MEMORY SYSTEM TAXONOMY

In addition to the language features discussed above, implementation differences in transactional memory systems can affect a system's programming model and performance. This section discusses some key differences relevant to both hardware and software transactional memory.

2.3.1   Transaction Granularity

Transaction granularity is the unit of storage over which a TM system detects conflicts. Most STM systems extend an object-based language and implement *object granularity*, which detects a conflicting access to an object even if the transactions referenced different fields. Other alternatives are *word granularity* or *block granularity*, which detect conflicting accesses to a memory word or adjacent, fixed-size group of words. Most HTM systems are agnostic of language characteristics and detect conflicts at block or word granularity. For convenience of discussion in this section, we will refer to a unit of shared resource as an "object," regardless of whether the unit is a word, block, data structure, or object in an object-oriented language.

Object granularity has implementation and comprehension advantages. A TM system must associate *metadata* with each item that it tracks. In some languages, it is easy to extend an object with a field to provide quick access to the TM metadata for the object. By contrast, an STM system must record metadata about a word or block of words in an auxiliary table, such as a hash table. In this case, mapping from a memory address to metadata can be expensive. HTM systems can associate metadata directly with the data; for example, a data cache entry can store metadata along with the cache line.

Moreover, object granularity provides a more understandable performance model than block granularity, since the memory layout of objects may not be visible to programmers, who can be confused by conflicts that arise when more than one object reside in a memory block.

However, word or block granularity permits finer grain sharing than do objects. This consideration is particularly important for aggregate data structures, such as arrays, which many transactions may concurrently access if they can be logically partitioned. Treating an array as a single object for conflict detection can cause unnecessary conflicts, which inhibit concurrency.

Finer grained approaches generally do not require changes to structure or object layout, the lack of which facilitates interoperability with nontransactional software and allows legacy and nontransactional-aware code to run inside a transaction. In particular, C and C++ impose severe constraints on TM implementations since structure layouts are fixed and the languages allow pointers into the interior of structures. However, even safe, object-oriented programs manipulate nonobject data, such as an integer static variable, that may require fine granularity.

2.3.2   Direct and Deferred Update

A transaction typically modifies an object. If the TM uses *direct update*, the transaction directly modifies the object itself and the system uses some form of concurrency control to prevent other transactions from concurrently modifying the object or committing after reading an updated value. Direct update requires that the system record the original value of a modified object, so it can be restored if the transaction aborts.

If a TM uses *deferred update*, the transaction updates the object in a location private to the transaction. The transaction must ensure that it reads the object's updated value from this location. Other transactions can concurrently modify their private copies of the object. When a transaction commits, it updates the actual object from its private copy. The object can be *updated in place* by copying values from the private copy, or it can be replaced by the private copy. If the transaction aborts, it discards the private copy.

Early STM systems used deferred update, but some recent systems use direct update, which appears to be more efficient. Direct update eliminates one or two memory references per read or write. In addition, it can reduce the amount of work needed to commit a transaction, by eliminating the need to copy from a private copy to an object.

On the other hand, direct update requires a mechanism to reverse (undo) memory updates and to impose concurrency control on memory accesses. Direct update increases the cost of aborting a transaction, since every modified object must be restored. Since aborting is intrinsically expensive, because it discards computation, it is sensible to favor the performance of commits over aborts.

Deferred update can increase a program's memory usage by making private copies of every object a transaction modifies. On the other hand, direct update must log the original value of the object. Logging may require less memory if objects are only partially modified.

The tradeoffs for HTMs differ slightly if a transaction executes successfully. An HTM can use the hardware caches that exist in processors for seamless and fast deferred updates. A cache stores a private copy of a memory location, and memory references first access the cached copy. Further, both commits and aborts can be local operations in a cache.

However, when a transaction exceeds the cache and overflows into memory, the tradeoffs for a deferred or direct update HTM become similar to that of an STM. Some HTMs support a mixture of deferred and direct updates, in which updates are deferred as long as data fits the caches. These systems use direct update when data overflows the cache.

Data races, including false data races, complicate the implementation of both systems.

Buffering and logging can have subtle interactions with isolation in a TM system, especially if buffering and logging occurs at a coarser granularity than updates in a transaction. Consider, for example, a non-transactional thread and a transactional thread updating two different fields of the same object. Since the locations differ, the updates do not conflict and the threads are not involved in a data race. Suppose, however, that the TM system records an entire object for undo. Care is required to ensure the non-transactional updates are not undone if the transaction aborts. This scenario is similar to false sharing.

In a TM system that provides weak isolation, the non-transactional access would not detect a conflict with a rollback and might observe an anomaly due to the data race. Even though HTM systems may buffer or log at the granularity of cache lines, HTM systems that provide strong isolation would detect the conflict and not observe an anomaly.

TM systems may need to ensure that a transaction's modifications become visible to other transactions in an order consistent with program order. For example, in a deferred-update system, the transaction should not release ownership of locations until all of them have been updated, so another transaction does not read an inconsistent intermediate state. A direct-update system has the opposite problem, when a transaction aborts, the system must make sure other transactions do not read the partially rolled back state.

TM systems that support weak isolation face a challenge in making the behavior of a deferred-update system consistent with the underlying hardware's memory model in the presence of data races. Updating locations during commit in the original program order requires capturing the order in which locations are modified, an expensive proposition. Direct-update systems face a similar challenge when they abort a transaction. In a weak isolation system, a conflicting read (data race) from a non-transactional thread might see an intermediate value that was later undone when a transaction aborted. A TM should probably deviate from legacy memory model behaviors in the presence of data races, because the cost of enforcing this behavior

may be disproportional to the benefit, which accrue primarily to incorrect programs. The issue of data races and TM semantics was briefly discussed earlier in Section 2.1.2.

2.3.3   Concurrency Control

A TM system that executes more than one transaction concurrently requires synchronization to mediate concurrent accesses to an object. This is necessary both in a direct update system, where transactions directly modify an object, and in a deferred-update system, in which the commit operation modifies the object.

A *conflict occurs* when two transactions perform conflicting operations on the same object. At least one of the transactions must modify the object, since two read accesses do not conflict. The conflicting transactions may not access a common memory location if the granularity of conflict detection is a block or an object.

The conflict is *detected* when the underlying system (SW, HW, or a combination) determines that the conflict occurs.

The conflict is *resolved* when the underlying system or code in a transaction takes some action to ensure correctness, by delaying or aborting one of the conflicting transactions.

These three events (conflict, detection, resolution) can occur at different times, but not in a different order—at least until systems predict or otherwise anticipate conflicts.

Broadly, there are two approaches to concurrency control.

With *pessimistic concurrency control*, all three events occur at the same point in execution; as a transaction is about to access a location, the system detects a conflict, and resolves it. This type of concurrency control allows a transaction to claim exclusive ownership of an object prior to proceeding and prevent other transactions from accessing it. Exclusive access can lead to a deadlock, in which each of two transactions holds exclusive access to an object and wants access to the object held by the other transaction. Deadlocks can be avoided by acquiring exclusive access to objects in a fixed, predetermined order, or they can be resolved by aborting a transaction in the deadlock cycle.

With *optimistic concurrency control*, conflict detection and resolution can happen after a conflict occurs. This type of concurrency control allows multiple transactions to access an object concurrently. It only must detect and resolve any conflicts before a transaction commits, which provides considerable implementation leeway. A conflict is resolved by aborting and restarting or by delaying one of the conflicting transactions. The other conflicting transaction can continue execution. If conflicts are infrequent, optimistic concurrency control can eliminate the cost of locking and increase concurrency by allowing more concurrency (Section 2.3.4).

Hardware systems can defer detecting conflicts to increase concurrency. For example, consider two transactions T_1 and T_2 executing concurrently on two different processors. T_1 reads a location X and T_2 writes location X. Assume that both processors cache location X in

a shared state. When the transactions issue their respective read and write requests, a conflict occurs.

However, neither processor must check for the conflict. T_1's read returns the cached value. T_2's write is queued into its store buffer. The coherence protocol eventually processes this write request. At this point, the protocol sends the write to T_1 and the coherence protocol detects the conflict. However, when T_2's write arrives at T_1, T_1 may have committed. Thus, even though both processors support early conflict detection, both transactions can optimistically commit successfully. Moreover, if T_1 has not committed and T_1 does not miss in its cache before it commits, T_1 is free to ignore the incoming invalidation because its state remains consistent and the execution of the transactions is still serializable. Thus, even though two transactions issue conflicting requests, both can commit.

Most HTM and STM systems use optimistic concurrency control. Some TM systems use hybrid mechanisms that combine aspects of both policies. For example, a common approach in direct update STMs is to use an exclusive lock to exclude other transactions that want to modify an object, but to allow readers to proceed without locking and to detect read–write conflicts later by validating a transaction's read set (Section 2.3.4). This approach simplifies the locks, as fair read–write locks are complex and more expensive than mutual-exclusion locks [21], but may change the semantics of transactions (Section 2.3.5).

Another dimension to concurrency control is its forward progress guarantees: the assurance that a transaction seeking to access an object will eventually complete its computation. Again, there are two alternative approaches.

Blocking synchronization is the most familiar form of concurrency control embodied in conventional locks, monitors, and semaphores. Programs constructed with blocking synchronization do not guarantee the forward progress of a system if threads may be preempted while holding exclusive access to resources. The system could deadlock or be delayed for an arbitrary period as one thread waits for a lock held by another thread.

Early TM systems grew out of research in nonblocking data structures, and naturally, these TM systems were constructed with *nonblocking synchronization*. This mechanism offers a stronger guarantee of forward progress: a stalled thread cannot cause all other threads to stall indefinitely. From a practical viewpoint, three most common guarantees are as follows:

- *Wait freedom.* This is the strongest guarantee that all threads contending for a common set of objects make forward progress in a finite number of their individual time steps.

- *Lock freedom.* This guarantee assures that at least one thread from the set of threads contending for a common set of objects makes forward progress in a finite number of time steps of any of the concurrent threads.

- *Obstruction freedom.* This is the weakest guarantee that a thread will make forward progress in a finite number of its own time steps in the absence of contention over shared objects. It is particularly well suited to transactions, as it allows a thread involved in a conflict to be aborted and retried.

Experience with STM systems, at least to date, suggests that those constructed with nonblocking synchronization are more complex and lower performing than those built with blocking synchronization; in part, because the former group uses deferred update and the latter uses direct update and in part because of the increase in memory traffic caused by nonblocking techniques.

STM systems can be built with blocking synchronization, and use timeouts to detect and abort deadlocked or blocked transactions. Systems built this way can provide higher performance, while maintaining strong forward progress guarantees for users of an STM system [12, 13, 22–24]. This claim should not be surprising, as database systems constructed with blocking synchronization provide database programmers with a nonblocking abstraction (transactions).

2.3.4   Early and Late Conflict Detection

A TM system can detect a conflict at three points in a transaction's execution:

- A conflict can be *detected on open*, when a transaction declares its intent to access an object (by "opening" or "acquiring" the object) or at the transaction's first reference to the object.
- Conflict can be *detected on validation*, at which point a transaction examines the collection of objects it previously read or updated, to see if another transaction opened them for modification. Validation can occur anytime, or even multiple times, during a transaction's execution.
- A conflict can be *detected on commit*. When a transaction attempts to commit, it may (often must) validate the set of objects that it read and updated to detect conflicts with other transactions.

HTM systems built on cache coherence typically detect conflicts early. A processor issues a memory request, and the cache coherence protocol sends it to the appropriate caches. If the HTM detects and resolves a conflict, the failing HTM transaction can abort immediately. Some HTMs do not abort a transaction immediately but leave an indication of a conflict in the hardware status and rely on software to check the status (detected on validation) and abort a transaction. Some HTM systems add specialized support in a cache controller to detect conflicts on commit.

A system may treat write–write or read–write conflicts differently. The former may be easier to detect, since an object may only have a single writer, whose identity may be stored with the object. Many systems do not track the identity of the transactions reading a data object, since the size of this collection of transactions may be unbounded, which makes it difficult to record in a fixed-size structure.

Either a transaction can validate an object by comparing its value against the value recorded when the transaction started or by recording the version number of an object, which must be updated when a transaction modifies the object. Version numbers avoid the "ABA" problem:

- Object O's initial value is "A."
- Transaction T_1 reads this value and records it in its read set.
- Transaction T_2 changes O's value to "B" and commits.
- Transaction T_1 reads O again, but this time sees the value "B."
- Transaction T_3 changes O's value back to "A" and commits.
- Transaction T_1 commits and validates its read set. Since O's value has returned to "A," the transaction will validate, even though it read two, inconsistent values for O.

Version numbers have a boundary condition, when a counter wraps around. This should be an infrequent occurrence with a suitably large counter, but must be handled correctly.

Detecting a conflict *early* (on open or validation before commit) reduces the amount of computation lost by the aborted transaction. On the other hand, aborting a transaction early can terminate a transaction that could have committed. For example,

- suppose transactions T_B and T_C conflict with transaction T_A over two different objects;
- T_B is aborted as soon as its conflict with T_A occurs;
- T_A is aborted because of a conflict with T_C;
- with T_A terminated, T_B could have completed execution.

Late detection could have avoided this problem, by waiting until a transaction attempts to commit its results before determining if the transaction is in conflict. However, late detection maximizes the amount of computation discarded when a transaction aborts. A *doomed transaction* cannot commit because of conflicts.

Detecting conflicts requires a TM system to maintain an association between transactions and the objects they access. There are two approaches to tracking this relation:

- A transaction can record the set of objects that it read or updated. A *read set* or a *write set* can be *private* to a transaction (also known as *invisible*), in which case other transactions

cannot examine the set to find potential conflicts. The set can also be *public* (*visible*), in which case other transactions can examine the objects.

- The TM system can record the transactions that have an object open for reading (in a *reader set*) or open for update (in a *writer set*).

Consider two transactions T_A and T_B that access the same object. Scott [8] identifies four practical policies for detecting conflicts (Fig. 2.1):

- *Lazy invalidation.* Transactions T_A and T_B conflict if T_A writes an object, T_B reads the same object, and T_A commits before T_B. Systems such as Harris and Fraser's WSTM System (Chapter 4) use this policy to provide late conflict detection.

- *Eager W–R.* Transactions T_A and T_B conflict if T_A and T_B have a lazy invalidation conflict or T_A writes an object, T_B subsequently reads the same object, but neither transaction has committed.

- *Mixed invalidation.* Transactions T_A and T_B conflict if T_A and T_B have a lazy invalidation conflict or T_A writes an object, T_B reads and then writes the same object, but neither transaction has committed.

- *Eager invalidation.* Transactions T_A and T_B conflict if T_A and T_B have an eager W–R conflict or T_B reads an object, T_A subsequently writes the same object, but neither transaction has committed. Systems, such as Herlihy et al.'s DSTM System (Chapter 4), use this policy to provide early conflict detection.

Fig. 2.1 also shows the relationship among these four policies. The circles represent the set of transaction executions each policy can identify as conflicting. Therefore, for example, eager invalidation subsumes the other policies, but not all pairs of transactions identified as conflicting by the eager W–R policy are conflicts under the lazy invalidation policy.

2.3.5 Detecting and Tolerating Conflicts

Most TM systems allow multiple transactions to read an object concurrently, which makes it impossible to track reader sets in a fixed-size data structure. A TM system that does not track readers (also called an *invisible read* TM system) cannot immediately detect a write-after-read conflict, which opens the possibility that a transaction will continue executing after such a conflict and consequently find itself running with inconsistent state in which some values changed after the transaction started. Inconsistencies of these sorts should cause a transaction to terminate, but few STM systems detect these problems immediately.

As an example, suppose that

- transaction T_R reads objects O_1 and O_2 and transaction T_W updates O_1 and O_2;
- T_R first reads O_1;
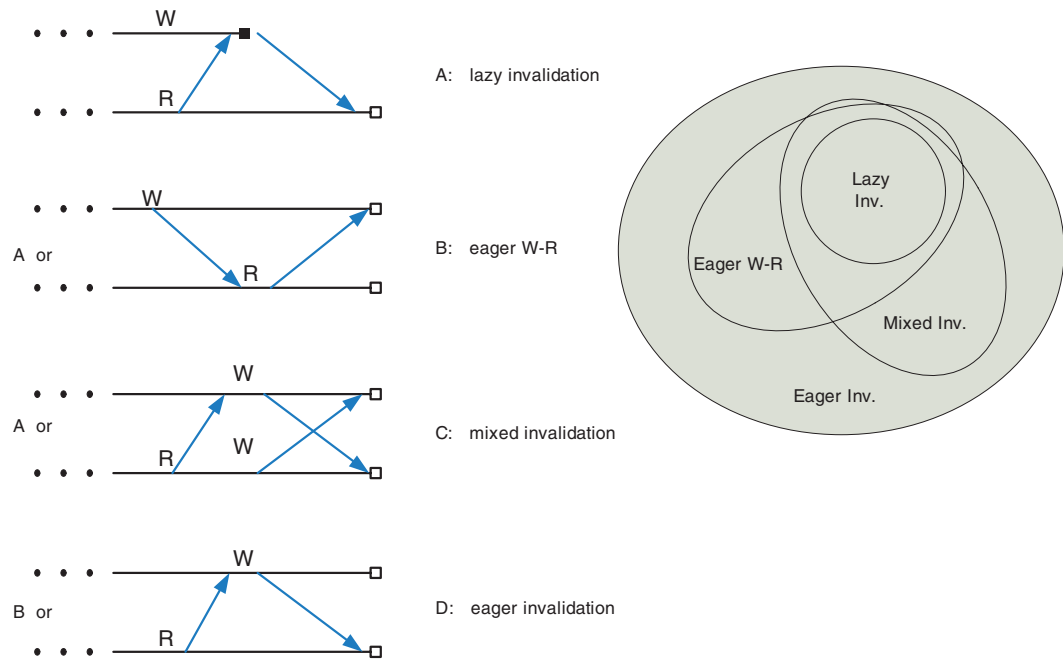
FIGURE 2.1: Scott's conflict classes

- T_W then executes and modifies O_1 and O_2;
- T_R resumes execution and reads O_2; the state of the two objects is inconsistent, and T_R should abort since its execution is no longer serializable.

Inconsistent state can cause incorrect or unexpected program behavior. For example, suppose both objects' value was initially nonnull, T_W set the values to null, and T_R executed the statement:

```
if (O1 != null)
   O2.f1 = 2;
```

T_R would evaluate the predicate with `O1` as nonnull, but when it executed the next statement, `O2` would be null, which produces a paradoxical null-pointer exception.

TM systems use three mechanisms to avoid problems with inconsistent updates:

- *Validation.* A transaction can validate each object in its read set by checking if another transaction modified an object. If all objects in a transaction's read set validate, the transaction's state is consistent.

In many deferred-update TMs, validating the read set when opening an object prevents inconsistent execution, since an update does not change a previously opened object. Inconsistencies between two objects, as in the example above, become apparent when opening the second object.

In a direct update TM or a deferred-update TM that replaces objects in place, an object may change between consecutive reads, so validation only assures that execution up to the validation itself is consistent. The TM must tolerate inconsistent execution (below).

Validating the entire read set when opening an object can be expensive, since the number of validations is $O(N^2)$, where N is the number of items in the set.

All invisible–read TMs must validate the read set before committing a transaction, to ensure that its execution is serializable.

- *Invalidation.* Another approach is to track the transactions that read an object (or a subset of these transactions, as in [25]), and abort these transactions when another transaction opens the object for updating.

- *Inconsistency toleration.* Because validation is potentially expensive and, in the case of a direct update TM, provides no assurance that subsequent execution remains consistent, transactions may execute with an inconsistent state. So long as an inconsistent transaction does not affect other threads and the transaction eventually validates its read set and aborts, inconsistency does not affect program correctness.

Inconsistent execution, however, may manifest itself by exceptions (null pointer, array bounds, type casting, etc.) in a safe programming language, such as Java or C#. The system can distinguish these exceptions by validating the read set. If the state is inconsistent, the exception should abort the transaction.

HTM systems can also suffer from inconsistent execution. An HTM system may not signal a transaction immediately on a conflict, but instead allow the transaction to continue executing until it validates (for example, by testing a hardware flag to determine the conflict occurred).

In unsafe languages, such as C or C++, inconsistency can cause incorrect behavior without raising an exception. This behavior can corrupt memory and affect the execution of other threads, before the inconsistency manifests itself. Validating before every memory write could detect this problem, but performance would be poor.

Moreover, in all languages, a program executing a loop or recursive call sequence whose termination might depend on values read from shared memory must periodically validate—or

else an inconsistency could cause an infinite loop. Consider the following two transactions (x=y=0; initially):

Transaction T$_1$:	*Transaction T$_2$:*

```
Transaction T₁:                        Transaction T₂:
atomic {                               atomic {
   temp1 = x;                            x = 10;
   // Transaction 2 runs here            y = 10;
   temp2 = y;                          }
   while (temp1 != temp2) { };
}
```

Suppose transaction T_1 executes its first statement and then T_2 runs in its entirety. T_1 will loop forever, and will not raise an exception. This problem occurs in both deferred and direct update TM systems. Fixing the problem requires that either T_1 validate its read set on reading y, validate its read set on the loop backedge, or expose its read set so that T_2 can detect the conflict.

However, validation and toleration are not always sufficient. In some TM implementations (e.g., those in Sections 3.5.2 and 3.5.4) that do not track publicly, but maintain private sets and rely on inconsistency tolerance, the program behaviour may not always be correct. Consider the following example,

```
Transaction T₁:                        Transaction T₂:

ListNode res;                           atomic {
atomic {                                   ListNode n = lHead;
   res = lHead;                            while (n != null) {
   if (lHead != null)                         n.val ++;
      lHead = lHead.next;                      n = n.next
}                                          }
use res multiple times;                 }
```

Transaction T_1 removes the first element from a list and then uses it outside the atomic block. It seems reasonable that code inside an atomic block should be able to remove an item from a shared structure and make it private; for example, if the list implements a queue distributing work among concurrent threads. Transaction T_2 iterates over the list, modifying each element. The transactions obviously conflict, but suppose they execute as follows:

- T_2 starts executing and stores lHead in a local variable in the atomic block.
- T_1 executes its atomic block.

- T_1 references `res` (the node it removed from the list) outside the transaction.
- T_2 iterates over the list, from its original head, and modifies each node.
- T_1 references `res` again and sees the modified value.
- T_2 attempts to commit, but validation fails because T_1 changed the head of the list.
- T_2 aborts.
- T_1 references `res` again and sees the unmodified value.

The correct behavior of this code is that either T_2 increments the node before T_1 removes it from the list or T_2 does not modify the node. This is the behavior when atomic blocks are implemented with the semantics-defining technique of single-lock atomicity. With a single lock, the interleaving above cannot arise since the execution of T_1 and T_2 will not overlap.

Even aggressive validation fails to resolve this conflict in a predictable manner, which means that many existing STM systems, both direct and deferred update, can fail in the manner above. Other implementation techniques produce the expected result. A lock-based implementation would not have data races and would produce equivalent correct executions. An exposed read TM would also work correctly since T_1 could abort T_2 before it modifies the list. Strong isolation (for example, in an HTM system) is another potential solution, since it detects the conflict between T_2 and the statements following T_1.

This example demonstrates the need for additional research to specify formally the behavior of transactional memory and to verify that the algorithms in an STM system are correct and are correctly implemented.

2.3.6   Contention Management

A conflict between two transactions over an object can be resolved by aborting either one. A TM system typically has a *contention manager*, which implements one or more *contention resolution policies* that decide which conflicting transaction to abort. The choice of which transaction to abort (or, alternatively, to delay) can affect the performance of the system and its semantic guarantees. For example, not all contention resolution policies guarantee fairness—in the sense that a given transaction will eventually prevail despite repeated conflicts. Scherer and Scott describe a number of contention resolution policies and provide examples to show that no policy is uniformly better than all other polices [26]. Guerraoui describes a hierarchical classification of contention managers, based on their cost [27].

A key concern for transactional memory is forward progress. Processor instructions are guaranteed to complete, no matter what else executes. Even atomic read-modify-write sequences (for example, compare-and-swap (CAS)) guarantee forward progress at the instruction level. At higher levels of abstraction, these primitives can construct synchronization methods that

provide limited guarantees of forward progress. HTM systems that provide transactional memory of bounded size have proposed hardware mechanisms to ensure varying degrees of forward progress. These vary from ad hoc policies to fair timestamps. Providing forward progress over operations that access large sets of memory locations typically requires contention management support from software, even in an HTM system.

Another question concerns the responsibility for contention management. Should a programmer specify policies for each transaction? Or, will a TM system be able to select contention resolution polices automatically?

2.4   PROGRAMMING AND EXECUTION ENVIRONMENT

For a new programming abstraction like transactional memory to become successful, it must fit into existing programming and execution environments, it must interoperate with existing libraries and services, and it must provide a usage model with few, if any, surprises in functionality and behavior. Below, we discuss various aspects of modern execution environments that interact with transactional memory.

2.4.1   Communication

Transactional memory, as its name implies, is an abstraction for manipulating data structures in memory. Because TM also affects a program's control flow—by aborting and reexecuting statements—it also changes the way in which the program interacts with the external world.

Perhaps the most serious challenge in using transactions is communication with entities not under the control of the TM system. Modern operating systems such as Unix and Windows provide a very large number of mechanisms for communication, including system calls on the operating system, file manipulation, database accesses, interprocess communication, network communication, etc.—many of which are buried in libraries or other programming abstractions. These operations cause changes in entities not under the control of a TM system, so if a transaction aborts, the TM system may have no way to revert these effects.

While there is no general mechanism to undo these changes, solutions exist on a case-by-case basis. A system can buffer operations such as file writes until a transaction commits, and then write the modified blocks to disk. Similarly, the system could also buffer input, to be replayed if the transaction aborted and reexecuted. These seem like simple solutions, until we combine them in a transaction that both reads and writes. Suppose the transaction writes a prompt to a user and then waits for input. Because the output is buffered, the user will never see the prompt and will not produce the input. The transaction will hang.

Another solution is to allow IO operations in transactions only if the IO supports transactional semantics, so the TM system can rely on another abstraction to revert changes. Databases

and some file systems are transactional. However, the granularity of these systems' transactions may not match the requirements of an atomic block. For example, Windows Vista supports transactional file IO. These transactions start when a file is first opened. Therefore, if an atomic block only performs one write, it is not possible to use a file system transaction to revert this operation, without discarding all other changes to the file.

Many systems support transaction processing monitors (TPMs) [28], which serve as a coordinator for a collection of systems, each of which supports transactions and wants to ensure that the operation of the collection appears transactional to all parties outside the group. TPMs generally use a two-phase commit protocol, in which the constituent transactions first all agree that they are ready to commit their transactions (and all abort if any wants to abort), and then commit en masse.

Another approach is to use compensating actions to undo the effects of a transaction. For example, a file write can be reverted by buffering the overwritten data and restoring it if a transaction aborts. Compensation is a very general mechanism, but it puts a high burden on a programmer to understand the semantics of a complex system operation and be able to revert it. This becomes particularly difficult in the presence of concurrency, when other threads and processes may be manipulating the same system resources.

A number of papers discuss transaction handlers that invoke arbitrary pieces of code when a transaction commits or aborts [3, 20, 29, 30]. These handlers can interface transactions to TPMs or other transactional systems and implement compensating actions to revert external side effects.

Similar challenges exist when a transaction performs a system call to the operating system. Even if the call does not cause IO operations, it may update kernel data structures. If the transaction subsequently aborts, kernel state may not match program state. Furthermore, the operating system is a shared system resource, so a user transaction should not interfere with unrelated user transactions through system calls. These considerations suggest that operating system kernels may need to support transactions, or transactions will need to be tightly circumscribed in their interaction with the system.

Another alternative is to serialize the execution of all external communications, by allowing only a single transaction to communicate at a time [31]. This transaction becomes nonabortable and always runs to completion without a rollback. However, eliminating overlap among IO operations may greatly reduce the performance and responsiveness of a system.

2.4.2   System Abstractions

Systems use software to implement complex abstractions, which appear as primitive operations to a program. Two common examples are virtual memory, implemented by an operation system's paging code, and garbage collection, implemented by a run-time system's garbage collector.

Although the code for these abstractions typically runs on the same processor as transactions, these abstractions probably should not execute transactionally.

Consider, for example, garbage collection. If a garbage collection occurs in the middle of a transaction, we do not want the storage reclamation to be reverted, even if the transaction has aborted. There are three ways to handle this situation.

The first is to distinguish between instructions that execute transactionally and those that do not. A garbage collector could use the latter type of instructions and so its changes would persist even if the surrounding transaction aborted. Of course, garbage collection moves data around in memory, and the collector must communicate these changes to the transactional memory system. Updating an STM system's metadata typically is not a problem, since the system's structures are conventional data updated by the collector. However, for HTMs with hidden transactional state, the garbage collector needs a mechanism to update the addresses of transactional objects that moved.

The next alternative is to treat the garbage collection as if it was executing in an open nested transaction. The garbage collector, if it completes successfully, will commit, and its changes will then be visible to all executing transactions, including the surrounding one. This approach puts a large burden on the transaction mechanism, as a garbage collector may access far more memory than a typical transaction.

The final approach is to abort all transactions if a page fault, context switch, or garbage collection occurs and then restart them after the operation completes. This approach has the advantage of simplicity, since it takes advantage of a necessary mechanism to avoid more complex mechanisms or semantics. However, if page faults or garbage collections are frequent, the approach could reduce performance. Perhaps more important, it also reduces the likelihood that a long-running transaction will complete and so puts an upper bound on the size and duration of permissible transactions.

2.4.3   Existing Programming Languages

Programming languages were designed without consideration of transactions. These languages contain many features that might interact poorly with transactions. However, since these features are standardized and existing programs use them, it is difficult to eliminate, change, or prohibit them.

Consider an object-oriented language running in a managed environment. Object finalizers are pieces of code associated with an object that are supposed to execute just before the garbage collector reclaims the object. Although finalizers are a source of considerable confusion and many errors, they are widely used to close file descriptors and free system resources [32]. Suppose an object containing a finalizer is created in a transaction. When and how does the finalizer run? If the transaction aborts, presumably the finalizer need not run, since the side

effects caused by creating the object (i.e., opening a file) should be reverted by the transaction. However, system resources may not all be transactional, so running the finalizer may be the only way in which the resource will be reclaimed. If the object is reclaimed when the transaction is still executing, should the finalizer run in the transaction? Alternatively, if the transaction commits and the object subsequently becomes garbage, how is the finalizer run? Does it execute in a transaction? In Java, a class may be loaded when first accessed. The class loading may also invoke a JIT compilation and a complex series of interactions with the underlying managed system. If the first access to a class occurs inside a transaction, what happens if the transaction aborts? Can the original semantics of class loading be maintained? In C#, managed code can pass a pointer to a managed object to native (unsafe) code. How do managed and native code interact if a native method is called inside a transaction? These design questions do not have clear answers.

2.4.4   Libraries

Another challenge is separately compiled and dynamically linked software libraries. Libraries are an integral part of any software and transactions need to call precompiled or newly compiled libraries. Today, we link libraries from different compilers and languages into a single application. Can we achieve this level of interoperability in the presence of transactional memory? Can a compiled library separate semantics from implementations? Can we compile libraries that do not carry implementation details of a particular TM system? If implementation details are embedded into a library, will a system need multiple versions of a library for different TM run-time systems? How do we deliver libraries to be used both inside and outside a transaction?

2.4.5   Synchronization Primitives

A further challenge is the numerous existing synchronization primitives. Transactional memory must safely coexist and interoperate with code that uses primitives such as conventional lock-based synchronization and atomic operations such as atomic increment and decrement. Introducing transactional memory should not break existing software synchronization. Does this require transactions to observe a locking protocol before referencing data? An alternative is to translate code that relies on locks to use transactions. This approach has subtleties that make automatic translation a challenge [5, 33].

2.4.6   Debugging

Software development tools and development environments must evolve to support transactional memory. The concepts of optimistic or aborted execution do not exist in today's tools.

Debugging is a critical issue. What does it mean to single step through an atomic transaction? A breakpoint inside the transaction should expose the state seen by the transaction. However, how does a debugger present a consistent view of a program's state, since part of the state not relevant to the transaction, but visible to the debugger, may have been modified by other threads? Furthermore, in deferred-update systems, transactionally modified state resides in two places: the original location and a buffered location. The debugger must be aware of this separation and able to find appropriate values. In addition, how does a programmer debug a transaction that aborts because of conflicts? Lev and Moir discuss the challenges of debugging transactional memory [34].

2.4.7   Performance Isolation

Programs today expect some degree of isolation from other programs executing on the same system. Performance isolation is not guaranteed, but an operating system can deschedule or terminate an application that consumes inappropriate amounts of system resources. Transactional memory must provide similar degrees of isolation. Performance isolation is not an issue for STMs, but using hardware support can introduce performance isolation challenges because of shared hardware resources. Zilles and Flint [35] explore the issue of performance isolation.

2.4.8   HTM Implementation

Modern processors employ numerous mechanisms to improve performance, such as pipelining, branch prediction, superscalar and out-of-order execution, load and store buffers, and caches. Processors use these features to build an instruction window and to keep the pipeline busy and flowing smoothly. Operations such as serializing instructions disrupt pipeline flow and hurt performance. Frequent interruptions can prevent a processor from fully extracting performance from the instruction stream and can limit scalable performance [36]. Hardware support for transactional memory—including new hardware features and instructions to utilize these features—must integrate seamlessly into modern processor implementations and instruction set architectures.

As transactions become common, it is important to ensure that they do not limit the ability of a processor to execute a sequential instruction stream efficiently. This raises a number of questions. How do transactions interact with existing processor features? Do they introduce complexity in high-performance implementations? Can these mechanisms avoid disrupting pipeline flow? Do the new mechanisms affect a critical path? How do these mechanisms and instructions interact with the load and store buffers and memory ordering? How do you ensure correct execution for these instructions in the presence of branch mispredictions and speculative execution? Commercial implementations of instructions for optimistic synchronization imposed

numerous restrictions that limited their utility (Chapter 4, Section 2.3.3). Hardware support for transactional memory faces similar challenges.

In addition, transactions increase the difficulty of validating a processor, not just the hardware but also the instruction set extensions and the overall software stack running on this hardware.

REFERENCES

[1]   Harris, T. and K. Fraser, *Language Support for Lightweight Transactions*, in *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 03)*. 2003: Anaheim, CA. pp. 288–402 doi:10.1145/949305.949340

[2]   Harris, T., et al., *Composable Memory Transactions*, in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2005: Chicago, IL. pp. 48–60 doi:10.1145/1065944.1065952

[3]   Carlstrom, B.D., et al., *The Atomos Transactional Programming Language*, in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*. 2006, ACM Press: Ottawa, Ontario, Canada. pp. 1–13 doi:10.1145/1133981.1133983

[4]   Hoare, C.A.R., *Monitors: An Operating System Structuring Concept.* Communications of the ACM, 1974: pp. 549–557 doi:10.1145/355620.361161

[5]   Blundell, C., E.C. Lewis, and M.M.K. Martin, *Deconstructing Transactions: The Subtleties of Atomicity*, in *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. 2005. http://www.cis.upenn.edu/acg/papers/wddd05_atomic_semantics.pdf.

[6]   Scott, M.L., *Sequential Specification of Transactional Memory Semantics*, in *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*. 2006: Ottawa, Canada. http://www.cs.rochester.edu/u/scott/papers/2006_TRANSACT_formal_STM.pdf.

[7]   Papadimitriou, C., *The Theory of Database Concurrency Control*. 1986, Rockville, MD: Computer Science Press..

[8]   Herlihy, M. and J.M. Wing, *Linearizability: A Correctness Condition for Concurrent Objects.* ACM Transactions on Programming Languages and Systems, 1990. **12**(3): pp. 463–492 doi:10.1145/78969.78972

[9]   Adve, S.V., *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. 1993, Computer Sciences Department, University of Wisconsin–Madison.

[10] Gharachorloo, K., *Memory Consistency Models for Shared-Memory Multiprocessors*. 1995, Computer System Laboratory, Stanford University.

[11]   Netzer, R.H.B. and B.P. Miller, *What Are Race Conditions? Some Issues and Formalizations.* ACM Letters on Programming Languages and Systems, 1992. **1**(1): pp. 74–88.

[12]   Lomet, D.B., *Process Structuring, Synchronization, and Recovery Using Atomic Actions*, in *Proceedings of the ACM Conference on Language Design for Reliable Software*. 1977: Raleigh, NC. pp. 128–137 doi:10.1145/800022.808319

[13]   Adl-Tabatabai, A.-R., et al., *Compiler and Runtime Support for Efficient Software Transactional Memory*, in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*. 2006, ACM Press: Ottawa, Ontario, Canada. pp. 26–37 doi:10.1145/1133981.1133985

[14]   Harris, T., et al., *Optimizing Memory Transactions*, in *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2006)*. 2006, ACM Press: Ottawa, Ontario, Canada. pp. 14–25 doi:10.1145/1133981.1133984

[15]   Blanchet, B., *Escape Analysis for Java: Theory and Practice.* ACM Transactions on Programming Languages and Systems, 2003. **25**(6): pp. 713–775 doi:10.1145/945885.945886

[16]   Boyapati, C. and M. Rinard, *A Parameterized Type System for Race-free Java Programs*, in *Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*. 2001: Tampa Bay, FL. pp. 56–69 doi:10.1145/504282.504287

[17]   Buhr, P.A. and A.S. Harji, *Implicit-signal Monitors.* ACM Transactions on Programming Languages and Systems, 2005. **27**(6): pp. 1270–1343 doi:10.1145/1108970.1108975

[18]   Lev, Y. and J.-W. Maessen, *Toward a Safer Interaction with Transactional Memory by Tracking Object Visibility*, in *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. 2005. http://hdl.handle.net/1802/2098.

[19]   Moss, J.E.B. and A.L. Hosking, *Nested Transactional Memory: Model and Preliminary Architecture Sketches*, in *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. 2005. http://hdl.handle.net/1802/2099.

[20]   Zilles, C. and L. Baugh, *Extending Hardware Transactional Memory to Support Non-busy Waiting and Nontransactional Actions*, in *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*. 2006: Ottawa, Canada. http://www-faculty.cs.uiuc.edu/~zilles/papers/non_transact.transact2006.pdf.

[21]   Mellor-Crummey, J.M. and M.L. Scott, *Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors*, in *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming* 1991. pp. 106–113

doi:10.1145/109625.109637

[22] Harris, T. and K. Fraser, *Revocable Locks for Non-blocking Programming*, in *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*. 2005: Chicago, IL. pp. 72–82 doi:10.1145/1065944.1065954

[23] Ennals, R., *Efficient Software Transactional Memory*. 2005, Intel Research Cambridge Tech Report: Cambridge, England. http://www.cambridge.intel-research.net/~rennals/051_Rob_Ennals.pdf.

[24] Ennals, R., *Software Transactional Memory Should Not Be Obstruction-Free*. 2006, Intel Research Cambridge Tech Report: Cambridge, England. http://www.cambridge.intel-research.net/~rennals/052_Rob_Ennals.pdf.

[25] Marathe, V.J., et al., *Lowering the Overhead of Nonblocking Software Transactional Memory*, in *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*. 2006: Ottawa, Canada. http://www.cs.purdue.edu/homes/jv/events/TRANSACT/.

[26] Scherer III, W.N. and M.L. Scott, *Advanced Contention Management for Dynamic Software Transactional Memory*, in *Proceedings of the Twenty-fourth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. 2005, ACM Press: Las Vegas, NV. pp. 240–248 doi:10.1145/1073814.1073861

[27] Guerraoui, R., M. Herlihy, and B. Pochon, *Polymorphic Contention Management*, in *Proceedings of the Nineteenth International Symposium on Distributed Computing (DISC)*. 2005, Springer Verlag: Krakow, Poland. pp. 303–323. http://lpdwww.epfl.ch/upload/documents/publications/neg–1700857499main.pdf.

[28] Bernstein, P.A., *Transaction Processing Monitors*. Communications of the ACM, 1990. **33**(11): pp. 75–86 doi:10.1145/92755.92767

[29] Harris, T., *Exceptions and Side-effects in Atomic Blocks*, in *Proceedings of the 2004 Workshop on Concurrency and Synchronization in Java programs*. 2004. pp. 46–53. http://research.microsoft.com/~tharris/papers/2004-csjp.pdf.

[30] McDonald, A., et al., *Architectural Semantics for Practical Transactional Memory*, in *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA)*. 2006. pp. 53–65 doi:10.1109/ISCA.2006.9

[31] Blundell, C., E.C. Lewis, and M.M.K. Martin, *Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions*. 2006, Department of Computer and Information Science, University of Pennsylvania: Philadelphia, PA. http://www.cis.upenn.edu/~eclewis/papers/tr06_unrestricted_transactions.pdf.

[32] Boehm, H.-J., *Destructors, Finalizers, and Synchronization*, in *Proceedings of POPL 2003: The 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2003: New Orleans, Louisiana. pp. 262–272 doi:10.1145/604131.604153

[33] Carlstrom, B.D., et al., *Transactional Execution of Java Programs*, in *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. 2005. http://hdl.handle.net/1802/2096.

[34] Lev, Y. and M. Moir, *Debugging with Transactional Memory*, in *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*. 2006: Ottawa, Canada.

[35] Zilles, C. and D. Flint, *Challenges to Providing Performance Isolation in Transactional Memories*, in *Proceedings of the Fourth Workshop on Duplicating, Deconstructing, and Debunking*. 2005. pp. 48–55. http://www-faculty.cs.uiuc.edu/~zilles/papers/xact_isolation.wddd2005.pdf.

[36] Chou, Y., B. Fahs, and S. Abraham, *Microarchitecture Optimizations for Exploiting Memory-Level Parallelism*, in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*. 2004, IEEE Computer Society: Munich, Germany. pp. 76–89. http://portal.acm.org/citation.cfm?id=1006708.

CHAPTER 3

Software Transactional Memory

3.1 INTRODUCTION

Software transactional memory (STM) is a software system that implements nondurable transactions with the ACI (failure atomicity, consistency, and isolation) properties for threads manipulating shared data. STM systems run, for the most part, on conventional processors; though some early systems postulated more aggressive synchronization primitives than implemented by current processors. Research on STM systems is distinct from the considerable previous work on persistent data structures, which focused on providing durable transactions (the missing "D") for in-memory data structures [1].

The performance of recent STM systems, particularly those integrated with an optimizing compiler, has reached a level that makes these systems a reasonable vehicle for experimentation and prototyping. However, it is still not clear how low the overhead of STM can reach without hardware support. STM systems, nevertheless, offer a number of advantages over HTM:

- Software is more flexible than hardware and permits the implementation of a wider variety of more sophisticated algorithms.
- Software is easier to modify and evolve than hardware.
- STMs can integrate more easily with existing (software) systems and language features, such as garbage collection.
- STMs have fewer intrinsic limitations imposed by fixed-size hardware structures, such as caches.

Given our limited experience implementing and using transactional memory, these considerations suggest that STM will play an important role in transactional memory. Approaches in which hardware accelerates STM operations that are frequent or expensive can help reduce STM overheads (Chapter 4).

3.1.1 Chapter Overview

A linear presentation of a rich set of research results inevitably becomes a compromise among competing imperatives. A chronological presentation can capture the large-scale evolution of

the area but may obscure smaller–scale interactions by separating related papers. Alternatively, grouping papers by topic raises questions of which are the most important dimensions of the research area and which contribution most clearly defines a paper.

The organization of this chapter is one such compromise. The chapter divides the STM research into four categories:

- Precursors to modern STM systems (Section 3.3).
- Deferred-update STM systems, in which a transaction modifies a copy of an object and updates the original object when the transaction commits (Section 3.4).
- Direct-update STM systems, in which a transaction modifies an object in-place (Section 3.5). These systems divide into those that use optimistic and pessimistic concurrency control.
- Language-oriented STM systems, which focus on integrating transactional memory into a programming language (Section 3.6).

Chapter 2 contains a taxonomy that outlines the main distinctions in STM systems. However, the difference between deferred- and direct-update systems appears to be fundamental, as the alternatives lead to very different implementation techniques and constrain many of the other design parameters. Hence, we will use them as broad categories for collecting and comparing similar research.

Fortuitously, this organization roughly parallels the evolution of the field—with the exception that the language research was intermixed with the development of deferred-update STM systems. In a linear arrangement, like a book, there is no good way to capture this relationship, except cross-references.

3.2   A FEW WORDS ON LANGUAGE

Papers in this chapter present the STM systems in many different programming languages. To improve clarity, when an implementation is described in detail, its code is recast in C#-like pseudolanguage [2], with a few extensions for low-level systems programming, such as type unions. The code generally follows a system's published description, both to avoid introducing errors and to make clear the connection with a paper. Consequently, the code is often incomplete and far from a paragon of programming style. It is as an algorithm sketch, rather than an implementation of a running system. Moreover, our implementations often rename variables and types to improve readability. However, we do not translate proposed language features for transactional memory programming.

The synchronization operations are compare-and-swap (CAS) and load-linked (LL) and store-conditional (SC). Section 4.3.3 describes the implementation of these instructions.

`CAS(a, o, n)` atomically replaces the contents at memory location `a` with `n`, if the location held the value `o`. Otherwise the location is left unchanged. `LL(a)` returns the value of memory location `a`. A subsequent `SC(a, n)` stores value `n` in location `a`, if not write to the location occurred since the previous LL operation. Unless noted, we assume the simple form of LL–SC that does not permit nesting of these operations and is the only one implemented in an actual processor.

The scope of identifiers is local to a particular system (or alternatively, a subsection of this paper). Most systems use similar naming conventions, e.g., `STMCommit`, but in this presentation there is no code sharing among the system descriptions.

3.3   STM PRECURSORS

STM systems evolved from research into better synchronization, in particular lock-free and nonblocking primitives, and hardware transactional memory. The former is a significant research area in its own right [3], which would take us too far afield to cover completely in this book. Chapter 4 discusses the latter.

The papers in this section are precursors of the STM systems because they either propose some of the key ideas used in these systems or are themselves preliminary attempts to build an STM system. Lomet, in 1977, proposed a programming language construct essentially identical to transactional memory (Section 3.3.1). Shavit and Touitou introduced the term "software transactional memory" and described one of the first implementations (Section 3.3.2).

3.3.1   Lomet, LDRS 77

Many of the concepts and implementation principles for STM were anticipated in a paper by Lomet in 1977 [4], which was published soon after the classic paper by Eswaran [5] on two-phase locking and transactions. Lomet reviewed the disadvantages and shortcomings of synchronization mechanisms, such as semaphores, critical regions, and monitors, and noted that programmers used these constructs to execute pieces of code atomically. His suggestion was to express the desired end directly and shift the burden of ensuring atomicity onto the system. An atomic action appears atomic from a programer's perspective, as no other thread or process can see the intermediate steps of an action until the action completes and commits. The paper introduced atomic procedures (called actions), whose syntax is

```
<identifier> : action(<parameter-list>);
    <statement-list>
end
```

The body of an action is invoked and executes in a manner similar to a procedure. However, its execution is isolated with respect to concurrently executing threads (strong isolation).

Data shared among threads must be annotated with a `shared` attribute, both to reduce implementation overhead, by forgoing locking nonshared data, and to document a programmer's intent.

Lomet recognized the necessity for a coordination mechanism between transactions and proposed delaying an action's execution until a predicate is satisfied:

```
<identifier> : action(<parameter-list>);
   await <predicate> then
   <statement-list>
end
```

Lomet's implementation only evaluated the predicate when the value of one of the explicitly listed parameters changed, to avoid busy waiting. The predicate's evaluation is part of the action and its execution is atomic with respect to the action's body, which can therefore assume that the predicate holds when it starts execution.

As an example, consider the following simple bounded queue that can pass data between threads:

```
buffer:class shared;

   frame:array(0:N-1) of T;
   count:integer initial(0);
   head:integer initial(0);

   send:action(x:T);
      await(count < N-1) then
      begin;
         frame[(head + count) % N] := x;
         count := count + 1;
      end;
   end send;

   receive:action(y:T);
      await(count > 0) then
      begin;
         y := frame[head];
         head := (head + 1) % N;
```

```
        count := count - 1;
      end;
   end receive;
end buffer;
```

The notation is from Lomet's paper, because the constructs do not have ready analogues in C#. send atomically adds an item to the queue, when space becomes available. `receive` atomically removes an item from the queue.

Lomet noted that this queue (or, in fact, even a conventional queue) is not particularly useful to atomic actions, since the changes made by the inner transaction (the queue operations) are not visible to other threads until the outer transaction commits (open nested transactions had not yet been invented). In particular, an action cannot receive any more messages than the queue contained when the action first started execution, since no other thread's enqueues can affect the state of the queue.

The paper also observed that failure atomicity without isolation is by itself a useful programming abstraction, previously introduced by Randell as a recovery block [6, 7]. In this role, an action is a mechanism for error recovery, which restores a program's state to its condition on entry to the action. Lomet elaborated this idea by introducing a `reset` operation, which aborts the surrounding action and rolls back the program's state.

STM Characteristics	
LOMET	
Strong or Weak Isolation	Strong
Transaction Granularity	Unknown
Direct or Deferred Update	Direct
Concurrency Control	Pessimistic
Synchronization	Blocking
Conflict Detection	
Inconsistent Reads	
Conflict Resolution	
Nested Transaction	
Exceptions	

Implementation

Lomet's paper explicitly disclaimed providing an implementation, but it suggested using two-phase locking to synchronize access to shared data and logging to rollback on an explicit reset.

3.3.2   Shavit, Touitou, PODC 1995

Shavit and Touitou's 1995 PODC paper [8] coined the term "software transactional memory" and described the first software implementation of transactional memory. Their programming abstraction required a transaction to declare, in advance, all memory locations that it might access and the system incurred significant memory overhead (a word per word of data). Knowing the memory locations enabled the STM system to acquire ownership of them with the two-phase locking protocol [5]. Once a transaction acquires ownership of its memory locations, it can execute to completion without the possibility of rollback. Moreover, the STM system can acquire ownership of the locations in a predetermined order (e.g., increasing memory address), which prevents two transactions from deadlocking.

Shavit and Touitou's STM system offers the strong liveness guarantee of lock-free progress that, after a finite number of attempts, some thread running a transaction will succeed, even in the face of thread failure and thread scheduling. STM implements this guarantee with a technique called *helping*, in which a transaction that fails to obtain ownership of a location attempts to execute the transaction that holds ownership, under the assumption that the thread running this latter transaction has been descheduled. As we will see, helping introduces complexity into the implementation, as it exposes data structures private to a transaction to manipulation by two or more threads.

STM Characteristics	
STM	
Strong or Weak Isolation	N/A
Transaction Granularity	Word
Direct or Deferred Update	Direct
Concurrency Control	Pessimistic
Synchronization	Nonblocking (lock-free)
Conflict Detection	Early
Inconsistent Reads	None (exclusive read access)
Conflict Resolution	Helping
Nested Transaction	
Exceptions	

Implementation

A transaction is represented by a data structure, which records the data necessary to execute the body of the transaction and information on the transaction's status. The structure field `addr` holds the addresses of the memory locations accessed by the transaction. This list is sorted in increasing order, to avoid deadlock and ensure the correctness of this algorithm. The field `body` contains a delegate[1] that supplies the code that comprises the transaction. The delegate's argument is a vector containing the contents of these memory locations. The field `oldValues` records the contents of the memory locations when the transaction acquired ownership. The `version`, `status`, and `executing` fields track the state of the transaction:

```
enum Status { NONE, SUCCESS, FAILURE };
struct TransStatus { Status status; int word; };

delegate void TransactionBody(MemWord[]);

struct Transaction {

    // Parameters to the transaction:
    MemAddr[] addr;              // Sorted in increasing order
    TransactionBody body;

    // Active transaction status:
    MemWord[] oldValues;
    int version = 0;
    TransStatus status;
    bool executing = false;
}
```

Each memory word has a corresponding entry that records which transaction (if any) owns the memory location:

```
int MemorySize = <constant>;
MemWord Mem[MemorySize];
Transaction Ownership[MemorySize];
```

The `StartTransaction` routine executes a transaction whose code is encapsulated in the delegate body. The second parameter lists all memory addresses that the transaction could access. This routine first initializes a new transaction object and then invokes `DoTransaction`, which acquires ownership of the locations and executes the transaction:

[1]In C#, a delegate is a pointer to a method and the object it will be applied to. It is similar to a closure in a functional language.

```
Status StartTransaction(TransactionBody body, MemAddr[] dataSet) {
    Transaction trans = new Transaction(body, dataSet);
    trans.executing = true;
    DoTransaction(trans, trans.version, true);
    trans.executing = false;
    trans.version++;
    return tran.status.status;  // SUCCESS or FAILURE
}
```

Throughout the STM system, a common idiom is

```
LL(trans.field);
...
if (version != trans.version) return;
SC(trans.field, newval);
```

This code sequence loads and uses a value from a transaction object, then checks that the transaction's version is unchanged before updating a field in its structure. The version check ensures that the value came from the appropriate version of the transaction. The atomicity of the entire code sequence is ensured by the store-conditional operation, which fails if `field` is modified by another thread while running between the LL (load-linked) and SC (store condition) operations. The STM implementation issues multiple, nested LL and SC operations, which is not a facility provided by any processor.

The `DoTransaction` routine atomically executes a transaction. The first step is to acquire ownership of the locations accessed by the transaction. If the routine acquires ownership, it changes the transaction's status to SUCCESS, records old (pretransaction values), executes the body of the transaction, updates memory, and then releases ownership of the locations:

```
void  DoTransaction(Transaction trans, int version, bool isInitiator) {
    AcquireOwnership(tran, version);
    TransStatus stat = LL(tran.status);
    if (stat.status == NONE) {
        if (version != tran.version) return;
        SC(tran.status, new TransStat(SUCCESS, 0));
    }

    stat = LL(tran.status);
    if (stat.status == SUCCESS) {
```

```
        // Transaction acquired all locations.
        RecordOldValues(tran, version);
        MemWord[] newValues = body(tran.input);        // Execute transaction!
        UpdateMemory(tran, version, newValues);
        ReleaseOwnership(tran, version);
     }
  else {
       // Transaction aborted since it could not acquire all locations.
       // Help conflicting transaction finish first.
       ReleaseOwnership(tran, version);
       if (isInitiator) {
            Transaction conflictTrans = Ownership[conflictAddr];
            if (conflictTrans == null) return;
            else {
                 int conflictVersion = conflictTrans.version;
                 if (conflictTrans.executing) {
                 // Help the conflicting transaction complete.
                  DoTransaction(conflictTrans, conflictVersion, false);
            }
         }
      }
   }
  }
}
```

If this routine cannot acquire ownership of a location, because of a conflict with another transaction, this transaction helps the conflicting transaction to complete. Helping ensures that STM is wait-free by preventing a descheduled or terminated transaction from retaining ownership of memory locations, thereby preventing other transactions from making forward progress. Much of the complexity of nonblocking STMs is due to helping: since helping causes two threads to execute the same transaction, the Transaction data structure must be carefully manipulated to avoid races.

It is difficult in general to determine if a thread is making progress or is blocked (for one such technique, see [9]). STM does not attempt to determine if a conflicting transaction is stalled, even though it makes little sense to help an executing transaction. In fact, overly aggressive helping can degrade performance by causing unnecessary conflicts, as several threads contend for cache lines or object ownership, and thereby increase the number of transactions that

fail. To avoid these problems, STM introduced a key restriction: it only permitted a top-level (nonhelping) transaction to help another.

The following routines acquire and release ownership of the memory locations for the transaction. If the `AcquireOwnership` routine acquires all the transaction's memory locations, the transaction's status is set to (`NONE, 0`), otherwise its status is set to (`FAILURE, i`), where `i` is the index of the first (lowest) memory location owned by another transaction:

```
void AcquireOwnership(Transaction tran, int version) {
   for (int i = 1; i < tran.addr.size; i++) {
      while (true) {
         MemAddr location = tran.addr[i];
         TransStatus stat = LL(tran.status);
         if (stat.status != NONE) return;  // Transaction already finished
         Transaction owner = LL(Ownership[tran.addr[i]]);
         if (tran.version != version) return;
         if (owner == tran) break;  // Already own location
         if (owner == null) {
            // No other transaction owns the location, so claim it
             if (SC(tran.status, new TransStat(NONE, 0))) {
                if (SC(Ownership[location], tran)) break;
             }
          }
          else {
            // Location owned by another transaction, so this transaction fails
             if (SC(tran.status, new TransStat(FAILURE, i))) return;
          }
       }
    }
}
void ReleaseOwnership(Transaction tran, int version) {
   for (int i = 1; i < tran.addr.size; i++) {
      MemAddr location = tran.addr[i];
      if (LL(Ownership[location]) == tran) {
         if (tran.version != version) return;
         SC(Ownership[location], null);
      }
   }
}
```

This routine captures the value in all locations accessed by a transaction. It may only be invoked when the transaction owns these locations:

```
void RecordOldValues(Transaction tran, int version) {
   for (int i = 1; i < tran.addr.size; i++) {
      MemAddr location = tran.addr[i];
      MemWord oldvalue = LL(tran.oldValues[location]);
      if (tran.version != version) return;
      SC(tran.oldValues[location], Memory[location]);
   }
}
```

Finally, after the transaction completes, this routine updates memory with the new values computed by the transaction's body. This routine too may only be invoked when the transaction owns these locations:

```
void UpdateMemory(Transaction tran, int version, MemWord[] newvalues) {
   for (int i = 1; i < tran.addr.size; i++) {
      MemAddr location = tran.addr[i];
      MemWord oldvalue= LL(Memory[location]);
      if (tran.version != version) return;
      if (oldvalue != newvalues[i]) {
         SC(Memory[location], newvalues[i]);
      }
   }
}
```

3.3.3   Other Precursors

Other papers also explored techniques to implement lock-free data structures. Some work explored ways to implement multiword load-linked, store-conditional or compare-and-swap operations on conventional processors [10–13], to allow several memory locations to be updated simultaneously. These papers missed other important aspects of transactions such as conflict detection or abort. Other systems anticipated implementation techniques used in STM systems [14, 15], but only used the techniques to transform algorithms to be lock-free, rather than as the basis for the transactional system. These papers (and there are many others) provide the background for, and anticipate many aspects of, transactional memory systems.

3.4   DEFERRED-UPDATE STM SYSTEMS

The first STM systems were developed by researchers from the field of nonblocking data structures, so it is not surprising that these systems were implemented using these techniques. These STM systems used deferred update, which permitted every transaction to compute optimistically on its own copy of the data, without interference from concurrently executing transactions. When a transaction attempted to acquire another object or to commit, it might discover a conflict with another transaction and be forced to abort. Experience using these systems showed the importance of the conflict resolution policy that determined which transaction in a pair of conflicting transactions would pause or abort and which would continue.

Herlihy et al. described the first dynamic STM system (Section 3.4.1). Harris and Fraser described a roughly contemporaneous STM system along with programming language support (Section 3.4.2). Fraser's Ph.D. thesis described another STM system (Section 3.4.3). In a series of papers, researchers at the University of Rochester explored a variety of conflict resolution policies (Sections 3.4.4, 3.4.6, and 3.4.8). Guerraoui et al. also explored techniques for incorporating these policies into an STM system (Section 3.4.5). Ananian and Rinard built an STM system that provides strong isolation (Section 3.4.7). Dice and Shavit described an STM system called TL that combines deferred update with blocking synchronization (Section 3.4.9).

3.4.1   Herlihy, Luchangco, Moir, and Scherer, PODC 2003

Herlihy, Luchangco, Moir, and Scherer's PODC paper [16] was the first published paper to describe a dynamic STM (DSTM) system that did not require a programmer to specify a transaction's memory usage in advance. It improved on Shavit and Touitou's system (Section 3.3.2) in several other ways:

- DSTM used obstruction freedom as the nonblocking progress condition for transactions. This criterion, which is weaker than lock freedom or wait freedom, guarantees that a halted thread does not prevent active threads from making progress. In general, this criterion does not preclude an active thread from causing a livelock (Guerraoui et al. showed that the greedy contention management policy allowed stronger guarantees [17]). However, obstruction freedom is simpler to implement than the stronger guarantees and offers potentially higher performance.

- DSTM introduced an explicit contention manager, which encapsulated the policy decision as to how to resolve a conflict between two transactions.

- DSTM allowed a transaction to release an object before committing. This mechanism can improve STM performance, by reducing the size of a transaction's read set and the cost to validate it; albeit with the possibility of introducing errors that allow a transaction to commit after reading an inconsistent state.

DSTM is the canonical deferred-update STM system and forms the basis for a considerable amount of subsequent research and engineering. As such, we will examine its implementation in detail.

STM Characteristics	
DSTM	
Strong or Weak Isolation	Weak
Transaction Granularity	Object
Direct or Deferred Update	Deferred (cloned replacement)
Concurrency Control	Optimistic
Synchronization	Nonblocking (obstruction free)
Conflict Detection	Early
Inconsistent Reads	Validation
Conflict Resolution	Explicit contention manager
Nested Transaction	Flattened
Exceptions	

Implementation

DSTM was implemented as a library usable from C++ and Java. A programmer must explicitly invoke library functions to create a transaction and to access shared objects. Transactions run on threads of a new class. The programmer must introduce and properly manipulate a container for each object involved in a transaction. For example, the method below inserts an integer in an ordered list. It uses a transaction to isolate multiple updates to a list. The DSTM library calls (explained below) are underlined in this code:

```
public bool insert(int v) {
   List newList = new List(v);
   TMObject newNode = new  TMObject(newList);
   TMThread thread = (TMThread)Thread.currentThread();
   while (true) {
      thread.beginTransaction();
      bool result = true;
      try {
```

```
            List prevList = (List)this.first. open(WRITE);
            List currList = (List)prevList.next. open(WRITE);
            while (eurrList.value < v) {
                prevList = currList;
                currList = (List)currList.next. open(WRITE);
            }
            if (currList.value == v) { result = false; }
            else {
                result = true;
                newList.next = prevList.next;
                prevList.next = newNode;
            }
        } catch (Denied d) {}
        if (thread. commitTransaction()) {
          return result;
        }
    }
}
```

The DSTM library exports two abstractions. The TMThread class extends the Java Thread
class with operations to start, commit, and abort a transaction:

```
class TMThread : Thread {
    void beginTransaction();
    bool commitTransaction();
    void abortTransaction();
}
```

The TMObject class is a wrapper for transactional objects. These objects must be explicitly
opened before being read or written in a transaction:

```
class TMObject {
    TMObject(Object obj);
    enum Mode { READ, WRITE };
    Object open(Mode mode);
}
```

Detailed Implementation

DSTM is a deferred-update STM. A transaction modifies a private copy of an object created by open(WRITE), which replaces the original object when the transaction commits. Conflicts are detected when a transaction first opens an object and finds that it is open for modification by another transaction or when the transaction validates its read set (on opening an object or committing the transaction).

DSTM adds two levels of indirection to an object (Fig. 3.1(a)). First, a transaction references an object through a TMObject, which provides a level of indirection that allows the commit operation to replace the object's Locator using an atomic read-modify write operation. The Locator is an immutable structure that points to a read-only version of the object and, if the object is opened for update, to a modifiable, cloned copy private to the transaction.

The open operation prepares a TMObject to be manipulated by a transaction and exposes the underlying object to the code in the transaction. The actions that open performs depend on whether an object is open for reading or writing.

Consider first opening an object for reading (mode parameter equal to READ). The open method records the TMObject and the underlying object in the transaction's read set, validates this set, and then returns the underlying object to the program (Fig. 3.1(b)).

Validating the transaction's consistency relies on the read set. The method Transaction::validate compares each object entry in a transaction's read set against the current version of the object (obtained by following the TMObject reference). If the objects differ, the transaction should abort since it potentially read an object subsequently updated by another transaction. (The DSTM implementation in the paper stored the read set in thread-local storage. For the sake of this presentation, we moved it to the transaction class, where it belongs.)

The current version of an object is found through the object's Locator. If the Locator does not contain a transaction, the object is opened read-only and the current version is the original object (oldVersion). If the Locator points to a transaction that has committed, the current version is the one modified by the transaction (newVersion). If the transaction aborted, the current version is the original object (oldVersion). If the transaction is active, it conflicts with the transaction trying to access the object. The contention manager must resolve the conflict by aborting or delaying one of the transactions.

At a conflict, the STM system has the freedom to choose which of the two conflicting transactions to terminate (or delay) and which to allow to continue. DSTM does not constrain which transaction must prevail at a conflict. Instead, it provides a general interface that allows a contention manager to implement a wide variety of policies. The only constraint on these managers is that they do not violate the system's obstruction-free policy, which informally requires that any active transaction that tries sufficiently many times must eventually receive permission to abort a conflicting transaction.
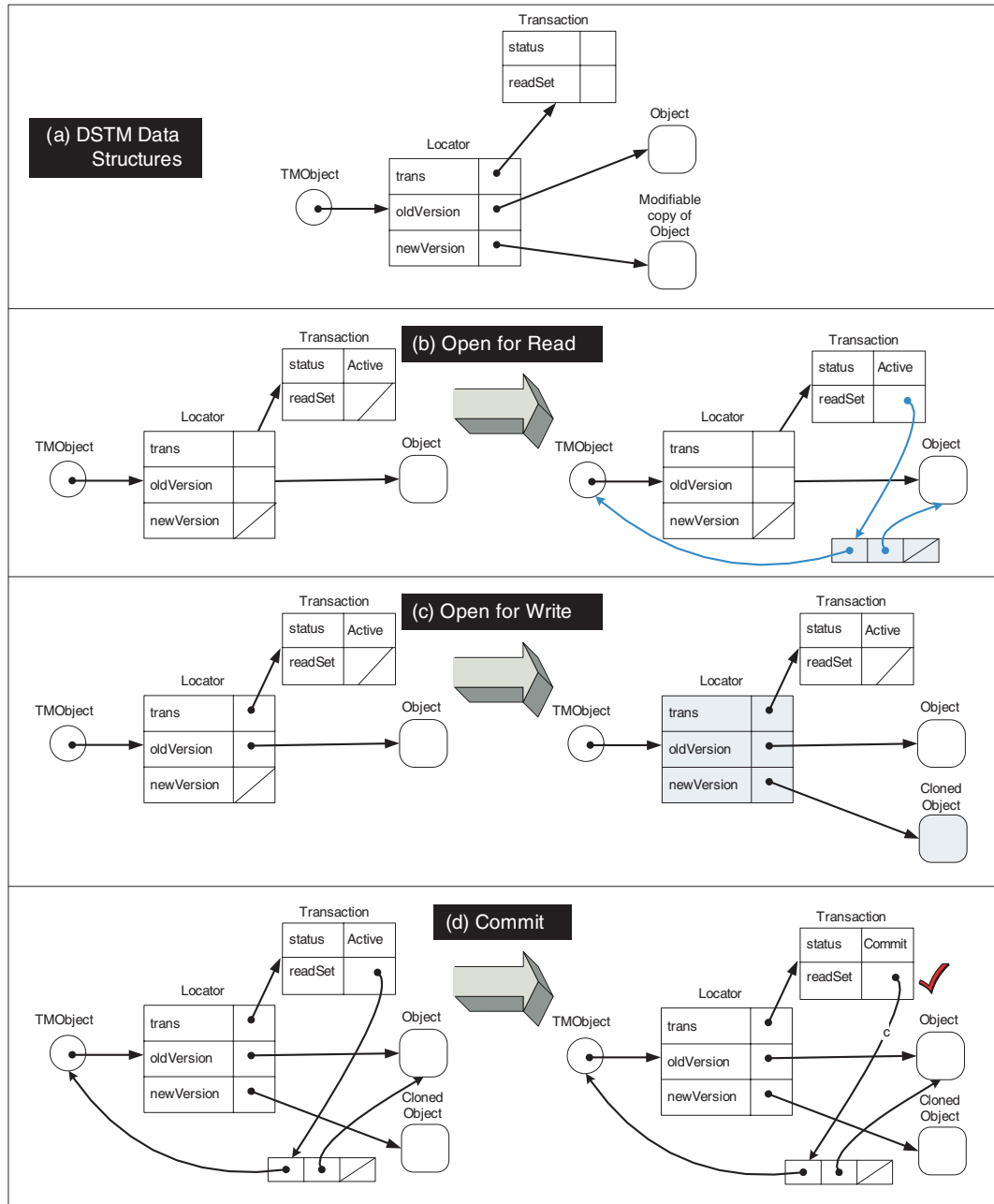
FIGURE 3.1: DSTM data structures and operations

When a transaction opens an object for writing (`mode` parameter equal to `WRITE`), the `open` method creates a cloned copy of the object that the transaction can modify (Fig. 3.1(c)). Until the transaction commits, this clone is visible only to the transaction, which can access it without synchronization. A transaction can upgrade from read-only to writable access by reopening an object for writing.

The `open` method ensures atomicity through the invariant that a `Locator` object is immutable (its fields do not change after they are initialized). Because of immutability, `open` can read multiple fields from this object by first copying the pointer from the `TMObject`, to ensure that the `Locator` will not change between accesses.

DSTM also provides a `release` operation (not shown) that removes an object from a transaction's read set because it will no longer be used. Releasing an object shrinks the transaction's read set, which lowers the cost of validating the set, and prevents the object from being the nexus of a conflict that causes a transaction to abort. This operation is potentially dangerous, since an unvalidated object can result in inconsistent execution. Releasing an object depends on knowing application-specific knowledge that precludes such inconsistency:

```
class TMObject {
   private class Locator {
      public Transaction trans;
      public Object oldVersion;
      public Object newVersion;
   }

   private Locator locInfo;

   public TMObject(Object obj) {
      locInfo = new Locator();
      locInfo.trans = null;
      locInfo.oldVersion = obj;
      locInfo.newVersion = null;
   }

   public enum Mode { READ, WRITE };
   public Object open(Mode mode) {
      TMThread curTMThread = (TMThread)Thread.currentThread();
      Transaction curTrans = curTMThread.getTransaction();
      if (mode == READ) {
```

```
      // Record the TMObject and its current value (version) in transaction's read table.
      curTrans.recordRead(this, currentVersion(locInfo));
      if (!curTrans.validate()) { throw new Denied(); }
      return version;
  }
  else {  // mode == WRITE
      // Create a new Locator pointing to a local copy of the object and install it.
      Locator newLocInfo = new Locator();
      newLocInfo.trans = curTras;
      do {
          Locator oldLocInfo = locInfo;
          newLocInfo.oldVersion = currentVersion(oldLocInfo);
          newLocInfo.newVersion = newLocInfo.oldVersion.clone();
      }  while (CAS(locInfo, oldLocInfo, newLocInfo) != oldLocInfo);
      if (!trans.validate()) { throw new Denied(); }
      return newLocInfo.newVersion;
  }
}

static private Object currentVersion(Locator loc) {
    if (loc.trans == null) {  // Read-only
      return locInfo.oldVersion;
  }
  else {
      switch (loc.trans.status) {
        case COMMITTED: return locInfo.newVersion; break;
        case ABORTED: return locInfo.oldVersion; break;
        case ACTIVE:  _/* conflict resolution */ break;
      }
    }
  }
}
```

Now, consider the implementation of the transactional thread class. The TMThread class creates the Transaction object for the transaction running on the thread (Fig. 3.1(d)). The methods in the class allocate, validate, and update the transaction's status, but most of the

work is done elsewhere:

```
class TMThread : Thread {
   private Transaction trans;
   Transaction getTransaction() { return trans; }

   public void beginTransaction() {
      trans = new Transaction();
      trans.status = ACTIVE;
   }

   public bool commitTransaction() {
      return trans.validate() && CAS(trans.status, ACTIVE, COMMITTED);
   }

   public void abortTransaction() {
      trans.status = ABORTED;
   }
}
```

The `Transaction` class records the status of a transaction and tracks its read set, so the transaction can be validated by verifying that these locations have not been updated since they were opened by the transaction. A transaction commits by validating its read set, and if that operation succeeds, by changing its status from `ACTIVE` to `COMMITTED`. This modification makes all of the transaction's modified objects into the current version of the respective objects:

```
class Transaction {
  public enum Status { ACTIVE, ABORTED, COMMITTED };
  public Status status;

  private class readPair {
    TMObject tmObj;
    Object obj;
    readPair(TMObject tmObj, Object obj)
        { this.tmObj = tmObj; this.obj = obj; }
  }
```

```
private HashSet readSet;
void recordRead(TMObject tmObj, Object value) {
  readSet.add(new readPair(tmObj, value));
}

bool validate() {
  for (Iterator e = readSet.iterator(); e.hashNext();) {
    readPair pair = (readPair)e.next();
    if ((pair.tmObj.locInfo.trans == NULL)
          && pair.tmObj.locInfo.oldValue != pair.obj)
      { return false; }
    if (!(pair.tmObj.locInfo.trans.status == COMMITTED
          && pair.tmObj.locInfo.newValue == pair.obj))
       { return false; }
  }
  return this.status == ACTIVE;
}
}
```

Kumar et al. describe hardware support to integrate DSTM with an HTM to eliminate some DSTM overheads (Section 4.6.2).

3.4.2   Harris and Fraser, OOPSLA 2003

Harris and Fraser's 2003 OOPSLA paper [18] was the first to describe a practical STM system integrated into a programming language. They implemented WSTM (word-granularity STM) in the ResearchVM from Sun Labs. It contained a number of advances:

- Like Herlihy et al.'s DSTM system (Section 3.4.1), but unlike Shavit and Touitou's original STM system (Section 3.3.2), WSTM did not require a programmer to declare the memory locations accessed within a transaction.

- Like Herlihy et al.'s DSTM system, WSTM used obstruction freedom as the forward-progress guarantee.

- WSTM is one of the few word-granularity STM systems. Although implemented in an object-oriented language (Java), WSTM detected conflicts at word granularity, which eliminated the need to change objects' layout, but did complicate the implementation.

- WSTM adopted the guard mechanism from Hoare's conditional critical regions [19] (anticipated in Lomet's transaction programming proposal (Section 3.3.1)) as a mechanism to coordinate atomic regions. The guard delays the execution of an atomic region until a predicate evaluates to true. Since the guard's predicate evaluates within the atomic region, its condition holds when the region itself executes. WSTM allowed the guards to depend on arbitrary program state, rather than explicit parameters, as in Lomet.

- WSTM improved on earlier mechanisms for implementing conditional critical regions by delaying reevaluation of the predicate until another transaction updates at least one of the variables in its expression.

- WSTM was integrated into a modern, object-oriented language (Java) by extending the language with the `atomic` operation. Strangely enough, WSTM did not exploit the object-oriented nature of Java and could support procedural languages as well.

STM Characteristics	
WSTM	
Strong or Weak Isolation	Weak
Transaction Granularity	Word
Direct or Deferred Update	Deferred (update in place)
Concurrency Control	Optimistic
Synchronization	Nonblocking (obstruction free)
Conflict Detection	Late
Inconsistent Reads	Inconsistency toleration
Conflict Resolution	Helping or aborting
Nested Transaction	Flattened
Exceptions	Terminate

Implementation

WSTM extended Java with a new statement:

```
atomic (<condition>) {
    <statements>;
}
```

A modified JIT (Just-In-Time, i.e., run-time) compiler translated this statement into

```
bool done = false;

while (!done) {
    STMStart();
    try {
        if (<condition>) {
            <statements>;
            done = STMCommit();
        } else {
            STMWait();
        }
    } catch (Exception t) {
        done = STMCommit();
        if (done) {
            throw t;
        }
    }
}
```

A notable aspect of this translation is that an exception within the atomic region's pred-icate or body causes the transaction to commit. Subsequent systems more typically treated an exception as an error that aborts a transaction. Harris and Fraser based this design choice on two considerations. First, aborting a transaction at an exception requires special handling for the ex-ception object, to avoid discarding it along with the rest of the failed transaction's state. Second, an exception thrown from within a nested transaction, but caught in its surrounding trans-action, will cause both transactions to abort with flattened nested transactions. A subsequent paper (Section 3.6.1) discussed and expanded these design choices.

The code produced by the compiler relies on five primitive operations provided by the WSTM library:

```
void STMStart()
void STMAbort()
bool STMCommit()
bool STMValidate()
void STMWait()
```

In addition, all references to object fields from statements within an atomic region are replaced by calls to an appropriate library operation:

```
STMWord STMRead(Addr a)
void STMWrite(Addr a, STMWord w)
```

The semantics of data differs for code executing in an atomic region. The current ("logical") value of a memory location manipulated by a transaction is found in an auxiliary data structure, rather than the memory. This duality enables a transaction to execute up to its commit point without modifying program state visible to other threads. The transaction then commits its changes to the global state in a single logically atomic operation.

Because the JVM's JIT compiler performs this translation, only references from Java bytecodes, not those in native methods, are translated to access these auxiliary structures. A few native methods were hand translated and included in a WSTM library, but a call on most native methods (including those that perform IO operations) from a transaction would cause a run-time error. Moreover, this translation only occurs for references within an atomic region.

The STM library maintains two auxiliary data structures, in addition to the application's original heap (Fig. 3.2(a)). The first auxiliary structure is a transaction descriptor. Each transaction has a unique transaction descriptor:

```
enum TransactionStatus { ACTIVE, COMMITTED, ABORTED, ASLEEP };

class TransactionEntry {
   public Addr loc;
   public STMWord oldValue;
   public STMWord oldVersion;
   public STMWord newValue;
   public STMWord newVersion;
}

class TransactionDescriptor {
   public TransactionStatus status = ACTIVE;
   int nestingDepth = 0;
   public Set<TransactionEntry> entries;
}
```

The `status` field records the transaction status. A transaction starts in state `ACTIVE` and makes a transition into one of the three other states. The `nestingDepth` field records the number of (flattened) transactions sharing this descriptor.
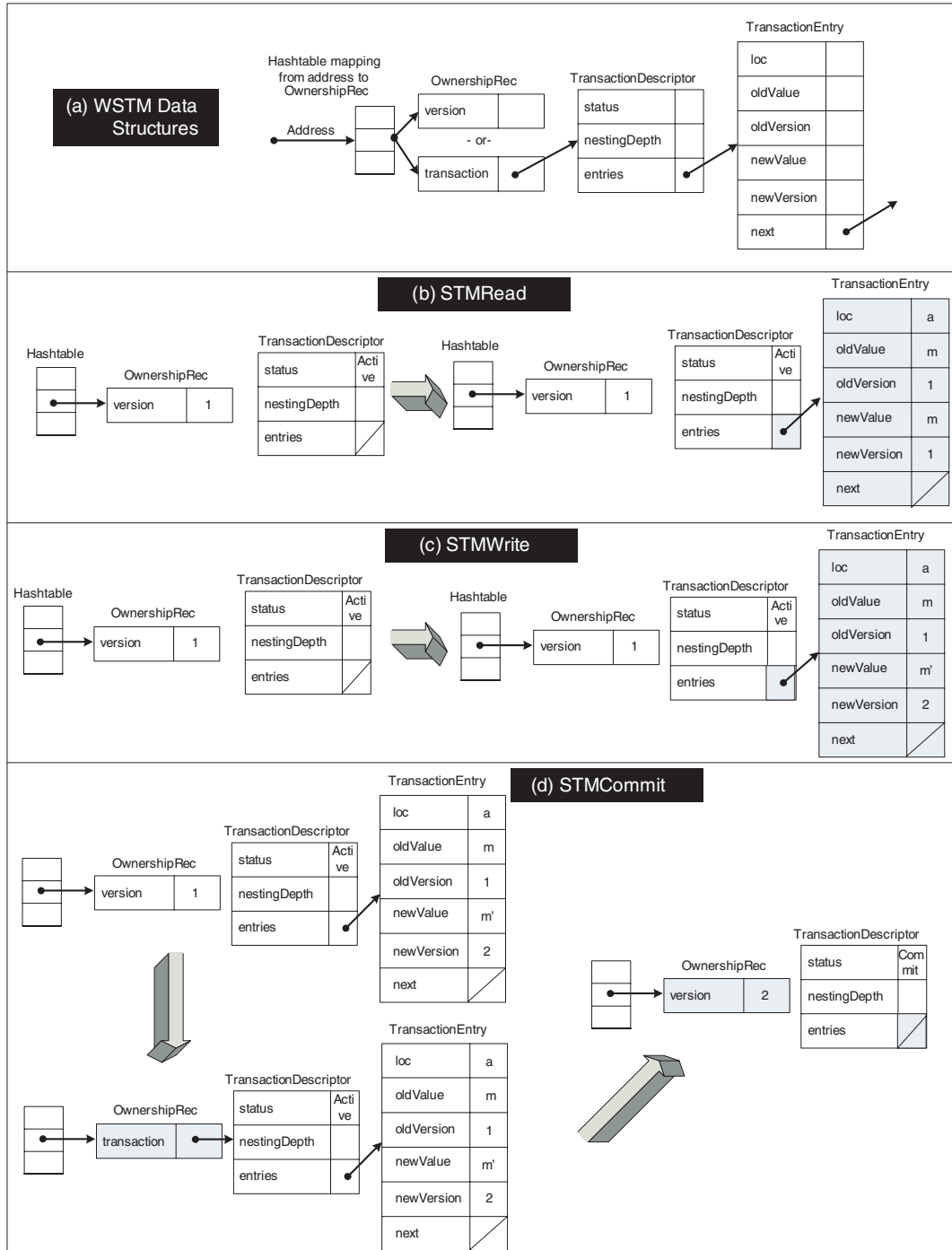
FIGURE 3.2: WSTM data structures and operations

The `entries` field holds a `TransactionEntry` record for each location the transaction reads or writes. The compiler redirects memory reads and writes to the appropriate descriptor. A `TransactionEntry` records a location's original value (before the transaction's first access) and its current value. For a location only read, the two values are identical. A transaction increments the version number when it modifies the location. The system uses the version number to detect conflicts.

The other auxiliary structure is a collection of ownership records for memory locations. These records serve two roles. First, an `OwnershipRec` records the version number of the memory location, produced by the most recent committed transaction that updated the location. Second, when a transaction is in the process of committing, an `OwnershipRec` records the transaction that has acquired exclusive ownership of the location. Each ownership record holds either a version number or a pointer to the transaction that owns the location:

```
class OwnershipRec {
   union {
      public STMWord version;
      public TransactionDescriptor* trans;
   } val;

   public bool HoldsVersion() { return (val.version & 0x1) != 0; }
}
```

A type union is not valid C# (but is valid for C, in which WSTM is written). Their low bit can distinguish the two values. `TransactionDescriptors` are word-aligned pointers with a zero low bit. Version numbers are odd numbers, with a nonzero low bit.

In practice, WSTM uses a fixed-size hash table to map from a memory address to an `OwnershipRec`. Consequently, WSTM shares an `OwnershipRec` among the multiple addresses that map to a hash-table bucket, which complicates the algorithms and causes spurious conflicts. This book avoids the complexity introduced by this sharing by assuming a one-to-one mapping from an address to its ownership record.

The function `FindOwnershipRec(a)` maps memory address `a` to its associated `OwnershipRec`. The function `CASOwnershipRec(a, old, new)` performs a compare-and-swap operation on the `OwnershipRec` for memory address `a`, replacing it with value `new`, if the existing entry is equal to `old`.

It is valuable to contrast the WSTM and DSTM (Section 3.4.1) implementations. A clear functional distinction is that WSTM is not tied to an object-oriented language, as it maintains transaction state at word, not object, granularity and does not assume the ability to clone an object. WSTM is suitable for languages, such as C or C++, whose weak type system

and unchecked pointers make it difficult to identify the boundary of a C struct or copy it. The McRT system (Section 3.5.2) addresses these issues by segregating C structs by size, so that the run-time system can find a struct's boundary, even from an interior pointer, and make a copy of it.

The principal disadvantage of WSTM's implementation is its high cost, both in memory overhead and computation. Each word of memory accessed in a transaction has an overhead of up to seven words, plus some fraction of the hash table. On the other hand, DSTM clones an entire object, so the overhead can be higher if a transaction accesses only a fraction of a large object, for example, an array.

In WSTM, each memory reference in a transaction entails a hash-table lookup and two additional dereferences. DSTM, by contrast, requires two dereferences to first open an object. Subsequently, the code in an atomic block directly references the cloned object.

Detailed Implementation

We can now describe the operation in the STM library. If no transaction is active, the STMStart operation creates a new transaction descriptor and stores it in thread-local storage. If a transaction is already active, the nested transaction shares its descriptor, so that the two transactions will commit or abort together (i.e., the transactions are flattened):

```
LocalDataStoreSlot ActiveTrans;  // Thread local storage
ActiveTrans = Thread.GetNamedDataSlot("Active Transaction");

void STMStart() {
   if (ActiveTrans == null
       || ActiveTrans.status != TransactionStatus.ACTIVE) {
       ActiveTrans = new TransactionDescriptor();
       ActiveTrans.status = TransactionStatus.ACTIVE;
   }
   AtomicAdd(ActiveTrans.nestingDepth, 1);
}
```

The STMAbort operation changes the active transaction's status. The descriptor and all of its records can be reclaimed immediately, since they are not visible to another thread:

```
void STMAbort() {
   ActiveTrans.status = TransactionStatus.ABORTED;
   ActiveTrans.entries = null;
   AtomicAdd(ActiveTrans.nestingDepth, -1);
}
```

The STMRead operation returns the transaction's current value for a location. There are two cases to consider. The first is that the transaction descriptor already contains a TransactionEntry (te) for the location. In this case, the location's value is found in the newVersion field. If no entry exists, the method creates an entry for the location and initializes it with the value in memory (Fig. 3.2(b)):

```
STMWord STMRead(Addr a) {
    TransactionEntry* te = ActiveTrans.entries.Find(a);
        if (null == te) {
        // No entry in transaction descriptor. Create new entry (get value from memory)
        // and add it to descriptor.
        ValVersion vv = MemRead(a);
        te = new TransactionEntry(a, vv.val, vv.version, vv.val,
            vv.version);
        ActiveTrans.entries.Add(a, te);
        return vv.val;
    }
    else {
        // Entry already exists in descriptor, so return its (possibly updated) value.
        return te.newValue;
    }
}
```

The function MemRead returns the value of a memory location, along with its version number. This information resides in different places, depending on whether another transaction that accessed the location is in the process of committing or has already committed:

- If no other transaction accessed the location and started committing, then the current value resides in the memory location and its ownership record contains the version number.

- If another transaction accessed the location and committed, the value is in the transaction's newValue field and the version in the newVersion field.

- If another transaction accessed the location and has started, but not finished committing, the value is stored in the transaction's oldValue field and the version in its oldVersion field:

```
struct ValVersion {
    public STMWord val;
```

```
    public STMWord version;
}


ValVersion MemRead(Addr a) {
    OwnershipRec orec = FindOwnershipRec(a);
    if (orec.HoldsVersion())
        {  // Location not owned by another transaction.
            STMWord version;
            STMWord val;
            do {
                    version = orec.val.version;
                    val = (*((STMWord*)a));
                    // Recheck after reading value to avoid inconsistency caused by a race.
            } while (version != orec.val.version);
            return new ValVersion (val, version);
        }
    else {  // Location owned by another transaction that has/is committing.
        TransactionDescriptor* td = orec.val.trans;
        TransactionEntry* te = td.entries.Find(a);
        if (td.status == TransactionStatus.COMMITTED)
            { return new ValVersion (te.newValue, te.newVersion); }
        else
            { return new ValVersion (te.oldValue, te.oldVersion); }
        }
}
```

STMWrite first ensures that a transaction record exists for the location (most simply by calling STMRead) and then updates the record's `newValue` and `newVersion` fields (Fig. 3.2(c)):

```
void STMWrite(Addr a, STMWord w) {
    STMRead(a);  // Create entry if necessary.
    TransactionEntry te = ActiveTrans.entries.Find(a);
    te.newValue = w;
    te.newVersion += 2;  // Version numbers are odd numbers.
}
```

The commit operation (Fig. 3.2(d)), STMCommit, first acquires the ownership records for all locations accessed by the transaction. If successful, STMCommit changes the transaction's state

to COMMITTED, copies the modified values to memory, and releases the ownership records. These three steps appear logically atomic to concurrent transactions because the committing transaction's status changes atomically (and irrevocably) from ACTIVE to COMMITTED using an atomic read-modify-write operation. Once this change occurs, MemRead will return the updated value, even before the transaction copies value back to memory.

WSTM detects conflicts as part of the commit operation, when it attempts to acquire a location. Unlike DSTM, WSTM does not repeatedly validate a transaction's read set, so code in the transaction may execute with inconsistent state. If the inconsistent state causes an exception, the atomic statement aborts and reexecutes the transaction. The compiler, however, inserts calls on STMValidate (below) in all loops, to ensure that the transaction does not loop endlessly because of an inconsistency.

STMCommit acquires ownership of a memory location by performing a compare-and-swap to replace the version number in the location's ownership record with a pointer to the transaction entry. The compare-and-swap operation can fail because the ownership record does not contain the expected version number or because it points to another transaction's descriptor. If the version number differs, then another (committed) transaction modified the location, so the committing transaction must abort. If the ownership record points to another transaction, then the two transactions are both in the process of committing and a decision must be made as to which one should be allowed to commit.

The STMCommit operation described in this book is blocking, since a thread must wait until another thread releases a location's ownership record before acquiring it. This implementation can be made nonblocking, at the cost of considerable complexity. To ensure that actual implementation is delay-free, WSTM allows a transaction to steal an OwnershipRec from another transaction, which may have stalled during the process of committing. Similar to the helping operation in other systems (Section 3.3.2), this process is too complicated to describe and to implement correctly. We will avoid presenting details, as other systems achieve this end with considerably less complexity.

STMCommit releases the acquired ownership records by replacing the transaction's descriptor with the appropriate version number (the old one if the transaction failed and a new one if it succeeded):

```
void STMCommit() {
    // Only outermost nested transaction can commit.
    if (AtomicAdd(ActiveTrans.nestingDepth, -1) != 0) { return; }

    // A nested transaction already aborted this transaction.
    if (ActiveTrans.status == TransactionStatus.ABORTED) { return; }
```

```
    // Acquire ownership of all locations accessed by transaction.
    int i;
    for (i = 0; i < ActiveTrans.entries.Size(); i++) {
       TransactionEntry* te = ActiveTrans.entries[i];
       switch (acquire(te)) {
          case TRUE: { continue; }
          case FALSE: {
                ActiveTrans.status = TransactionStatus.ABORTED;
                goto releaseAndReturn;
          }
          case BUSY: {  /* conflict resolution */ }
       }
    }


    // Transaction commits.
    ActiveTrans.status = TransactionStatus.COMMITTED;

     //Copy modified values to memory.
    for (i = 0; i < ActiveTrans.entries.Size(); i++) {
       TransactionEntry te = ActiveTrans.entries[i];
       *((STMWord*)te.loc) = te.newValue;
    }

releaseAndReturn:  // Release the ownership records.
      for (int j = 0; j < i; j++) { release(te); }
}

bool acquire(TransactionEntry* te) {
      OwnershipRec orec = CASOwnershipRec(te.loc, te.oldVersion,
                          ActiveTrans);
      if (orec.HoldsVersion())
         { return orec.val.version == te.oldVersion; }
      else {
         if (orec.val.trans == ActiveTrans) { return true; }
         else { return BUSY; }
      }
}
```

```
void release(TransactionEntry* te) {
    if (ActiveTrans.status == TransactionStatus.COMMITTED) {
        CASOwnershipRec(te.loc, ActiveTrans, te.newVersion);
    } else {
        CASOwnershipRec(te.loc, ActiveTrans, te.oldVersion);
    }
}
```

Nested transactions slightly complicate committing a transaction since only the outermost transaction can make the updated values visible, but any inner transaction can cause the surrounding transactions to abort. Each transaction descriptor tracks the number of active, nested transactions using the descriptor and defers committing the transactions until the final, outermost transaction commits. If an inner transaction previously aborted, the commit does not occur. A more sophisticated implementation could unwind nested transactions on an abort, rather than let them continue executing.

STMValidate is a read-only operation that checks the ownership records for each location accessed by the current transaction, to ensure that they are still consistent with the version the transaction initially read:

```
bool STMValidate() {
    for (int i = 0; i < ActiveTrans.entries.Size(); i++) {
        TransactionEntry* te = ActiveTrans.entries[i];
        OwnershipRec orec = FindOwnershipRec(te.loc);
        if (orec.val.version != te.oldVersion) { return false; }
    }
    return true;
}
```

STMWait can be used to implement a conditional critical region by suspending the transaction until its predicate should be reevaluated. It aborts the current transaction and waits until another transaction modifies a location accessed by the first transaction. It acquires ownership of the TransactionEntry accessed by the transaction, changes the transactions status to ASLEEP, and suspends the thread running the transaction. When another transaction updates one of these locations, it will conflict with the suspended transaction. The conflict manager should allow the active transaction to complete execution and then resume the suspended transaction, which

releases its ownership records and then retries the transaction:

```
void STMWait() {
    int I;
    for (i = 0; i < ActiveTrans.entries.Size(); i++) {
        TransactionEntry* te = ActiveTrans.entries[i];
        switch (acquire(te)) {
           case TRUE: { continue; }
           case FALSE: {
                 ActiveTrans.status = TransactionStatus.ABORTED;
                 goto releaseAndReturn;
           }
           case BUSY: { /* conflict resolution */ }
        }
  }
  // Transaction waits, unless in conflict with another transaction and
  // needs to immediately re-execute.
  ActiveTrans.status = TransactionStatus.ASLEEP;
  SuspendThread();
  // Release the ownership records.
releaseAndReturn:
      for (int j = 0; j < i; j++) { release(te); }
}
```

In the design above, if two transactions share a location that neither one modifies, one transaction will be aborted, since the system does not distinguish read-only locations from modified locations. This performance issue is easily corrected. STMWrite can set a flag (isModified) in a transaction entry to record a modification of the location. STMCommit should acquire ownership of modified locations and validate unmodified locations, to ensure that no other transaction updated them. This latter step introduces a new transaction status (READ_PHASE) before validation. The transaction remains in this state until it commits. The read phase introduces a window of vulnerability, during which another transaction can read a location and be unaware of the final state of the transaction. The value read is consistent, but the read can cause the transaction to fail [20]:

```
void STMCommit() {
   for (int i = 0; i < ActiveTrans.entries.Size(); i++) {
```

```
      TransactionEntry* te = ActiveTrans.entries[i];
      if (te.isModified) {
         switch (acquire(te)) {
         case TRUE: { continue; }
         case FALSE: {
            ActiveTrans.status = TansactionStatus.ABORTED;
            goto releaseAndReturn;
         }
         case BUSY: {  /* conflict resolution */ }
      }
   }
}


ActiveTrans.status = TransactionStatus.READ_PHASE;
for (int i = 0; i < ActiveTrans.entries.Size(); i++) {
   TransactionEntry* te = ActiveTrans.entries[i];
   if (!te.isModified) {
       ValVersion vv = MemRead(te.loc);
       if (te.oldVersion != vv.version) {
           // Another transaction updated this location.
           ActiveTrans.status = TransactionStatus.ABORTED;
           goto releaseAndReturn;
       }
     }
   }

   // Transaction commits. Write modified values to memory.
   ActiveTrans.status = TransactionStatus.COMMITTED;
   for (int i = 0; i < ActiveTrans.entries.Size(); i++) {
      TransactionEntry te = ActiveTrans.entries[i];
      *((STMWord*)te.loc) = te.newValue;
   }

   // Release the ownership records.
releaseAndReturn:
      for (int j = 0; j < i; j++) { release(te); }
}
```

3.4.3   Fraser, Ph.D.

Fraser's Ph.D. dissertation at the University of Cambridge [20] described an STM called OSTM, which was similar to Herlihy et al.'s DSTM system (Section 3.4.1), except that OSTM is lock free. It was implemented as a library for C code.

STM Characteristics	
OSTM	
Strong or Weak Isolation	Weak
Transaction Granularity	Object
Direct or Deferred Update	Deferred (cloned replacement)
Concurrency Control	Optimistic
Synchronization	Nonblocking (lock free)
Conflict Detection	Late
Inconsistent Reads	Validation
Conflict Resolution	Abort transaction trying to commit
Nested Transaction	Closed
Exceptions	

Implementation

Fig. 3.3 illustrates the data structures used by OSTM. An object header points to an object (actually a struct, since the language is C), which provides a level of indirection to the data manipulated by a program. This indirection is essential to lock and update objects when committing a transaction. Distinct from an object header is an object handle, which a transaction descriptor holds on either its list of readers or writers. A handle records the object's header, its data, and, if it is being updated, a shadow copy of the data for the transaction to modify.

When a transaction attempts to commit, it acquires ownership of all objects on its write list in a canonical order and validates all objects on its read list. OSTM acquires ownership by performing a CAS operation on an object's header, to replace the pointer to the object by a pointer to the transaction descriptor. If its CAS fails because another transaction updated the object header to point to an updated copy of the object, the transaction attempting to commit will abort. However, if its CAS fails because another transaction is in the process of committing
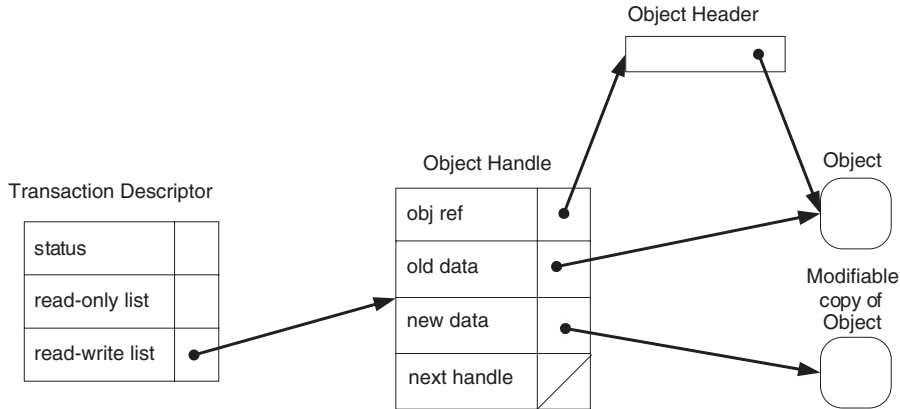
FIGURE 3.3: OSTM data structure

(i.e., its descriptor is in the header), then the first transaction recursively helps this transaction to ensure the system's lock-free behavior (see discussion of helping in Section 3.3.2).

3.4.4   Scherer and Scott, PODC 05

Scherer and Scott investigated contention management policies for the DSTM system [21, 22]. Contention management becomes necessary when a transaction attempts to open an object for reading and finds that another transaction previously opened the object for writing or when a transaction attempting to write the object finds another transaction reading or writing it. The contention manager selects which of the two transactions to abort. Alternatively, the manager can delay the acquiring transaction to allow the other transaction additional time that might enable it to completion.

Policies

A contention manager can implement a wide variety of policies, which vary considerably in complexity and sophistication:

- *Polite.* The acquiring transaction uses exponential backoff to delay for a fixed number of exponentially growing intervals before aborting the other transaction. After each interval, the transaction checks if the other transaction has finished with the object.

- *Karma.* This manager uses a count of the number of objects that a transaction has opened (cumulative, across all of its aborts and reexecutions) as a priority. An acquiring transaction immediately aborts another transaction with lower priority. If the acquiring

transaction's priority is lower, it backs off and tries to reacquire the object N times, where N is the difference in priorities, before aborting the other transaction.

- *Eruption.* This manager is similar to Karma, except that it adds the blocked transaction's priority to the active transaction's priority; to help reduce the possibility that a third transaction will subsequently abort the active transaction.

- *Kindergarten.* This manager maintains a "hit list" of transactions to which a given transaction previously deferred. If the transaction holding the object is on the list, the acquiring transaction immediately terminates it. If it is not on the list, the acquiring transaction adds it to the list and then backs off for a fixed interval before aborting itself. This policy ensures that two transactions sharing an object take turns aborting (hence the name).

- *Timestamp.* This manager aborts any transaction that started execution after the acquiring transaction.

- *Published Timestamp.* This manager follows the timestamp policy, but also aborts older transactions that appear inactive.

- *Polka.* This is a combination of the Polite and Karma policies. The key change to Karma is to use exponential backoff for the N intervals.

The goal of a contention manager is to produce good system throughput. The paper measured these policies for a variety of simple, data structure manipulating benchmarks. No policy performed best for all benchmarks, though Polka appeared to be the best overall.

The paper also evaluated the alternative of invalidating a transaction that opened an object for reading when another transaction opens the object for writing to avoid inconsistent reads. Each object maintains a list of transactions that are reading it (visible reads). When a transaction opens the object for update, it invalidates these transactions. Visible reads greatly reduced the validation overhead, but the bookkeeping cost of maintaining the reader list was sometimes large enough to negate these performance improvements. Subsequent work confirmed the high cost of visible reads for conventional clustered shared-memory multiprocessors (Section 3.4.8). This conclusion may not apply to multicore processors, which have lower communication costs.

3.4.5   Guerraoui, Herlihy, and Pochon, DISC 05

SXM [23] is a deferred, object-based STM system implemented as a library for C# code. It is similar in operation to Herlihy et al.'s DSTM system (Section 3.4.1). Its innovation is to permit a transaction to select a contention manager from a collection of managers that implement a diverse set of policies (see Scherer and Scott [Section 3.4.4] for a description of some policies). The system also provides a mechanism to adjudicate conflicting transactions' different policies.

In addition, SXM used run-time code generation to produce some of the boilerplate code that programmers need to write in other library-based STM systems.

STM Characteristics	
SXM	
Strong or Weak Isolation	Weak
Transaction Granularity	Object
Direct or Deferred Update	Deferred (cloned replacement)
Concurrency Control	Optimistic
Synchronization	Nonblocking (obstruction free)
Conflict Detection	Early
Inconsistent Reads	Inconsistency toleration
Conflict Resolution	Explicit contention manager
Nested Transaction	Closed
Exceptions	

Implementation

SXM's deferred-update structures are similar to those of DSTM, although the implementation technology differs. SXM's major contribution is *polymorphic contention management*, which is a flexible, high-performance framework for managing conflicts between transactions. Each transaction selects a contention management policy. The system provides the mechanism to resolve the differences between the policies of two conflicting transactions.

A contention manager implements a specific policy, represented by a specific class. A transaction is bound to an instance of the class that implements the desired policy. The SXM system invokes methods in the manager object when the associated transaction starts, attempts to read or write an object, aborts, and commits. The system also queries the manager when the transaction accesses an object and encounters a conflict with another transaction. The manager can select the transaction to abort, or it can delay the first transaction's execution to allow it to retry the access. A manager may use the history of the transaction (or other information) to make this decision.

SXM provides conflict resolution policies similar to those in Section 3.4.4 [22]. Another level of resolution is necessary when conflicting transactions choose different policies. SXM's

goal is to apply the policy that results in the best system throughput. To this end, SXM classifies policies based on the cost of the state that they maintain:

RANK	POLICY CLASS	POLICY	STATE
1		Aggressive, Polite	–
2	Ad hoc	Greedy, Killblocked	Transaction start time
3	Local	Timestamp	Transaction start time, variable
4		Kindergarten	List of transactions
5	Historical	Karma, Polka, Eruption	List of objects

A higher–numbered policy is more expensive to compute than a lower–ranked one; but it is not necessarily more predictive of future behavior. SXM does not favor higher–ranked policies.

Instead, the ranking identifies policies that are comparable to each other. SXM assumes that two transactions with policies from different policy classes were not intended to conflict, so neither transaction's policy is preferable. Instead, SXM uses the Greedy policy, which aborts the transaction that has executed the least amount of time. On the other hand, if the two transactions' policies belong in the same class, SXM applies the conflict policy from the transaction that discovered the conflict (i.e., wants to acquire the object).

3.4.6   Marathe, Scherer, and Scott, DISC 05

Adaptive STM (ASTM) is a deferred, object-based STM system that explored several performance improvements to the basic design of Herlihy et al.'s DSTM system (Section 3.4.1):

- ASTM eliminated a memory indirection in accessing fields in an object not open for update, thereby reducing the cost of reading objects. When a transaction opens an object for modification, it modifies the object's data representation.

- ASTM uses an adaptive (run-time) system to change its conflict resolution policy from early to late detection for transactions with a small number of memory writes and a large number of reads.

Implementation

ASTM follows the design and implementation of Herlihy et al.'s DSTM system (Section 3.4.1) in many respects. An object opened for update has the same representation, with two levels of indirection between the TMObject and the object's fields. In ASTM, however, an object opened

STM Characteristics	
ASTM	
Strong or Weak Isolation	Weak
Transaction Granularity	Object
Direct or Deferred Update	Deferred (cloned replacement)
Concurrency Control	Optimistic
Synchronization	Nonblocking (obstruction free)
Conflict Detection	Early or late (selectable)
Inconsistent Reads	Validation
Conflict Resolution	Explicit contention manager
Nested Transaction	
Exceptions	

for reading by a transaction has a single level of indirection, so the `TMObject` points directly to the object's fields (Fig. 3.4(a)). This representation eliminates a pointer dereference when reading the object's fields.

The reduction in indirection was inspired by OSTM (Section 3.4.3). Other systems, such as RSTM (Section 3.4.8), further reduce indirection. Direct-update systems (Section 3.5) eliminate this overhead entirely.

Naively implemented, this approach would allocate and deallocate `Locator` objects when a transaction opens an object for modification and subsequently when the transaction commits or aborts. ASTM reduces this overhead by deferring the deallocation until another transaction subsequently opens the object for reading. This heuristic eliminates unnecessary changes in representation that would have occurred when the object is repeatedly written.

ASTM implements either early or late conflict detection for objects opened for update. (DSTM only implemented early conflict detection.) With both policies, objects opened for reading are recorded in a private read list and are revalidated when every subsequent object is opened.

With late conflict detection, a transaction does not acquire ownership of an object that it is modifying. Instead, it modifies a cloned copy of the object and defers conflict detection until the transaction commits, which reduces the interval in which an object is locked and avoids some unnecessary transaction aborts. When the transaction commits, it may find that the
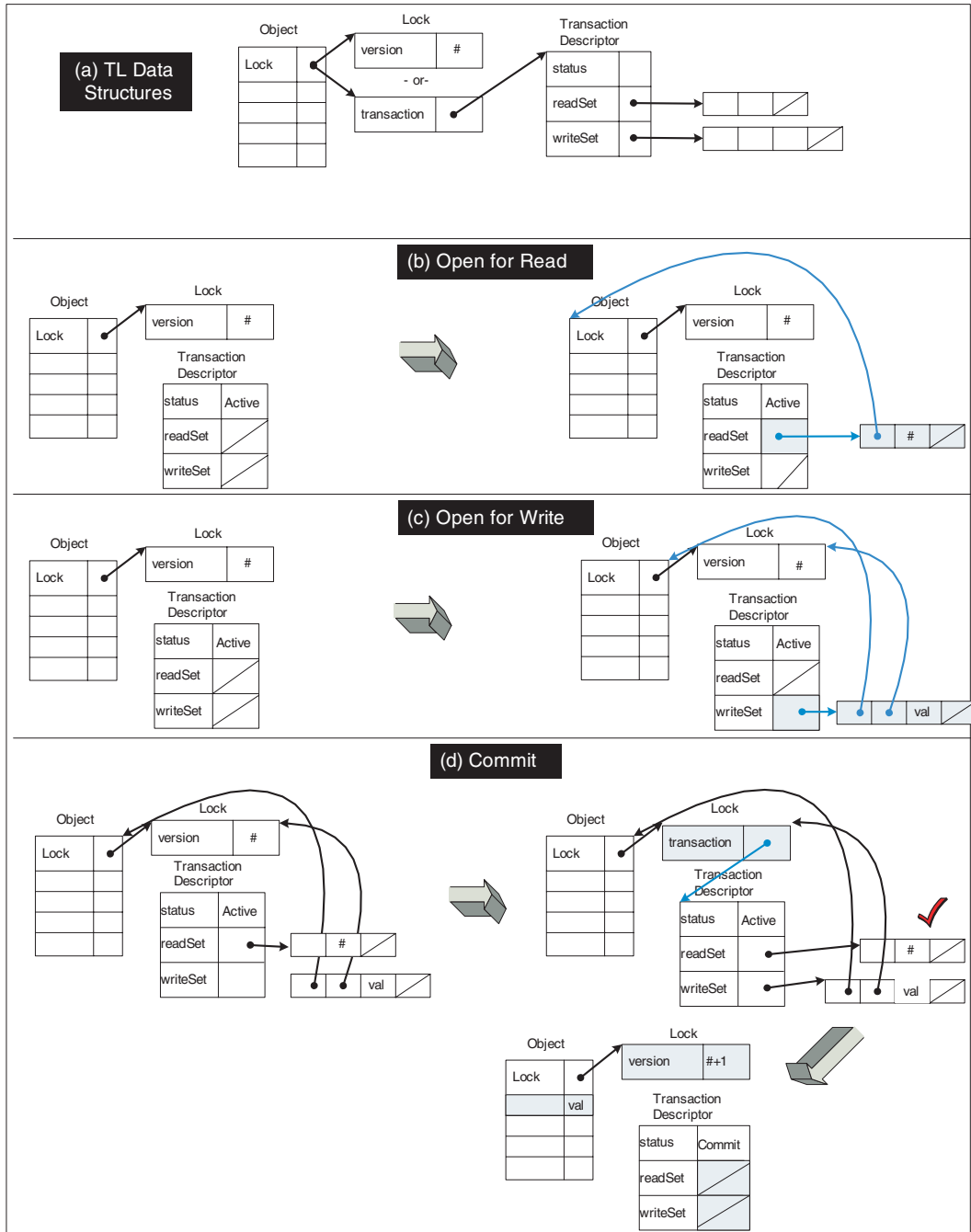
FIGURE 3.4: ASTM data structures and operations

object modified or may find another transaction in the process of committing changes to the object. The first case causes the transaction to abort, while the second one invokes a contention manager to resolve the conflict.

To avoid erroneous behavior in a doomed transaction, each transaction maintains a private write list and revalidates it every time an object is opened, to guard against the possibility that the transaction read the object before modifying it.

The paper observes that both policies have comparable overheads for most benchmarks, but early detection is simpler to implement. However, late acquire performs better for a long-running transaction that reads a large number of objects and updates only a few, because this policy allows concurrent readers and writers. In this situation, early acquire allows a transaction to hold an object open for a long interval, which causes conflicts with other transactions that read the object. Late conflict detection only aborts transactions actually reading an object when the writer transaction commits.

ASTM implements an adaptive policy that recognizes a transaction that modifies few objects (below a threshold) but reads many objects (above a threshold). ASTM then switches the thread executing the transaction from early to late detection for subsequent transactions. If a transaction falls below the thresholds, it reverts to early detection.

Simple benchmarks show that the optimized data representation improved performance by up to a factor of 3–4 on read-dominated benchmarks, as compared to DSTM. The adaptive algorithm provided little benefit for most benchmarks, but improved the performance of long-running, read-dominated transitions by a similar margin.

3.4.7   Ananian and Rinard, SCOOL 05

Ananian and Rinard describe a software transactional memory system with the distinguishing attribute of strong isolation [24]. In this system, a memory access outside of a transaction can abort a running transaction. Most other STM systems implement weak isolation, in which nontransactional reads and writes may access transactional data, but do not conflict with transactions, to avoid the overhead cost of adding instrumentation to every nontransactional memory read and write. Ananian and Rinard's system lowers the overhead of the instrumentation by placing a sentinel value in memory locations accessed by a transaction. The test for the sentinel is inexpensive enough to perform at every load and store in a program. This technique was previously used in the Shasta DSM system [25].

Their STM system is implemented in the FLEX Java system with object-granularity conflict detection, similar to Herlihy et al.'s DSTM system (Section 3.4.1). The system runs on a simulated computer architecture, which is used to study hybrid STM systems.

Another novel aspect of this system is that the STM algorithm is written in Promela, so it can be directly verified with the SPIN model checker [26]; an exercise that found several data races.

STM Characteristics	
ANANIAN AND RINARD	
Strong or Weak Isolation	Strong
Transaction Granularity	Object
Direct or Deferred Update	Deferred (in-place)
Concurrency Control	Optimistic
Synchronization	Nonblocking
Conflict Detection	Early
Inconsistent Reads	Invalidation
Conflict Resolution	Abort conflicting transaction
Nested Transaction	Flatten
Exceptions	Terminate or abort

Implementation

Each Java object is extended by two fields. The first, named `versions`, is a linked list containing a version record for each transaction that modified a field in the object. A version record identifies the transaction and records the updated value of each modified field. The second field, named `readers`, is a list of transactions that read a field in the object.

A novel aspect of this system is the use of a sentinel (signalling a conflicting access) value in a memory location to redirect read and write operations to the transactional data structures. Testing for a sentinel is a simple comparison, performed at every load and store in a program. The sentinel introduces complexity when a program manipulates the sentinel value itself, but careful choice of the sentinel (e.g., an invalid memory address and not a small, common immediate value) can minimize the program's legitimate use of the value.

When a *nontransactional* read encounters a sentinel, it aborts the transaction modifying the object containing the value (but not those just reading the object), restores the field's value from the most recently committed transaction's record, and re-reads the location (taking into account the possibility that the location actually holds the sentinel value). A *nontransactional* write aborts all transactions reading and writing the object and directly updates the field in the

object (overwriting the sentinel). If the program is actually writing the sentinel value, the write instruction is treated as a short transaction to ensure proper handling.

A *transactional* read first ensures that its transaction descriptor is on the object's reader list. It aborts all uncommitted transactions that modified the object. After this bookkeeping, the transaction can read the field in the object and directly use any values other than the sentinel. The sentinel value, on the other hand, requires a search of the version records to find one with the same version and the updated value of the field.

A *transactional* write aborts all other uncommitted transactions that read or wrote the object. It also creates, if none previously existed, a version object for the transaction. The next step is to copy the unmodified value in the field to all version records, including those of the committed transactions, so that the field can be restored if the transaction is rolled back. If the `versions` list does not contain a committed transaction, one must be created to hold this value. Finally, the new value can be written to the running transaction's version record and the object field set to the sentinel value.

This STM system aggressively resolves conflicting references to an object. A read aborts transactions in the process of modifying the object and a write aborts all transactions that accessed the object. In essence, the system implements a multireader, single-writer lock on an object.

The system requires an extended (multiword) version of load-linked–store-conditional, rather than mutual exclusion, which makes the low-level synchronization nonblocking. The simple conflict resolution policy (of aborting conflicting transactions) does not provide a guarantee of forward progress for a transaction, which can be repeatedly aborted while accessing heavily contended objects. On the other hand, this policy eliminates the need to validate transactional reads to avoid observing an inconsistent state.

Lie describes an integration of this STM with an HTM, to improve STM performance (Section 4.6.1).

3.4.8   Marathe, Spear, Heriot, Acharya, Eisenstat, Scherer, Scott, TRANSACT 06

RSTM is a nonblocking STM system implemented as a C++ class library [27]. It contains several enhancements intended to improve the performance of a deferred-update STM:

- RSTM only uses a single level of indirection to an object, instead of the two levels used by previous systems such as DSTM (Section 3.4.1) or ASTM (Section 3.4.6). This feature, however, requires modifying objects' layout to add two fields.
- RSTM avoids dynamically allocating many of its data structures and contains its own memory collector, so it can work with nongarbage collected languages such as C++.

- RSTM uses invalidation to avoid inconsistent reads. It employs a new heuristic for tracking an object's readers, which reduces the bookkeeping overhead found by previous systems of these sorts [22].

The paper strongly argues that STM should be implemented with nonblocking synchronization, since blocking synchronization remains subject to a host of problems, including priority inversion, thread failure, convoying, preemption, and page faults.

STM Characteristics	
RSTM	
Strong or Weak Isolation	Weak
Transaction Granularity	Object
Direct or Deferred Update	Deferred (cloned replacement)
Concurrency Control	Optimistic
Synchronization	Nonblocking (obstruction free)
Conflict Detection	Early or late (selectable)
Inconsistent Reads	Bounded invalidation
Conflict Resolution	Conflict manager
Nested Transaction	Flatten
Exceptions	

Implementation

In RSTM, every transactional object is accessed through an `ObjectHeader`, which points directly to the current version of the object (Fig. 3.5). RSTM uses the low-order bit of the `NewData` field in this object as a flag. If the bit is zero, then no transaction has the object open for writing and the field points directly to the current version of the object. If the flag is 1, then a transaction has the object open.

The `TransactionDescriptor` referenced through an object's header determines the transaction's state. If the transaction commits, then `NewDataObject` is the current version of the object. If the transaction aborts, then `OldDataObject` is the current version. If the transaction is active, no other transaction can read or write the object without aborting the transaction.
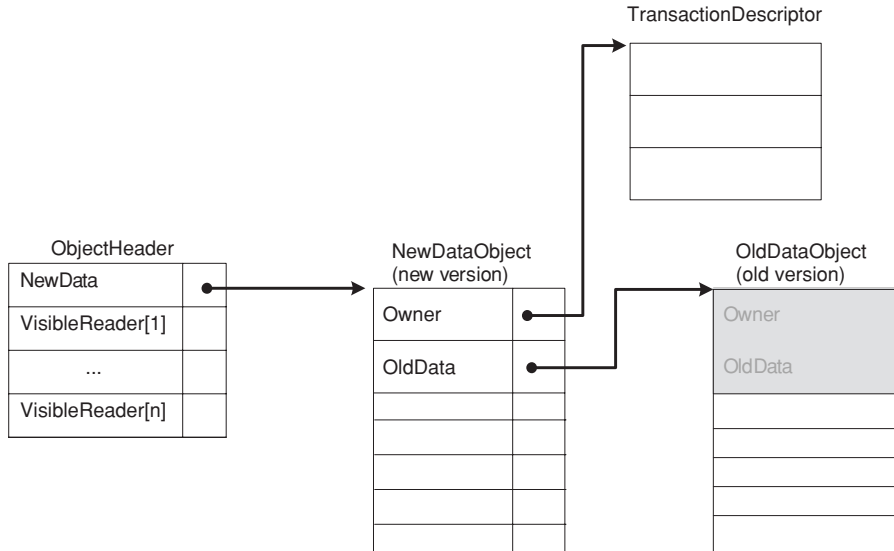
FIGURE 3.5: RSTM data structures

A transaction opens an object before accessing it:

1. If opening the object for update, the transaction must first acquire the object with the following actions:
    (a) Read the object's `NewData` pointer and make sure no other transaction owns it. If it is owned, invoke the contention manager.
    (b) Allocate the `NewDataObject` and copy values from the object's current version.
    (c) Initialize the `Owner` and `OldData` pointers in the new object.
    (d) Use a CAS to atomically swap the pointer read in step (a) with a pointer to the newly allocated copy.
    (e) Add the object to transaction's private write list.
    (f) Iterate through the object's visible reader list, aborting all transactions it contains.
2. If opening the object for reading and space is available in the object's visible reader list, add the transaction to this list. If the list is full, add the object to the transaction's private read list.
3. Check the status word in the transaction's descriptor, to make sure another transaction has not aborted it.
4. Incrementally validate all objects on the transaction's private read list.

One of RSTM's new contributions is a visible reader list (`VisibleReader[]`) which avoids the quadratic cost of validating a transaction's read list each time it opens an object. An `ObjectHeader` contains a fixed-size list of transactions that have the object open for reading. When a transaction acquires the object for update, it aborts these transactions. A transaction on an object's visible reader list does not need to validate reads, since a conflicting write will abort the transaction. However, if the list is full, the transaction must add the object to its private read list and validate it. Even so, visible readers can reduce the size of this list and the cost of validating it.

RSTM runs without garbage collection. It is parsimonious about memory allocation: a single transaction descriptor suffices for all transactions executed on a thread, and it does not dynamically allocate read or write lists (until they overflow). When a transaction completes, RSTM is able to reclaim its data structures.

Limited performance measurement of RSTM suggests that visible readers are actually more costly, because of the extra cache traffic caused by updating the visible read table in each object. In addition, the measurements found that late conflict detection performed better than early, because it permits more transactions to complete execution in situations with complex conflict relationships among transactions.

Shriraman et al. describe instruction set extensions and hardware extensions to accelerate the RSTM implementation (Section 4.6.3).

3.4.9   Dice and Shavit, TRANSACT 06

Dice and Shavit described an STM system called transactional locking (TL), which combined deferred update with blocking synchronization [28]. Unlike other lock-based STM systems—such as McRT-STM (Section 3.5.2) or BSTM (Section 3.5.4), which lock an object at first access—TL locks an object when a transaction commits. Deferred locking requires deferred update to hold uncommitted modifications. Using a lock simplifies the data structures required by other nonblocking STM systems, such as DSTM (Section 3.4.1), and improves performance. Limited experiments found that acquiring locks late, when a transaction commits, generally performed better than acquiring them early, at first access to an object. This performance improvement was particularly noticeable in benchmarks with heavy contention.

Implementation

The paper sketched and evaluated several variants of the TL algorithm. The preferred variant acquires locks when a transaction commits (Fig. 3.6). An alternative is to acquire locks at first access to an object. The implementation of the two is similar, but we will only present the former, which performs better. Similarly, we only discuss object-granularity locking, not the word- or region-granularity locking that is also feasible, though generally slower.

STM Characteristics	
TL	
Strong or Weak Isolation	Weak
Transaction Granularity	Object, word, or region
Direct or Deferred Update	Deferred (in-place)
Concurrency Control	Optimistic
Synchronization	Blocking
Conflict Detection	Early or late (selectable)
Inconsistent Reads	Inconsistency toleration
Conflict Resolution	Delay and abort
Nested Transaction	Flatten
Exceptions	

Each object is associated with a lock, similar to the one in McRT-STM (Section 3.5.2). A lock is either locked, and pointing to the transaction holding exclusive access, or unlocked, and recording the object's version, which is incremented when a transaction updates the object.

A transaction maintains a read set and a write set. An entry in the read set contains the address of the object and the version number of the lock associated with the object. An entry in the write set contains the address of the object, the address of its lock, and the updated value of the object.

When the transaction executes a write, it first looks for the object's entry in its write set. If it is not present, the transaction creates an entry for the object (Fig. 3.6(b)). The write modifies the entry in the write set, not the actual object. The transaction does not acquire the lock in the preferred variant of the algorithm.

A memory load first checks the write set (using a Bloom filter [29]), to determine if the transaction previously updated the object. If so, it uses the updated value from the write set. If the object is not in the set, the transaction adds the referenced object to the read set and attempts to read the actual object (Fig. 3.6(b)). If another transaction locked the object, the reading transaction can either delay and retry or abort itself.

When a transaction commits, it first acquires locks for all objects in its write set (Fig. 3.6(d)). A transaction will only wait a bounded amount of time for a lock to be released; otherwise, the system would deadlock if two transactions acquire locks in opposite orders. After acquiring locks for the objects it modified, the transaction validates its read set. If successful, the transaction can complete by copying updated values into the objects, releasing the locks, and freeing its data structures.
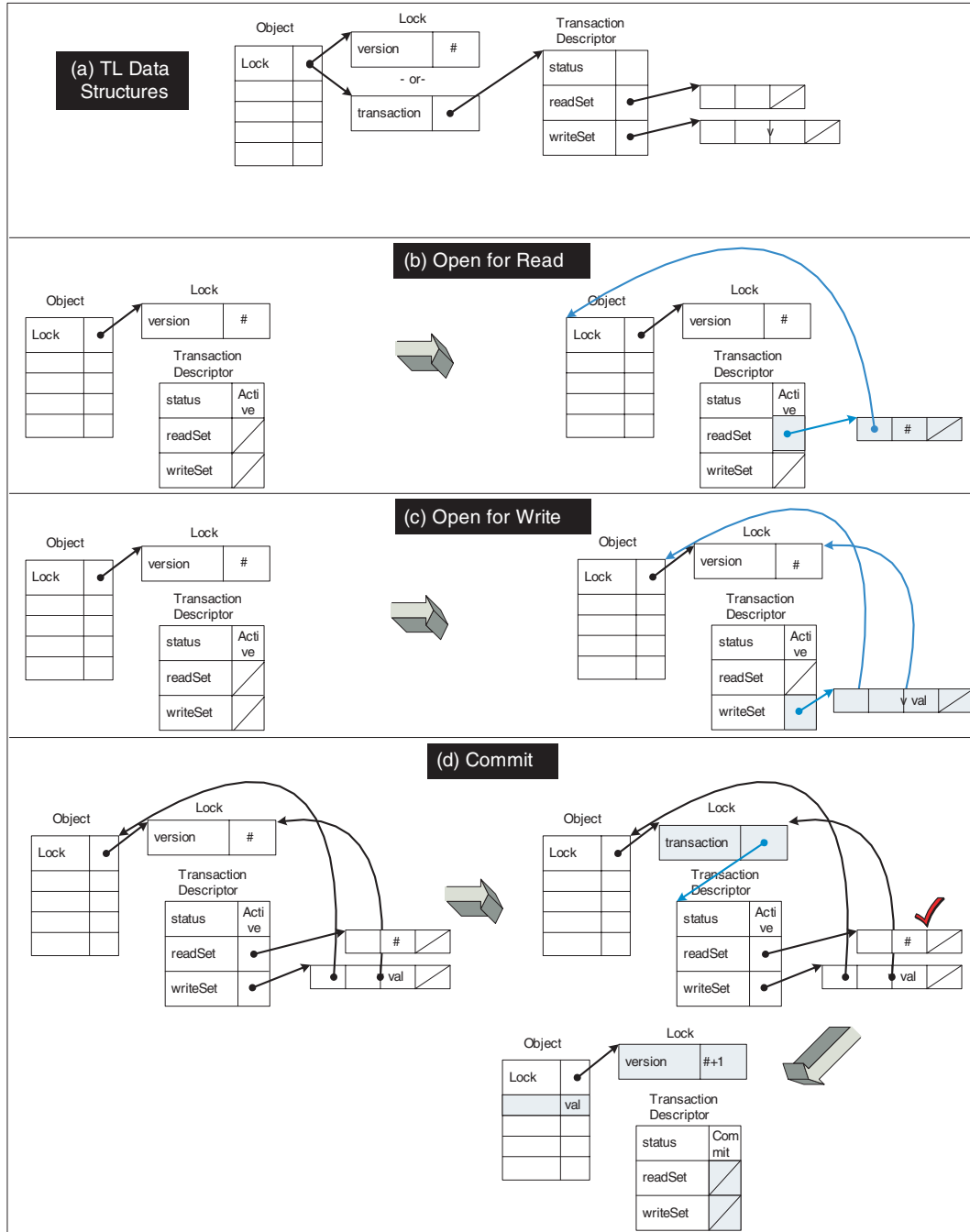
FIGURE 3.6: TL data structures and operations

Performance measurements on simple benchmarks showed that TL performed better than other STM systems, such as Harris and Fraser's nonblocking WSTM system (Section 3.4.2) and Ennals' STM system [30]. The measurements showed that acquiring locks at commit time performed as well or better than acquiring locks on first access to an object. The benefits of acquiring locks later increased with higher contention for a shared data structure. In addition, per-object locking had lower overhead than locking memory words or regions, even though the rate of conflicts was higher. Finally, measurements also showed that the cost of maintaining and validating read sets is large.

3.5   DIRECT-UPDATE STM SYSTEMS

Recently, researchers have started to explore alternative implementation strategies for STM systems. The most important of these is direct update, in which a transaction immediately modifies an object and uses explicit synchronization to prevent other transactions from reading or writing the modified object. Despite this synchronization, most of these systems require a mechanism to roll back a transaction's changes if it aborts. When a transaction encounters no conflicts, these systems can be very efficient, as they incur only the overhead of locking and logging.

Many of the direct-update systems use optimistic concurrency control. Manassiev et al. describe an STM system built for a cluster of computers running a distributed shared-memory system (Section 3.5.1). Adl-Tabatabai et al. investigate a variety of STM implementation techniques, including direct update (Sections 3.5.2 and 3.5.3). Harris et al. described a very similar direct-update STM system (Section 3.5.4).

However, other systems have investigated pessimistic concurrency control. McCloskey et al. described a TM-like system in which a user annotates data structures with the locks that protect them and the system uses this information to implement atomic regions (Section 3.5.5). Hicks et al. described a technique to infer locks and a consistent locking order necessary to implement transactions using pessimistic concurrency control (Section 3.5.5).

3.5.1   Manassiev, Mihailescu, Amza, PPoPP06

Distributed multiversioning (DMV) [31] differs fundamentally from the other systems in this survey. It does not implement transactional memory for threads executing in the address space of a single computer. Rather, it implements transactions on the Treadmarks software distributed shared-memory system (S-DSM) [32], which provides shared memory among a cluster of computers connected by a network.

Treadmarks creates a shared address space among processes running on distinct machines by trapping the first write to a page of memory with the processor's virtual memory hardware. Treadmarks records the initial contents of the page and enables subsequent writes to the page. At

synchronization points, the system sends other computers a list of modified pages. When another computer subsequently references one of these pages, it requests the page's change list and updates the replica of the page, which is mapped into its address space at the same virtual address. The system provides a consistent view of memory across the computers. In general, S-DSM systems implement a release-consistent memory model [33], in which updates are propagated at synchronization points and installed on other computers at synchronization points. A properly synchronized program sees a sequentially consistent view of memory across all computers.

This paper observes that the mechanisms of an S-DSM system can easily be adapted to implement transactional memory. The key differences are that transactions provide a mechanism to specify the execution interval over which memory updates are buffered on a machine before being propagated, and that transactions provide a different semantics for resolving conflicting memory references.

The DMV system tracks page modifications using the Treadmarks mechanisms. When a transaction commits, DMV sends the change list to other computers, which record, but do not apply the changes. After all computers acknowledge receipt of the change list, the transaction can commit. On a given computer, if the change list updates a page being modified by an active transaction, the transaction is aborted. If the page is not being modified by a transaction, the changes are applied on demand, when the page is subsequently referenced, to avoid delaying the committing transaction. DMV also allows a read-only transaction to continue executing with an outdated version of a page, even after receiving an update, to reduce the number of rollbacks on updates.

STM Characteristics	
DMV	
Strong or Weak Isolation	Strong
Transaction Granularity	Word
Direct or Deferred Update	Direct
Concurrency Control	Optimistic
Synchronization	None (distributed memory)
Conflict Detection	Late
Inconsistent Reads	None (multiple version)
Conflict Resolution	Abort uncommitted transaction
Nested Transaction	
Exceptions	

3.5.2   Saha, Adl-Tabatabai, Hudson, Minh, Hertzberg, PPoPP06

A group at Intel built an STM system for Java and C/C++ code called McRT-STM, after its underlying concurrent run-time system named McRT [34, 35]. McRT-STM has three distinguishing features.

- Transactions directly update memory locations, rather than buffering updates until commit time. Before modifying a location for the first time, a transaction must record the location's value, to be able to restore if the transaction aborts. Direct updating reduces the cost of reading a location and of committing a transaction, as compared to buffered updates. It increases the cost of aborting a transaction, but this operation is inherently expensive, since it discards computation, and should not occur frequently.

- McRT-STM uses two-phase locking rather than a nonblocking design. Locking simplified the STM implementation and yielded higher performance by reducing the frequency of transaction aborts relative to a nonblocking implementation.

- The user-perceptible benefits of a nonblocking STM are maintained by linking the STM system to the run-time system's thread scheduler, so a thread will wait to acquire a lock held by an executing transaction, and by using timeouts and aborts to detect and resolve deadlocks.

- McRT-STM implements transactions for C/C++ code, as well as Java. It supports both object- and cache-line-granularity locking for both safe and unsafe code.

STM Characteristics	
McRT-STM	
Strong or Weak Isolation	Weak
Transaction Granularity	Word or object
Direct or Deferred Update	Direct
Concurrency Control	Optimistic read, Pessimistic write
Synchronization	Blocking
Conflict Detection	Early write–write conflict Late write–read conflict
Inconsistent Reads	Inconsistency toleration
Conflict Resolution	Abort
Nested Transaction	Closed
Exceptions	

The McRT STM systems implement a different semantic model than the single-lock atomicity described in Chapter 2. For example, it fails to ensure the serializeability of the list example discussed in Section 2.3.5. Other STM systems also used direct update and locking: Ennals' STM system [30], Hindman and Grossman's AtomicJava [36], and Microsoft's Bartok STM system (Section 3.5.4). Dice and Shavit's TL system used deferred update and locking (Section 3.4.9).

Implementation

The paper described and evaluated alternative implementations of several STM mechanisms for lock granularity, locking discipline, and deferred and direct update. We will describe in detail the preferred implementation and briefly note the alternatives. Section 3.5.4 presents a more detailed implementation of a similar system.

McRT-STM uses two-phase locking to control access to an object. This protocol allows multiple transactions to read an object, but only a single transaction to modify the object. After acquiring a read lock, a transaction records the object's version number in its read set, to permit subsequent validation. After acquiring a write lock, the transaction records the location's value and object's version number in a log, to allow rollback if the transaction aborts. Conflicting writes from other transactions are detected by verifying that the version numbers of all locations read by a transaction remain unchanged when the transaction commits. Deadlock is detected by timeouts on lock acquires. Aborting a transaction and rolling back its side effects resolves conflicts or deadlocks.

Each object contains a transaction lock, which can be in one of two states. If the object is unlocked or only being read, its lock contains the object's current version number. In this state, another transaction can open the object for reading by simply recording, in its read set, the object and its current version.

The other state indicates that the object is open for writing. In this case, the lock points to the transaction descriptor of the transaction modifying the object. When a transaction opens the object for writing, it records the object and its version number (from the lock) and replaces the version number in the lock with a pointer to its transaction descriptor. When the transaction commits and releases its lock, it increments the object's version number and replaces the transaction descriptor in the lock with the new version number. Upgrades—from reading to writing an object—require that the reader acquire a write lock and, at commit time, verify that the object's read version number matches the initial write version number. Active readers do not prevent a writer from acquiring a lock and modifying an object; the conflict is detected and resolved when a reader transaction attempts to commit.

When an object is open for writing, other transactions that attempt to open the object for reading or writing will block on its write lock. McRT-STM implements a policy of suspending

these contending transactions if the thread holding the lock is active and aborting the transaction holding a write lock if its thread is suspended. This policy approximates the obstruction-free behavior of other STM systems, by ensuring that a suspended transaction does not prevent active transactions from making forward progress.

When a transaction opens an object for writing and modifies its value, it does not abort transactions that previously read the object. Each of these transactions will detect the conflict when it checks the version numbers of the objects in its read set, as part of the commit process. Since transactions directly modify locations, any of these transactions may read values modified by another transaction and so observe inconsistent program state. This can cause abnormal executions, for example, unexpected exceptions or improper loop termination. The former problem can be handled by catching these exceptions, aborting the transaction, and subsequently retrying it. The latter problem requires the consistency of a transaction's read set to be validated on the backedge of every loop whose termination may be dependent on state potentially modified by another transaction.

McRT-STM also investigated a more complex locking protocol that uses read locks to prevent a transaction from reading inconsistent state. The scheme allowed multiple concurrent readers, but gave writers priority (including upgrades from reader to writer) to avoid the well-known problem that "writers starve"; as a steady stream of readers never allow the read count to drop to zero so a writer can acquire its exclusive lock. Read versioning was considerably faster than read locking (by up to an order of magnitude) for two reasons. First, read locking required atomic memory operations at every manipulation of the lock word. These operations are expensive in some processor implementations. Second, read locking increased the latency of the common operation of upgrading from reader to writer, since a transaction must wait for all other readers to complete their transaction and release their lock.

McRT-STM also compared two mechanisms that enable a program's state to be restored when a transaction aborts: deferred update and direct update and logging. Deferred update stores a modified value in a location distinct from the original object, for example in a clone of an object. Other transactions can read the original value from the original location, but the transaction modifying the value must both read and write the new, transaction-specific location. If the transaction aborts, the clone can be discarded. By contrast, direct update acquires exclusive access to an object and modifies it in place. The original value of the object must be captured in a log, so it can be restored if the transaction aborts.

Deferred update only requires that locks be held when a transaction is in the process of committing, not when it is computing the new values, which can reduce contention and the frequency of conflict-induced rollback. This is the approach taken in Dice and Shavit's TL system (Section 3.4.9). On the other hand, a transaction modifying a location must find and use the buffered value for subsequent reads, which increases the cost of reading these values.

McRT-STM found that logging performed better (by a factor of 2 to 6) than buffering because of the overhead of searching for the most recent value. This considerations may not apply to all systems since eager locking is easier to implement in a HTM system. Late locking requires a more complex synchronization protocol and increases the amount of wasted work when a transaction aborts, which may be an important consideration for a HW design.

The McRT-STM system can provide transactional memory for C/C++ code, as well as Java. A major complication is that programs written in C/C++ can manipulate "interior" pointers, which point into the middle of a structure or object. Transactions need a mechanism to map one of these pointers to its containing object, which contains the transactional lock. The McRT run-time system provides size-segregated heaps, each of which holds objects of a single size that is a power of 2. A pointer into an object can be easily converted into a pointer to its heap, by masking the low-order bits. A heap's header records the objects' size, which can be used as a mask to convert an interior pointer to the object's starting point in the heap.

In addition to object locking, McRT-STM also investigated memory region locking, similar to Harris and Fraser's WSTM system (Section 3.4.2) [18]. McRT-STM used the middle bits of a pointer as an index into a table of locks, which provided a fast, but not necessarily unique, mapping between a pointer and the lock on the cache line it referenced. Neither form of locking performed consistently better. Performance depended on object size and access patterns.

3.5.3   Adl-Tabatabai, Lewis, Menon, Murphy, Saha, Shpeisman, PLDI06

This paper further elaborates the McRT-STM system by providing more specific details about the system's implementation than the previous paper (Section 3.5.2) and by describing the compiler optimizations used to reduce the STM overhead [35]. The system's implementation is similar to Harris et al. (Section 3.5.4), which also appeared in PLDI 2006. We will defer a detailed description of the implementation of a direct-update STM system to the discussion of that paper and will focus on the McRT-STM compiler optimizations.

The StarJIT compiler used in the McRT-STM system optimizes STM-related code, both with conventional and STM-specific optimizations. A key issue in all compiler optimizations is to ensure that the program transformation preserves the program semantics. In the low-level representation seen by a compiler, the STM operations must be explicitly connected to a specific transaction so a compiler does not optimize across transaction boundaries; for example, by eliminating logging of a variable in all but the first atomic block in a series of transactions. In addition, disparate STM-related operations, such as opening and logging a memory reference, must be explicitly connected, so the compiler can remove all STM operations when it eliminates the reference.

McRT-STM translates transactions into an intermediate form that explicitly represents these program constraints, so existing compiler optimizations correctly transform STM code.

This system translates an atomic block into a series of statements that delimit a scope corresponding to the transaction. For example, the statements

```
atomic { obj.f1 = x1; };
atomic { obj.f2 = x2; obj.f2 = x2; }
```

are translated into

```
while (true) {
   TxnHandle th = txnStart();
   try { txnOpenObjectForWrite(th, obj);
         txnLogObjectRef(th, obj, f1_offset);
         obj.f1 = x1;
         break; }
   finally { if (!txnCommit(th)) continue; }
}
while (true) {
   TxnHandle th = txnStart();
   try { txnOpenObjectForWrite(th, obj);
      txnLogObjectRef(th, obj, f2_offset);
      obj.f2 = x2;

      txnOpenObjectForWrite(th, obj);
      txnLogObjectRef(th, obj, f2_offset);
      obj.f2 = x2;
      break; }
   finally { if (!txnCommit(th)) continue; }
}
```

Although their names are identical, the two transaction handles are distinct values held in distinct variables whose scope is delimited by the transaction (loop) bodies. Compiler optimizations will not try to eliminate the calls on `txnOpenObjectForWrite` in the second loop as redundant, since their first arguments differ from calls in the previous loop.

The compiler encodes the dependence between an object reference and its open and logging statements with pseudovariables (called "proof variables" [37]) that exist within the compiler representation, but not the generated code. For example, if redundancy elimination in the compiler deleted the first assignment to `obj.f2` in the second atomic statement, the compiler will also eliminate the associated calls on `txnOpenObjectForWrite` and `txnLogObjectRef` because of these proof variables.

The compiler also performs STM-specific optimizations, such as hoisting the load of a transaction descriptor from thread-local storage out of a loop, as a reference to thread-local storage is expensive. Similarly, it replaces an open for reading with an open for writing, if the latter operation dominates the former. In addition, the compiler aggressively in-lines STM operations. Finally, the compiler eliminates open and logging operations on references to immutable locations and local objects whose lifetime is contained within a transaction, since the former are read-only and the latter cannot be shared by other transactions.

The compiler optimizations produce a substantial improvement in the speed of executed code, in some cases reducing McRT-STM's overhead by an order of magnitude.

3.5.4   Harris, Plesko, Shinnar, Tarditi, PLDI06

This STM system from Microsoft is similar in many aspects to the McRT-STM from Intel (Sections 3.5.2 and 3.5.3) [38]. Microsoft's BSTM was implemented with the Bartok compiler and run-time system, an optimizing MSIL (Microsoft Intermediate Language) to x86 compiler developed at Microsoft Research. BSTM is a direct-update STM that uses two-phase locks to ensure exclusive access to objects being updated, version numbers to ensure read consistency, and compiler optimizations to eliminate unnecessary STM operations. BSTM also aggressively attempts to reduce the size of logs, both by filtering unnecessary entries and through tight integration with the system's garbage collector. This system is the successor to a previous system [39], which provided transactions without isolation as an error recovery mechanism. The Bartok STM systems implement a different semantic model than the single-lock atomicity described in Chapter 2. For example, it fails to ensure the serializeability of the list example discussed in Section 2.3.5.

Detailed Implementation

Conceptually, BSTM adds a field of type `STMWord` to each object (Fig. 3.7). This field contains two values. The first is a bit indicating if a transaction has the object open for updating. If this bit is set, the field's other value points to the transaction descriptor of the transaction that opened the object. If the bit is not set (the object is not open for update), the other field is the object's version number. The following atomic operations manipulate the field:

```
word GetSTMWord(Object o)
bool OpenSTMWord(Object o, word prev, word next)
void CloseSTMWord(Object o, word next)
```

BSTM also associates a hint, called a Snapshot, with each object. It changes value when a thread that opened the object for update calls `CloseSTMWord`. A Snapshot is an inexpensive indication that the object has been updated. BSTM stores the `STMWord` in a dynamically

STM Characteristics	
BSTM	
Strong or Weak Isolation	Weak
Transaction Granularity	Object
Direct or Deferred Update	Direct
Concurrency Control	Optimistic read, Pessimistic write
Synchronization	Blocking
Conflict Detection	Early write–write conflict Late write–read conflict
Inconsistent Reads	Inconsistency toleration
Conflict Resolution	Abort
Nested Transaction	Closed
Exceptions	Abort

allocated header word, so the Snapshot incurs no additional cost, as the address of this header changes when the word is updated. For simplicity, the code below does not use Snapshots, and so only illustrates the slow, but general path in case this hint fails.

A transaction is represented by a `TMMgr` structure. It is manipulated by the usual set of operations:

```
TMMgr TMGetTMMgr()
void TMStart(TMMgr tx)
void TMAbort(TMMgr tx)
bool TMCommit(TMMgr tx)
bool TMIsValid(TMMgr tx)
```

A transaction contains three append-only logs. The read and update logs record the objects that the transaction reads or modifies. When opening an object for reading, the system records the object and its version number in the read log (Fig. 3.7(b)):

```
void TMOpenForRead(TMMgr tx, object obj) {
   tx.readLog.obj = obj;
   tx.readLog.stmWord = GetSTMWord(obj);
   tx.readLog ++;
}
```
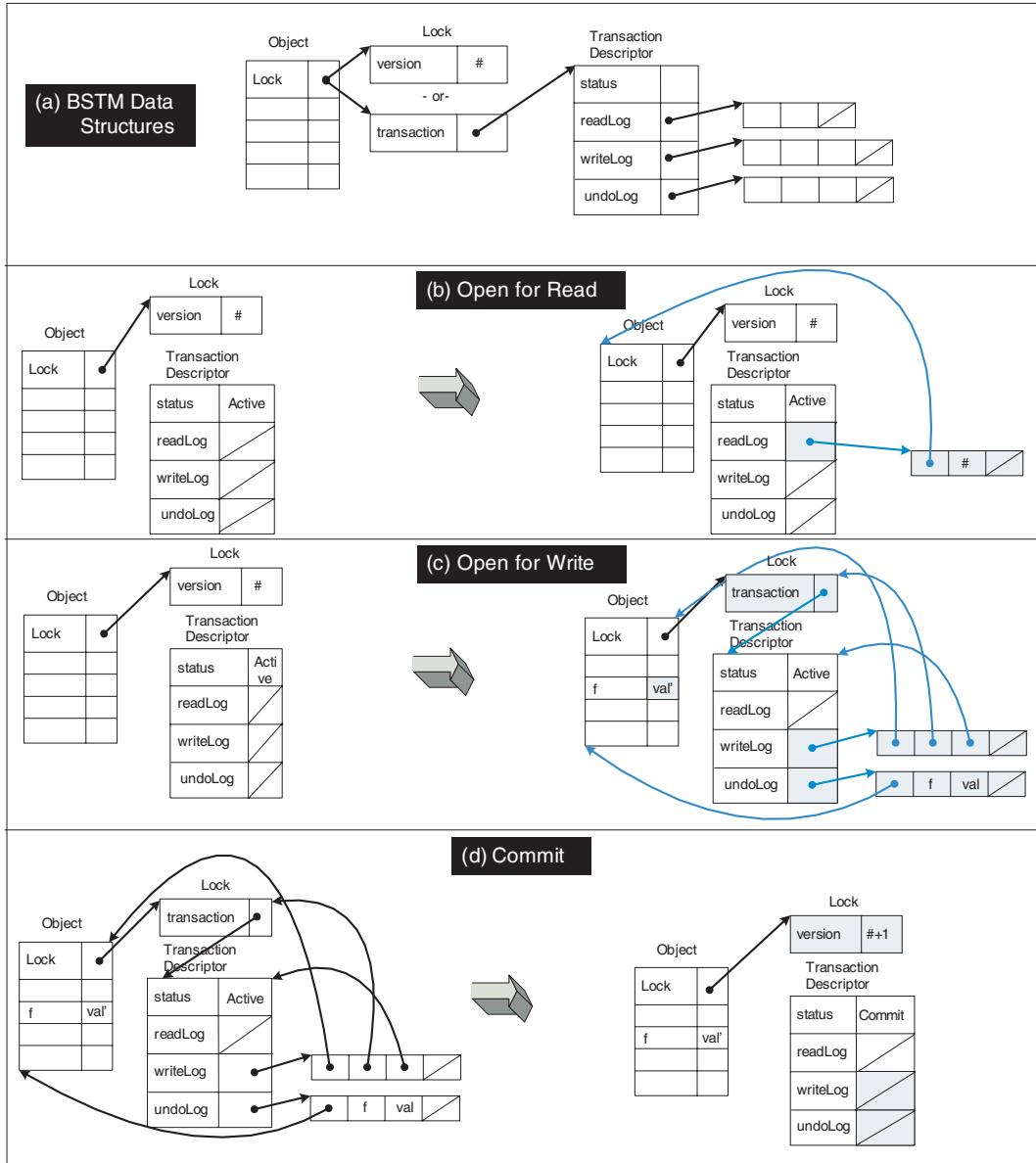
FIGURE 3.7: BSTM data structures and operations

Note that this operation does not check if the object is open for modification. When the reading transaction validates its read set, it detects a conflict when an object is locked for writing or its version number is updated. Since a transaction must always validate its read set to detect conflicts that arise after it opened a location for reading, checking for a conflict as part of the

`TMOpenForRead` operation would slow down the common case and only provide benefit if conflicts are frequent.

Opening an object for updating is more complex, as the operation acquires an exclusive lock on the object that prevents other transactions from updating it (Fig. 3.7(c)). To acquire this lock, a transaction must ensure that no other transactions either have the object open for update or are concurrently opening the object for update:

```
void TMOpenForUpdate(TMMgr tx, object obj) {
   word stmWord = GetSTMWord(obj);
   if (!IsOwnedSTMWord(stmWord)) {
      // Object is not owned by any transaction
      tx.updateLog.obj = obj;
      tx.updateLog.stmWord = stmWord;
      tx.updateLog.tx = tx;

      word newSTMWord = MakeOwnedSTMWord(tx);
      if (OpenSTMWord(obj, stmWord, newSTMWord)) {
         // Open succeeded: advance our log pointer
         tx.updateLog ++;
      } else {
         // Open failed: make the transaction invalid (and/or invoke contention manager)
         BecomeInvalid(tx);
      }
   } else if (GetOwnerFromSTMWord(stmWord) == tx) {
         // The object is already open for update by the current transaction: nothing more to do
   } else {
         // The object is already open for update by another transaction: abort our transaction
          BecomeInvalid(tx);
   }
}
```

After opening an object for reading, a program can directly manipulate it. However, before updating a field in an object, the system must record its value in the transaction's undo log, so the modification can be rolled back:

```
void TMLogFieldStore(TMMgr tx, object obj, int offset) {
   tx.undoLog.obj = obj;
   tx.undoLog.offset = offset;
```

```
   tx.undoLog.value = obj[offset];  // pseudo code
   tx.undoLog ++;
}
```

TMCommit commits a transaction in two phases (Fig. 3.7(d)). It first validates the consistency of the objects read by the transaction and then closes the objects opened for updating. Only the first phase can cause this operation to fail, since the transaction holds exclusive locks on the objects it opened for updating:

```
bool TMCommit(TMMgr tx) {
   foreach (ReadLogEntry e in tx.readLog) {
      if (!ValidateReadObject(tx, e.obj, e.stmWord)) { return false; }
   }
   foreach (UpdateLogEntry e in tx.updateLog) {
      CloseUpdatedObject(tx, e.obj, e.stmWord);
   }
   return true;
}
```

Validating an object ensures that (1) no other transaction had the object open for updating when this transaction first opened it for reading, (2) no other transaction opened the object for updating after this transaction opened it for reading, and (3) no other transaction has the object open for updating:

```
bool ValidateReadObject(TMMgr tx, object obj, STMWord oldSTMWord) {
  word curSTMWord = GetSTMWord(obj);
  if (!IsOwnedSTMWord(oldSTMWord)) {
      // Object originally was not opened by us for update
      if (oldSTMWord == curSTMWord) {
         // No intervening access by another transaction
      } else if (!IsOwnedSTMWord(curSTMWord)) {
         // Object was opened and closed by another transaction
         BecomeInvalid(tx);
         return false;
      } else if (GetOwnerFromSTMWord(curSTMWord) == tx) {
         // Object is currently opened by this transaction
         UpdateLogEntry *updateEntry =
            tx.GetEntryFromSTMWord(curSTMWord);
         if (updateEntry.stmWord != oldSTMWord) {
```

```
                // Object was opened and closed by another transaction
                BecomeInvalid(tx);
                return false;
            } else {
                // No intervening access by another transaction
            }
        } else {
            // Object is opened by another transaction
            BecomeInvalid(tx);
            return false;
        }
    } else if (GetOwnerFromSTMWord(curSTMWord) == tx) {
        // Object was opened by us for update before opening it for reading
    } else {
        // Object was opened by another transaction for update before we opening it for reading
        BecomeInvalid(tx);
        return false;
    }
    return true;
}
```

Closing an object opened for update has two effects: increment the object's version number and release the exclusive lock held by the transaction:

```
void CloseUpdatedObject(TMMgr tx, object obj, word oldSTMWord) {
    word newSTMWord = GetNextVersion(oldSTMWord);
    CloseSTMWord(obj, new_word);
}
```

TMAbort aborts a transaction by restoring the locations that it modified and by releasing its exclusive locks. Objects opened for reading require no action:

```
bool TMAbort(TMMgr tx) {
  foreach (UndoLogEntry e in tx.undoLog) {
    object o = e.obj;
    int offset e.offset;
    object value = e.value;
    o[offset] = value;
  }
```

```
  foreach (UpdateLogEntry e in tx.updateLog) {
    CloseSTMWord(e.obj, e.stmWord);  // do not change version number
  }
  return true;
}
```

Compiler Optimizations

The Bartok compiler, like the StarJIT compiler in the McRT-STM (Section 3.5.3), performs optimizations to eliminate unnecessary calls on the STM system. These optimizations fall into a number of categories:

- Conventional compiler optimizations can optimize BSTM operations whose semantics are known to the optimizer. For example, `TMGetTMMgr` returns a constant result within a transaction, and consequently can be moved out of a loop by loop-invariant code motion. `TMOpenForRead` and `TMOpenForUpdate` are idempotent within a transaction, and so duplicate calls can be eliminated.

- Objects allocated within a transaction need not be logged, since they cannot be referenced by another transaction and are discarded (and garbage collected) if the transaction aborts. Bartok uses a simple dataflow analysis to determine which variables always contain newly allocated objects and eliminates logging on these variables.

- Redundant open operations—for example, a method opening an object already opened by its caller—can be eliminated by moving calls on `TMOpenForRead` or `TMOpenForUpdate` to a method's caller and eliminating the redundant call there.

- Objects that are first read, then updated, for example in the statement `o.a = o.a + 1`, incur two open operations. The call on `TMOpenForRead` can be avoided by initially opening the object for update.

- The buffer overflow checks in the logging operations in a transaction can sometimes be optimized into a single test, which checks whether there is sufficient space for all of the data logged in the transaction.

These optimizations can reduce the number of log entries by 40–60% on a variety of benchmarks, with a roughly similar improvement in execution time.

Run-time Optimizations

Even after the compiler optimization, the logs contain many duplicate and unnecessary entries. BSTM uses run-time tests to eliminate more unnecessary log entries. The expense of these

tests must be traded off against a reduction in memory usage and in the cost of committing or aborting a transaction. These optimizations fall into several categories:

- Objects allocated in a transaction do not need undo log entries, since these objects become unreachable if the transaction aborts. Static compiler analysis cannot identify all of these objects. BSTM tracks objects allocated in a transaction and does not record them in the undo log. However, these objects require read and update log entries, so they can be committed.

- BSTM uses a hash-table filter to detect and eliminate duplicate entries in the read and undo logs.

- Bartok's garbage collector (GC) compacts the logs when it performs a garbage collection. This compaction removes unreachable (garbage collected) objects' log entries, entries in the read log for objects subsequently opened for update, and duplicate log entries.

Although these run-time optimizations can increase some programs' execution time, they can also dramatically reduce the execution time of other applications by a large amount.

3.5.5   McCloskey, Zhou, Gay, Brewer, POPL06

McCloskey et al's Autolocker system adopted a different approach to implementing concurrency control for transactions [40]. It used pessimistic concurrency control and two-phase locking, the approach Lomet proposed (Section 3.3.1). With this system, the code generated for a transaction acquires locks for all shared locations accessed by the transaction. The locks prevent other transactions from accessing the locations until the first transaction completes and releases them. The system ensures that locks are acquired in a well-defined order to prevent a deadlock. Transactions do not abort, but they may block waiting for a lock.

The Autolocker system transforms C code by inserting the lock acquire and release operations. A programmer delimits transactions in an atomic block and supplies annotations that relate a lock to the shared data it protects. An annotation names an explicit lock (either exclusive or reader/writer) and a data structure. The Autolocker tool inserts code to acquire the lock on entry to an atomic block that accesses the structure and to release the lock on exit. Autolocker will reject a program if it cannot determine that the locks it needs can be acquired in a deadlock-free order.

Hicks et al. described a technique for automatically inferring these locks and a consistent locking order (Section 3.5.6). McCloskey, on the other hand, argues that explicit (programmer-specified) locks provide control over locking granularity, which can improve performance by increasing concurrency with finer grained locking. By separating the specification of locking from its implementation and automating the insertion of lock acquires and releases, Autolocker

supports a composable locking model (albeit one that requires global program analysis) and facilitates modification and evolution of code.

The paper also argues that the optimistic concurrency mechanisms in STM systems are handicapped by their need to roll back transactions, a mechanism that coexists poorly with other system facilities such as IO. The paper also criticizes these systems for not providing programmers with mechanisms to control performance. McCloskey concedes, however, that optimistic locking is valuable in situations in which fine-grain locking algorithms are complex to implement, for example, structures such as red-black trees.

STM Characteristics	
AUTOLOCKER	
Strong or Weak Isolation	Weak
Transaction Granularity	User-selectable
Direct or Deferred Update	Direct
Concurrency Control	Pessimistic
Synchronization	Blocking
Conflict Detection	None (exclusive locking)
Inconsistent Reads	None (exclusive locking)
Conflict Resolution	None (exclusive locking)
Nested Transaction	
Exceptions	

3.5.6   Hicks, Foster, Pratikakis, TRANSACT 06

Hicks et al., like the Autolocker system (Section 3.5.5), use pessimistic concurrency control and two-phase locking to insert locks in C code containing explicit atomic blocks [41]. Before executing, a transaction acquires exclusive locks for all shared locations it will read or write. The locks ensure exclusive access to the location, until the transaction commits and releases them.

The contribution of this paper is a technique to infer automatically the collection of locks that a transaction must acquire and to determine an order in which to acquire them to avoid deadlock. The technique relies on a points-to–analysis; a standard, scalable compiler analysis that creates an abstract name to represent one or more memory locations and computes the mapping from a variable reference to the set of names representing locations accessed through the variable.

Hicks associates a lock with each symbolic name and computes a total order on these locks. At the start of a transaction, generated code acquires the locks for the transaction, based on the points-to-sets of the statements in the transaction. The locks are acquired in an order consistent with the total order. The paper also describes optimizations that eliminate locks on locations not shared between threads and that recognize that one lock subsumes another, which eliminates the need to acquire the latter. Until this system is implemented and evaluated, it will not be clear if points- to-sets are precise enough to allow fine-grain locking or if the large number of points-to-sets accessed in a transaction will produce high locking overhead.

STM Characteristics	
HICKS	
Strong or Weak Isolation	Weak
Transaction Granularity	Variable
Direct or Deferred Update	Direct
Concurrency Control	Pessimistic
Synchronization	Blocking
Conflict Detection	None (exclusive locking)
Inconsistent Reads	None (exclusive locking)
Conflict Resolution	None (exclusive locking)
Nested Transaction	
Exceptions	

3.6   LANGUAGE-ORIENTED STM SYSTEMS

Beyond the implementation of STM systems themselves, research has used STM systems to explore the semantics and programming models for transactional memory. Harris and Fraser proposed and implemented extensions to Java, which made the `atomic` construct a programming construct, not just a notation for specifying behavior [42] (Section 3.4.2). Harris's subsequent paper explored the interaction between exceptions and IO and transactions (Section 3.6.1). Pizlo et al. showed how transactional memory could help resolve priority inversion problems in Real-Time Java (Section 3.6.2). Harris et al.'s influential paper on Haskell STM introduced the `retry` and `orElse` statements and showed how a type system could aid in the static verification of transactions' properties (Section 3.6.3). Ringenburg and Grossman implemented

AtomCaml, which adds atomic execution to the Objective Caml (OCaml) language (Section 3.6.4).

3.6.1   Harris, CSJP04

Harris's paper [43] elaborated on two important issues left undiscussed in his and Fraser's OOPSLA paper (Section 3.4.2) [18]. The first is the semantics of exceptions within atomic blocks. In the earlier paper, an exception terminated an atomic block, but did not abort it. That is, any side effects produced by code in the atomic block, up to the point of the exception, remained visible after the exception transferred control out of the block. This paper explored an alternative semantics: an exception causes an atomic block to abort and roll back all side effects. The second issue is the inclusion of IO operations in an atomic block, which was prohibited in the earlier paper.

The paper motivates the change in the semantics of exceptions with an example that demonstrates the utility of transactions as an error recovery mechanism (an aspect explored elsewhere [6, 39]). Consider moving an item between two lists:

```
Bool move(List<Item> s, List<Item> d, Item item) {
   atomic {
        if (!s.Remove(item)) { /* R1 */
              return false;  /* Could not find object */
   }
   else {
        try {
           d.Add(item); /* A1 */
        }
        catch (RuntimeException e) {
              s.Add(item); /* A2 */
              throw e; /* Move failed */
        }
        return true; /* Move succeeded */
     }
   }
}
```

Without atomic blocks, the code needs to provide explicit compensation (statement A2) to return the item to the first list, if the insert into the second list fails (note, moreover, that if the reinsertion fails, then the item is lost).

Allowing an exception to abort an atomic construct simplifies the code by eliminating the need for explicit error recovery code:

```
Bool move(List<Item> s, List<Item> d, Item item) {
   try {
      atomic {
         if (!s.Remove(item)) { /*  R1  */
            return false;  /* Could not find object */
         } else {
            d.Add(item); /*   A1  */
            return true;  /* Move succeeded */
         }
      }
   } catch (RuntimeException e) {
      return false;  /* Move failed */
   }
}
```

If statement R1 or A1 fails with an exception, the atomic statement aborts and rolls back changes to both lists, and so leaves the program's state at the point it was when control first entered the function. (Harris's example is slightly more complex, because he distinguishes between classes of exceptions that terminate and abort atomic blocks, and so needs to translate the RuntimeException exception into an aborting exception.)

Harris pointed out that this semantics for exceptions causes problems:

- The rollback may violate invariants that previously held after the exception occurred. These invariants might expose intermediate program states. It is easy to argue that this is an undesirable state of affairs, but it is probably common in practice, since explicit compensation code is complex and difficult to implement correctly.

- When an atomic block rolls back a program's state, it also discards the exception object allocated within the block. This object requires special treatment, to pass it across the atomic block's boundary. Harris suggests serializing the exception object into a byte array—the mechanism used to pass objects between machines. Additional restrictions on this object are probably also necessary, since the exception object can point to objects allocated both before and within the atomic block, and this serialization does not

preserve sharing of the former group of objects, nor does it provide a reasonable semantics for the second group.

This paper also raised a warning of the complexity of invoking IO operations in an atomic block. It described a wrapper library, similar to a transaction-processing monitor [44], that facilitated buffering data in IO classes. The wrapper provided methods that enabled the class to vote on whether to commit, and that provided notification of the surrounding transaction's abort and commit. The approach, unfortunately, has serious and obvious flaws that point out the need for considerably more research in this area.

For example, to make IO transactional, an input operation could buffer the values read, to allow them to be replayed if the atomic block aborted and the input statement reexecutes. Similarly, output could be buffered and only transmitted when an atomic block commits. Unfortunately, even this simple buffering can introduce deadlocks, as demonstrated by a simple example of printing a prompt string and waiting for user input. The string only appears when the surround atomic block commits, which of course cannot happen until the user types his or her response.

Moreover, Harris pointed out that integrating database operations into atomic transactions is difficult, without a two-phase commit protocol for each transaction that would permit the other to cause an abort.

3.6.2   Pizlo, Prochazka, Jagannathan, Vitek, CJSP04

Pizlo et al. describe an application of transactional memory to resolve priority inversion problems in Real-Time Java, which is an extended version of Java for embedded systems [45]. Priority inversion occurs when a low-priority thread running in a critical section precludes a higher priority thread from entering the section, thereby delaying its execution despite the threads' priorities.

Pizlo el al. proposed transactional lock-free (TLF) objects, which support atomic methods on objects. These methods are transactional critical sections, in which a high-priority thread can preempt a lower priority thread and evict it from an atomic method. The state changes caused by the low-priority thread are rolled back before the high-priority thread enters the critical section, thereby hiding the potentially inconsistent state resulting from early termination.

Atomic methods, unlike other STM systems, use mutual exclusion to provide a critical section for only a single object (similar to Java's synchronized methods). The conflict resolution policy, in addition, is closely tied to thread priority so that a high-priority thread always aborts a lower priority thread.

Within an atomic method, writes to the object are logged, so they can be rolled back if a higher priority thread aborts the method.

STM Characteristics	
TLF	
Strong or Weak Isolation	Weak
Transaction Granularity	Object (single)
Direct or Deferred Update	Direct
Concurrency Control	Optimistic
Synchronization	Blocking (single lock)
Conflict Detection	None
Inconsistent Reads	None
Conflict Resolution	Abort lower priority transaction
Nested Transaction	Flatten
Exceptions	Terminate

3.6.3   Harris, Marlow, Peyton Jones, Herlihy, PPoPP 05

This paper describes the HSTM system for Concurrent Haskell [46], a lazily evaluated functional language [47]. At first glance, this combination may seem like a strange mixture, since functional languages are not supposed to perform side effects, and so they should not have conflicts between concurrent threads. However, even functional languages must perform IO, a side effect, so Haskell introduced a mechanism, called a monad, which allows side-effecting operations to coexist with a pure, functional base language. Moreover, communication between concurrent threads is also a side effect that requires a synchronization mechanism such as transactions.

Haskell's clean semantics and robust type system provided the basis to explore several interesting extensions to previous STM systems:

- HSTM used the Haskell type system to distinguish atomic statements from other side-effecting IO statements and to prevent the latter from appearing in atomic blocks.
- It also used the type system to ensure that transactional locations were only referenced within atomic blocks, thereby trivially ensuring strong isolation without the need to instrument every variable access.

- HSTM refined an idea from Harris and Fraser [18] by using conditional variables as the principal synchronization mechanism. HSTM made this mechanism an explicit programming construct.
- HSTM provided explicit operations to compose two transactions as alternatives.

STM Characteristics	
HSTM	
Strong or Weak Isolation	Strong
Transaction Granularity	Word
Direct or Deferred Update	Deferred (cloned replacement)
Concurrency Control	Optimistic
Synchronization	Blocking
Conflict Detection	Late
Inconsistent Reads	Inconsistency toleration
Conflict Resolution	
Nested Transaction	None (not allowed by type system)
Exceptions	Abort

Haskell STM Extensions

HSTM introduced new, language-appropriate mechanisms to express transactions. We will discuss the extensions without going into detail about the syntax or semantics of the Haskell language [48], which differs greatly from more common imperative languages, such as C and Java.

Transactions update values stored in transactional variables (TVar). A nontransactional variable is bound, so its value cannot be modified after it is initialized, and is obviously unaffected by transactions. A TVar is read by an explicit readTVar operation and modified by writeTVar. For example,

```
type Resource = TVar Int
putR :: Resource -> Int -> STM ()
putR r i = do { v <- readTVar r; writeTVar r (v+i) }
```

introduces a new type (`Resource`) for a TVar holding an integer and then defines function putR, which increments a `Resource` r by i units. Both `readTVar` and `writeTVar` return STM actions, which tentatively update `TVars`. The `atomic` function transactionally committed (or aborted) these updates:

```
main = do { ...; atomic (putR r 3); ... }
```

Haskell's type system has a dual perspective that splits the world into disjoint halves. STM operations, including updating `TVars`, are functional statements allowed in an `atomic` function. Moreover, STM actions, such as accessing `TVars`, can only appear inside a transaction, so there cannot be a conflict between a transaction and a statement executed outside of an atomic block. IO operations, by contrast, can only execute outside of a transaction.

HSTM also introduced an explicit `retry` statement as a coordination mechanism between transactions. The `retry` statement aborts the current transaction and prevents it from reexecuting until at least one of the `TVars` accessed by the transaction changes value. For example,

```
getR . Resource.Int.STM ()
getR r i = do { v <- readTVar r
              ; if (v < i) then retry
                else writeTVar r (v-i) }
```

atomically extracts i units from a `Resource`. It uses a `retry` statement to abort an enclosing transaction if the `Resource` does not contain enough units. If this function executes `retry`, r is the only `TVar` read, so the transaction reexecutes when r changes value.

This construct is a development of the conditional critical regions in [18], which only permitted a predicate to be evaluated before a transaction begins executing. HSTM's `retry` builds on Harris and Fraser's earlier observation that there is no reason to reexecute a transaction until another transaction modifies some part of the state the first transaction read.

HSTM also introduced the binary `orElse` operator for composing two transactions. This operator first starts its left-hand transaction. If this transaction commits, the `orElse` operator finishes. However, if this transaction retries, the operator tries the right-hand transaction instead. If this one commits, the `orElse` operator finishes. If it retries, the entire `orElse` statement waits for changes in the set of `TVars` read by *both* transactions before retrying. For example, this operator turns getR into an operation that returns a true/false success/failure result:

```
nonBlockGetR . Resource.Int.STM Bool
nonBlockGetR r i = do { getR r i ; return True }'orElse' return false
```

The left-hand transaction invokes the blocking version of `getR`. If it fails to find enough resources, and retries, the `orElse` operation invokes the right-hand transaction, which simply returns false.

In HSTM, an exception aborts a transaction (unlike Harris's first STM system). The paper notes that this design choice reflects the semantics of Haskell, which uses exceptions to signal error conditions, rather than effect control transfer. To get the opposite semantics, a programmer only needs to catch the exception within the transaction and return normally.

Implementation

The initial version of HSTM did not run on parallel machines, and consequently could take advantage of the well-defined points at which a thread context switch is allowed to occur in Haskell to avoid the need for low-level synchronization. A transaction's reads and writes to `TVars` all access a transaction log, which hides these variable references from other transactions. When the transaction commits, it first validates its log entries, to ensure that no other transaction modified the `TVars` values. If valid, the transaction installs the new values in these variables. If validation fails, the log is discarded and the transaction reexecuted. Since HSTM uses Haskell threads, validation and installing new values do not require low-level synchronization.

If a transaction invokes `retry`, the transaction is validated (to avoid retries caused by inconsistent execution) and the log discarded after recording all `TVars` read by the transaction. The system binds the transaction's thread to each of these variables. When a transaction updates one of these variables, it also restarts the thread, which reexecutes the transaction.

The `orElse` statement requires a closed nested transaction to surround each of the two alternatives, so that either one can abort without terminating the surrounding transaction. If either transaction completes successfully, its log is merged with the surrounding transaction's log, which can commit. If either or both transactions invoke `retry`, the outer transaction waits on the union of the `TVars` read by the transactions that retried.

3.6.4   Ringenburg, Grossman, ICFP 05

Ringenburg and Grossman implemented AtomCaml, which adds atomic execution to the Objective Caml (OCaml) language [49]. OCaml is mostly a functional language based on the ML programming language. It runs on uniprocessors, although it supports multiple threads with preemptive scheduling. The absence of true concurrency simplified the implementation (which only implemented failure atomicity, not isolation). Atomicity provides a useful mechanism to synchronize access to data shared among threads.

STM Characteristics	
ATOMICCAML	
Strong or Weak Isolation	Weak
Transaction Granularity	Word
Direct or Deferred Update	Direct
Concurrency Control	Optimistic
Synchronization	None
Conflict Detection	None
Inconsistent Reads	None
Conflict Resolution	None
Nested Transaction	Flattened
Exceptions	Terminate

Language Extensions

AtomCaml introduces two new programming constructs. The `atomic` primitive is a first class function, which accepts a function as its argument and executes it atomically. For example,

```
atomic (fun () .
   let totalWidgets= !blackWidgets + !blueWidgets in
      if totalWidgets > 0
      then print_string (pickWidget () ^ " available")
      else raise NoWidgets)
```

atomically reads the quantity of each widget (the ! operator) and then prints a result or throws an exception. If the thread running this atomic block runs without preemption, the block executes normally. However, if the thread is preempted, the side effects within the block are rolled back. The atomic statement reexecutes the next time the thread runs.

An uncaught exception in an atomic block is handled consistent with this semantics: it is a control transfer that leaves the atomic function and commits its side effects. Similarly, nested transactions need no special treatment. They all execute without preemption and commit, or their thread is preempted and they all reexecute.

The other language construction is a mechanism for conditional critical regions, similar to those in Harris et al. [18, 48]. OCaml provides a `yield` function, which yields control of the processor to another thread. In AtomCaml, `yield` terminates and rolls back an atomic

block, which subsequently can reexecute. AtomCaml supports an optional second argument to `yield`, which specifies a single mutable reference whose value must change before rescheduling the yielded thread.

Implementation
Without true concurrency, an `atomic` primitive is easily implemented. ML's type system distinguishes mutable from read-only data. Only the former needs to be logged and rolled back, because read-only values created in an atomic block have a shorter lifetime than the block (unless a mutable value is set to point to them) and consequently their values will be reevaluated when the block reexecutes.

Side effects new bytecodes added to the OCaml virtual machine log changes to the mutable global state. A modified compiler produces two versions of each function: one for use in an atomic block and one for use outside. When an atomic block terminates normally, its log is discarded. However, if the block is preempted, the log is rolled back and mutable locations are restored to the value they held on entry to the block.

REFERENCES
[1] M. P. Atkinson, and O. P. Buneman, "Types and persistence in database programming languages," ACM Comput. Surv., Vol. 19(2), pp. 105–170, 1987 doi:10.1145/62070.45066

[2] A. Hejlsberg, S. Wiltamuth and P. Golde, *The C# Programming Language*. Reading, MA: Addison-Wesley, 2004.

[3] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Fransisco, CA: Morgan Kaufmann, 2007.

[4] D.B. Lomet, "Process structuring, synchronization, and recovery using atomic actions," In *Proc. ACM Conf. on Language Design for Reliable Software*. Raleigh, NC, 1977, pp. 128–137 doi:10.1145/800022.808319

[5] K.P. Eswaran, et al., "The notions of consistency and predicate locks in a database system," *Commun. ACM*, Vol. 19(11), pp. 624–633, 1976 doi:10.1145/360363.360369

[6] J.J. Horning, et al., "A program structure for error detection and recovery," In *Proc. Int. Symp. on Operating Systems*. Berlin: Springer, 1974, pp. 171–187 (http://portal.acm.org/citation.cfm?id=647641.733522).

[7] T. Anderson and R. Kerr, "Recovery blocks in action: A system supporting high reliability," In *Proc. 2nd Int. Conf. on Software Engineering*, San Francisco, CA, 1976, pp. 447–457 (http://portal.acm.org/citation.cfm?id=807718).

[8] N. Shavit and D. Touitou, "Software transactional memory," In *Proc. 14th ACM Symp. on Principles of Distributed Computing*, Ottawa, Canada, 1995, pp. 204–213 doi:10.1145/224964.224987

[9] B. He, W. N. Scherer, III and M. L. Scott, "Preemption adaptivity in time-published queue-based spin locks," In *Proc. 12th Annu. IEEE Int. Conf. on High Performance Computing (HiPC)*, Goa, India, 2005 (http://www.cs.rochester.edu/~scherer/papers/2005-HiPC-TPlocks.pdf).

[10] Y. Afek, et al., "Disentangling multi-object operations," (extended abstract), In *Proc. 16th Annu. ACM Symp. on Principles of Distributed Computing (PODC)*. Santa Barbara, CA: ACM Press, 1997, pp. 111–120 doi:10.1145/259380.259431

[11] J.H. Anderson and M. Moir, "Universal constructions for large objects," *IEEE Trans. Parallel Distrib. Syst.*, Vol. 10(12), pp. 1317–1332, 1999 doi:10.1109/71.819952

[12] A. Israeli and L. Rappoport, "Disjoint-access-parallel implementations of strong shared memory primitives," In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, 1994, pp. 151–160 doi:10.1145/197917.198079

[13] M. Moir, "Transparent support for wait-free transactions," In *Proc. 11th Int. Workshop on Distributed Algorithms*. Berlin: Springer, 1997, pp. 305–319 (http://research.sun.com/people/moir/Papers/moir-wdag97.ps).

[14] G. Barnes, "Method for implementing lock-free shared data structures," In *Proc. 5th Annu. ACM Symp. on Parallel Algorithms and Architectures*, 1993, pp. 261–270 doi:10.1145/165231.165265

[15] J. Turek, D. Shasha and S. Prakash, "Locking without blocking: Making lock based concurrent data structure algorithms nonblocking," In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, 1992, pp. 212–222 doi:10.1145/137097.137873

[16] M. Herlihy, et al., "Software transactional memory for dynamic-sized data structures," In *Proc. 22nd Annu. Symp. on Principles of Distributed Computing*, Boston, MA, 2003, pp. 92–101 doi:10.1145/872035.872048

[17] R. Guerraoui, M. Herlihy and B. Pochon, "Toward a theory of transactional contention managers," In *Proc. 24th Annu. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing (PODC '05)*, Las Vegas, NV, 2005, pp. 258–264 doi:10.1145/1073814.1073863

[18] T. Harris and K. Fraser, "Language support for lightweight transactions," In *Proc. 18th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 03)*, Anaheim, CA, 2003, pp. 288–402 doi:10.1145/949305.949340

[19] C.A.R. Hoare, "Towards a theory of parallel programming," In *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott, Eds. New York: Academic Press, 1972, pp. 61–71.

[20] K. Fraser, *Practical Lock Freedom*. Cambridge, UK: University of Cambridge Computer Laboratory, 2004, p. 116 (http://www.cl.cam.ac.uk/users/kaf24/lockfree.html).

[21]   W. N. Scherer, III and M. L. Scott, "Contention management in dynamic soft-ware transactional memory," In *Proc. PODC Workshop on Concurrency and Syn-chronization in Java Programs*, 2004 (http://www.cs.rochester.edu/u/scott/papers/2004_CSJP_contention_mgmt.pdf).

[22]   W. N. Scherer, III and M. L. Scott, "Advanced contention management for dynamic software transactional memory," In *Proc. 24th Annu. ACM SIGACT-SIGOPS Symp. on Principles of Distributed Computing*. Las Vegas, NV: ACM Press 2005, pp. 240–248 doi:10.1145/1073814.1073861

[23]   R. Guerraoui, M. Herlihy and B. Pochon, "Polymorphic contention management," In *Proc. 19th Int. Symp. on Distributed Computing (DISC)*. Krakow: Springer, 2005, pp. 303–323 (http://lpdwww.epfl.ch/upload/documents/publications/neg–1700857499-main.pdf).

[24]   C.S. Ananian and M. Rinard, "Efficient object-based software transactions," In *Proc. OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005 (http://hdl.handle.net/1802/2102).

[25]   D.J. Scales, K. Gharachorloo and C.A. Thekkath, Shasta: A low overhead, software-only approach for supporting fine-grain shared memory, In *Proc. 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*. Cambridge, MA: ACM Press, 1996, pp. 174–185 doi:10.1145/237090.237179

[26]   G.J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.* , Vol. 23(5), pp. 279–295, 1997 doi:10.1109/32.588521

[27]   V.J. Marathe, et al., "Lowering the overhead of nonblocking software transactional memory," In *Proc. 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hard-ware Support for Transactional Computing (TRANSACT)*, Ottawa, Canada, 2006 (http://www.cs.purdue.edu/homes/jv/events/TRANSACT/).

[28]   D. Dice, and N. Shavit, What Really Makes Transactions Fasler?, *Proceed-ings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hard-ware Support for Transactional Computing (TRANSACT)*, Ottawa, Canada, 2006 (http://research.sun.com/scalable/pubs/TRANSACT2006-TL.pdf).

[29]   B.H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, Vol. 13(7), pp. 422–426, 1970 doi:10.1145/362686.362692

[30]   R. Ennals, "Efficient software transactional memory," Intel Research Cam-bridge Tech Report, Cambridge, England, 2005 (http://www.cambridge.intel-research.net/~rennals/051_Rob_Ennals.pdf).

[31]   K. Manassiev, M. Mihailescu, and C. Amza, "Exploiting distributed version concurrency in a transactional memory cluster," In *Proc. 11th ACM SIGPLAN Symp. on Principles and*

Practice of Parallel Programming (PPoPP '06). New York: ACM Press, 2006, pp. 198–208 doi:10.1145/1122971.1123002

[32]   A.L. Cox, et al., "Software versus hardware shared-memory implementation: A case study," In *Proc. 21st Annu. Int. Symp. on Computer Architecture*, 1994, pp. 106–117 doi:10.1145/191995.192021

[33]   S.V. Adve and M.D. Hill, "A unified formalization of four shared-memory models," *IEEE Trans. Parallel Distrib. Syst.*, Vol. 4(6), pp. 613–624, 1993 doi:10.1109/71.242161

[34]   B. Saha, et al., "McRT-STM: A high performance software transactional memory system for a multi-core runtime," In *Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '06)*. New York: ACM Press, 2006, pp. 187–197 doi:10.1145/1122971.1123001

[35]   A.-R. Adl-Tabatabai, et al., "Compiler and runtime support for efficient software transactional memory," In *Proc. 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2006)*. Ottawa, ON: ACM Press, 2006, pp. 26–37 doi:10.1145/1133981.1133985

[36]   B. Hindman and D. Grossman, *Strong Atomicity for Java Without Virtual-Machine Support*. Seattle, WA: Department of Computer Science and Engineering, University of Washington, 2006 (http://www.cs.washington.edu/homes/djg/papers/atomjava_tr_may06-abstract.html).

[37]   V.S. Menon, et al., "A verifiable SSA program representation for aggressive compiler optimization," In *Proc. 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*. Charleston, SC: ACM Press, 2006, pp. 397–408 doi:10.1145/1111037.1111072

[38]   T. Harris, et al., "Optimizing memory transactions," In *Proc. 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2006)*. Ottawa, ON: ACM Press, 2006, pp. 14–25 doi:10.1145/1133981.1133984

[39]   A. Shinnar, et al., *Integrating Support for Undo with Exception Handling*. Microsoft Research, 2004 (ftp://ftp.research.microsoft.com/pub/tr/TR-2004-140.pdf).

[40]   B. McCloskey, et al., "Autolocker: Synchronization inference for atomic sections," In *Proc. 33rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL)*. Charleston, SC: ACM Press, 2006, pp. 346–358doi:10.1145/1111037.1111068

[41]   M. Hicks, J.S. Foster and P. Prattikakis, "Lock inference for atomic sections," In *Proc. 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, Ottawa, Canada, 2006 (http://www.cs.umd.edu/~polyvios/publications/transact06.pdf).

[42]   C. Flanagan and S. Qadeer, "Types for atomicity," In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2003, pp. 1–12 doi:10.1145/604174.604176

[43] T. Harris, "Exceptions and side-effects in atomic blocks," In *Proc. 2004 Workshop on Concurrency and Synchronization in Java programs*, 2004, pp. 46–53 (http://research.microsoft.com/~tharris/papers/2004-csjp.pdf).

[44] P.A. Bernstein, "Transaction processing monitors," *Commun. ACM*, Vol. 33(11), pp. 75–86, 1990 doi:10.1145/92755.92767

[45] F. Pizlo, et al., "Transactional lock-free objects for real-time Java," In *Proc. PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, Newfoundland, 2004, pp. 54–62 (http://www.ovmj.org/transactions/papers/csjp04.html).

[46] S.P. Jones, A. Gordon and S. Finne, "Concurrent Haskell," In *Proc. 23rd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. St. Petersburg Beach, FL: ACM Press, 1996, pp. 295–308 doi:10.1145/237721.237794

[47] P. Hudak, et al., "Report on the programming language Haskell: A non-strict, purely functional language version 1.2," *SIGPLAN Not.*, Vol. 27(5), pp. 1–164, 1992 doi:10.1145/130697.130699

[48] T. Harris, et al., "Composable memory transactions," In *Proc. 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Chicago, IL, 2005, pp. 48–60 doi:10.1145/1065944.1065952

[49] M.F. Ringenburg and D. Grossman, "AtomCaml: First-class atomicity via rollback," In *Proc. 10th ACM SIGPLAN Int. Conf. on Functional Programming*, 2005 doi:10.1145/1090189.1086378

CHAPTER 4

Hardware-Supported Transactional Memory

4.1 INTRODUCTION

Hardware transactional memory (HTM) is a hardware system that support implementing nondurable ACI (failure atomicity, consistency, and isolation) properties for threads manipulating shared data. HTM systems typically have a moderate software component and a larger hardware component. Their goal is to achieve the best performance with low overheads. HTMs introduce new hardware features to track transactional accesses, buffer tentative updates, and detect conflicts, while STMs use software to do the same.

Although STM systems are more flexible than HTMs and offer some advantages (Section 3.1), HTM systems have several key advantages over STM systems:

- HTMs can typically execute applications with lower overheads than a.
  HTMs can have better power and energy profiles and higher performance.
  HTMs execute independent of compiler technology and independent of memory access characteristics.

- HTMs are minimally invasive in an existing execution environment as they can potentially accommodate transactions that call functions that are not transaction aware, transaction-safe legacy libraries, and third-party libraries.

- HTMs can provide high-performance strong isolation without requiring application-wide changes.

- HTMs are better suited for unmanaged and unsafe environments.

4.1.1 Chapter Overview

The chapter divides HTM research into five categories:

- Precursors (Section 4.3).
- Bounded/large HTMs (Section 4.4).

- Unbounded HTMs (Section 4.5).

- Hybrid HTM–STMs/hardware-accelerated STMs (Section 4.6).

- HTM semantics (Section 4.7).

The chapter does not present the papers in strict chronological order; instead, it discusses related approaches together. A basic division is between modern HTMs that support unbounded transactions and those that support large but bounded transactions. Most of the HTMs focus on mechanisms to increase the buffering for transactions seamlessly. Recent efforts broaden this focus to include ways of bridging the performance and flexibility gap between STMs and HTMs by either integrating more software components and language constructs in an HTM or by using HTM hardware to speed up an STM. The papers in the last section present a discussion of possible interfaces to an HTM.

In this chapter, we use "atomic" and its variants strictly in the sense of a set of committed memory operations appearing to occur instantaneously at a single point in some ordering of all memory operations in the system. All other operations are either ordered before or after this set of memory operations. We start by presenting a brief discussion of hardware mechanisms pertinent to HTM.

4.2   A FEW WORDS ON HARDWARE

Transactional memory assists parallel applications that run on multiprocessors. The communication mechanism between processors often differentiates multiprocessor systems. Two common communication mechanisms are message passing and shared memory. In message-passing systems, each processor has local memory accessible only to itself, and communicates through explicit messages. Shared-memory systems make at least part of the memory accessible to all processors. Processors communicate through read and write operations to memory. Shared-memory systems have emerged as the dominant class of systems due to the relative ease of writing shared-memory parallel programs as compared to message-passing programs.

Three aspects of hardware systems are pertinent to HTMs. Memory consistency models help to reason about the ordering of load and store operations in a multiprocessor system, cache coherence ensures that multiple processors have a coherent view of locally cached data and speculative execution techniques provide mechanisms to recover architectural register state and to tolerate the latency of serializing operations.

4.2.1   Memory Consistency Models

A *memory consistency* model defines the semantics of memory operation and allows programmers to use shared memory correctly. The memory consistency model specifies the behavior of memory with respect to read and write operations from multiple processors. The strictest, but

most intuitive, model considers the sequential semantics of memory operations in uniprocessors and views a multiprocessor as a multiprogrammed uniprocessor. Lamport formally defined this *sequential consistency* model in which the result of any execution is the same as if the operations of all processors were executed in some sequential order and the operations on each processor appeared in this sequence in the order specified by its program (an order known as program order) [30].

Sequential consistency provides the behavior that most programers expect [24]. Consider multiple processors sharing memory. Each processor issues its memory operation in program order. Memory operations execute one at a time; they appear to occur atomically with respect to other memory operations. The order of servicing of operations from different processors may be arbitrary, thus leading to an interleaving of memory operations from different processors into a single sequential order. An execution of a program is sequentially consistent if at least one execution exists on a sequentially consistent system that can produce the same result.

Besides caches, many modern processors use write buffers (also known as store buffers) between the processor and caches, to allow a processor to continue executing while a store operation is made visible through the cache coherence protocol. Caching and write buffering introduce complexity in the specification, definition, and implementation of memory consistency models. This has led to significant work in the area in relaxing memory ordering [15].

Broadly, memory consistency models are differentiated based on two characteristics [2]:

1. How is the program order of memory operations relaxed? This may involve relaxing the order—when the operations access different addresses—of a write following a read, a read following a read or write, or between two writes.

2. How is write atomicity relaxed? This relaxation may allow a read to return the value of *another* processor's write before all other processors have seen the write. Here, the relaxation applies to operation pairs on the same address.

Sequential consistency enforces strict ordering requirements, but relaxed models allow relaxing these orderings. With relaxed models, programmers can use explicit mechanisms to prevent such reordering from occurring. A tutorial on shared-memory consistency models by Adve and Gharachorloo [2] provides background on various memory consistency models.

An open question concerning transactional memory is which memory model programmers should assume when programming with transactional memory, a topic briefly discussed in Section 2.1.2.

4.2.2   Caches and Cache Coherence

Processors use local caches to store frequently referenced data. This reduces long latency memory operations and bandwidth requirements. However, caching in shared-memory multiprocessors

results in multiple copies of a given memory location. Cache coherence is the mechanism used to keep all copies up-to-date. Broadly speaking, the mechanisms underlying any cache coherence protocol are as follows:

1) Mechanisms to locate all cached copies of a memory location.
2) Mechanisms to keep all cached copies of a memory location up-to-date.

Snoop-based and directory-based are two common schemes for locating copies of a memory location. A snoop-based coherence protocol broadcasts the address of a memory location to all caches. A directory-based coherence protocol maintains a directory per memory location to record a where all the copies of a location reside. Various alternatives exist, but all require some mechanism to locate all cached copies.

A write operation to a cache must keep all cached copies up-to-date. This operation often involves either invalidating stale (out-of-date) copies or updating the cached copies to the newly written value. If two processors simultaneously issue a write to the *same* location with different values, cache coherence protocols ensure that all processors observe the two writes in the same order, with the same value persisting in all copies.

In most invalidation-based cache coherence protocols, the cache with the dirty copy (i.e., the cache that owns the copy and that has modified it with respect to memory) of the cache line is responsible for servicing read requests from other processors for either shared copies or exclusive copies of the cache line. A cache that does not have the cache line in dirty state need not respond. For such caches, an incoming read request is simply ignored and an incoming read-for-exclusive-ownership request is treated as an invalidate request.

Systems maintain cache coherence at the granularity of a cache line (64 bytes being a common size) and processors perform fetches and invalidations at the granularity of a cache line. While larger granularity improves performance if data accesses have spatial locality, poor spatial locality degrades performance due to *false sharing*. Goodman and Woest [18] were the first to define false sharing as a situation when two processors alternately read or write different parts of the same cache line, resulting in the line being moved repeatedly between the two processors as if the data were shared, when in fact no values are flowing between the processors.

Sweazey and Smith [48] classified cache coherence protocols based on the stable states of a cache line, and proposed the modified, owned, exclusive, shared, and invalid (MOESI) classification. A cache line in stable state has valid data and has no outstanding state transition:

- *Modified*. The line is dirty (memory copy is stale) and exclusively owned by the cache.
- *Owned*. The line is dirty (memory copy is stale), the line is possibly shared among multiple caches, and this cache is responsible for ensuring memory is kept up-to-date.

- *Exclusive*. The line is clean (memory copy is up-to-date) and exclusively owned by the cache.

- *Shared*. The line is clean (memory copy is up-to-date) and the line is possibly shared among multiple caches.

- *Invalid*. The line is not present in the cache.

While stable states identify the cache coherence protocol, implementing high-performance cache coherence protocols often requires additional states, also known as transient or pending states. This is because a delay exists between the request initiation and request completion phases of a memory operation. The processor may perform other operations during this delay. A cache line makes a transition out of the stable state (typically one of the MOESI states or variants) at the request initiation phase and makes another transition into a stable state at the end of the request completion phase (which may involve the completion of data transfer). The cache line remains in a pending state between the two phases and may transition to multiple pending states depending on the coherence events occurring. Hennessy and Patterson [21] provide an example to demonstrate the complexity introduced by the addition of pending states to a cache coherence protocol.

4.2.3   Speculative Execution and Modern Processors

Programs have a sequential execution model based on a simple processor model. In this model, a program counter identifies the instruction that the processor fetches from memory. The processor executes the instruction, may reference memory as part of the instruction, and may operate on registers in which data may be stored. When the execution completes, the program counter is incremented to identify the next instruction to execute. In a parallel program, each individual thread or task executing on a processor executes the above sequence.

While processors still maintains a sequential execution model, their implementations often perform tasks in parallel. The processor takes in the sequential specification in the program, and uses various mechanisms, including control, data, and dependence prediction, to execute instructions in parallel. Thus, the processor, while maintaining the appearance of a sequential execution, is internally executing a parallel version of the sequential program, in which multiple instructions execute in an out-of-order fashion. Processors record sufficient recovery information that, on an incorrect prediction, they restore state and restart execution from a prior point. This includes restoring any register state in processors. Numerous schemes exist for such designs, and Smith and Sohi provide an overview of modern high-performance processor designs [44].
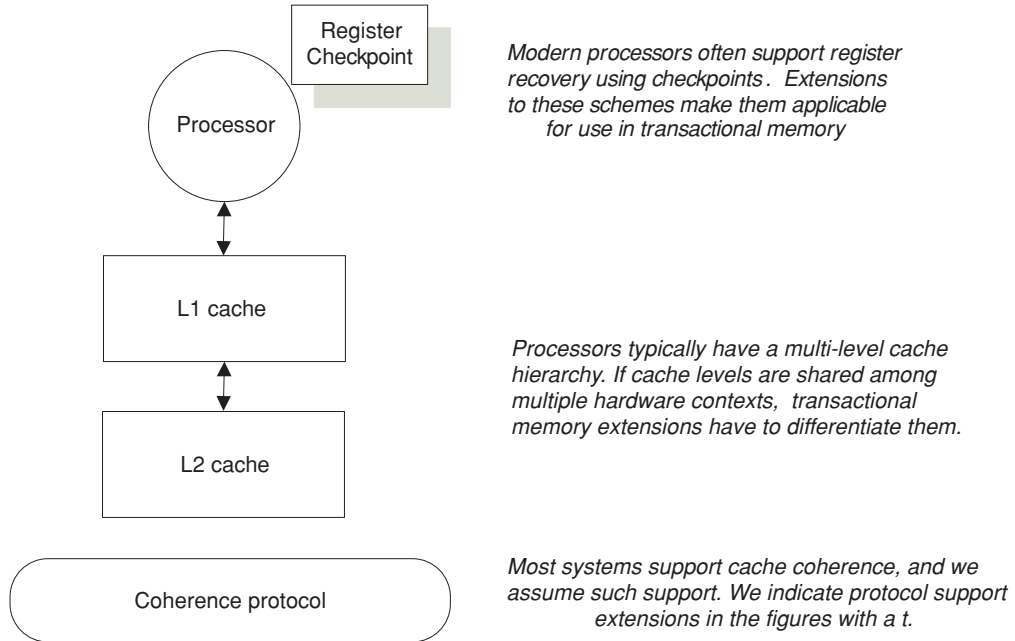
FIGURE 4.1: Baseline hardware organization

Next to the processor/register checkpoint:

Modern processors often support register recovery using checkpoints. Extensions to these schemes make them applicable for use in transactional memory

Next to the caches:

Processors typically have a multi-level cache hierarchy. If cache levels are shared among multiple hardware contexts, transactional memory extensions have to differentiate them.

Next to the coherence protocol:

Most systems support cache coherence, and we assume such support. We indicate protocol support extensions in the figures with a t.

4.2.4   Baseline Hardware Framework

Transactional memory relies on mechanisms that enable a processor to execute a sequence of instructions optimistically, to detect conflicting data accesses with other concurrently executing transactions, to use caches to buffer updates temporarily and not make any intermediate updates visible until commit, to make all state instantaneously visible on commits, and to discard any updates and register modifications on an abort and restore state to a known good point in the past. The Fig. 4.1 shows the baseline organization of a hardware system that we will use in this book to discuss various HTM proposals.

HTMs exploit various hardware mechanisms such as speculative execution, register checkpoints, caches, and cache coherence. Speculative execution techniques allow nonserialized execution of transactional memory regions. Register checkpoints enable a processor to execute a code segment (similar to how a branch is predicted) with high performance and without requiring a compiler to have visibility into the function for optimizing register saving. Cache coherence mechanisms allow for no-overhead conflict detection among concurrently executing transactions, and caches provide the ability to record reads and writes of a transaction, and buffer intermediate uncommitted state. Various HTM proposals use some or all the above mechanisms.

4.3   PRECURSORS

We start with two seminal works on hardware support for transactions. The 801 Storage Manager added hardware for database transactions and was the first industrial implementation of transactional locking in hardware (Section 4.3.1). It introduced lock bits in page table entries and translation look-aside buffers, and implicitly invoked hardware assists to call software routines for lock management. Knight added hardware for speculatively parallelizing single-threaded programs (Section 4.3.2). He proposed the use of caches to buffer speculatively updated memory state and to track read and write sets, and the use of a cache coherence protocol for conflict detection. While not discussed in this book, the Multiscalar work [45] popularized and drove later research into speculative parallelization. Other speculative parallelization proposals followed the Multiscalar work [19, 46].

The next three papers focused on transactional memory. Jensen et al. presented the first proposal for optimistic synchronization on a single word (Section 4.3.3). The other two papers presented proposals for optimistic synchronization over multiple words; Stone et al. used reservation registers to specify the multiple locations (Section 4.3.4), while Herlihy and Moss used a special cache (Section 4.3.5). These proposals assumed simple atomic updates, limited the size of a transaction to the quantity of local reservation registers or cache space, and restricted atomic updates to one scheduling quantum.

The last two papers in this section, by Rajwar and Goodman (Sections 4.3.6 and 4.3.7), connected critical sections with transactional memory and showed how to use speculative execution mechanisms such as register checkpointing and speculative write buffering to execute critical sections as transactions. These proposals executed and committed lock-based critical sections without any thread acquiring locks, thus achieving lock-free execution when possible. Other proposals for speculative execution of critical sections focused on overlapping execution latency with lock acquisition latency, maintained locking behavior, required one thread to always acquire a lock, and serialized commit operations on the lock acquisition or release [16, 33, 42].

4.3.1   Chang and Mergen, ACM TOCS 1988

Overview

This paper describes the IBM 801 storage manager system [10, 37]. The system provided hardware support for locking in database transactions. Specifically, the system associated a lock with every 128 bytes in each page under control of the transaction locking mechanism. It extended the page table entries (PTE) and translation look-aside buffers (TLB) to do so. If during a transaction, a memory access did not find the associated lock in the PTE in an expected state, the hardware would automatically invoke a software function. This software function then

performed the necessary transaction lock management. This implicit invocation of transaction management functions by hardware was analogous to the implicit invocation of page fault handlers by hardware to implement virtual memory. The paper argued that removing the need to insert library calls directly into software and instead having the hardware invoking them simplifies software development and engineering. Requiring explicit transaction management function calls wherever the program references transaction data presents difficulties for multiple languages. In addition, the called functions need use-specific parameters, thus complicating software engineering.

The proposal extended the processor with new architectural registers to capture information about the executing transaction. It extended the page tables to record which transaction owned a lock in a page and provided locking at 128-byte granularity. If the executing transaction accessed such a page and its transaction identifier did not match the owner of the lock, then the processor triggered a hardware assist, called the lock-interrupt fault. The hardware assist transfered control to a software function, called the lock-fault handler, which performs appropriate transactional lock management functions.

This is, to the best of our knowledge, the first industry implementation of transactional locking in hardware. The paper describes a uniprocessor implementation and discusses details in terms of files on disk. The ideas, however, are equally applicable to a multiprocessor implementation and memory.

Implementation

The paper describes transactions over a set of file segments. The paper defines a transaction as all operations performed by a process between two calls to a commit operation. Each process has a unique transaction identifier, the `tid`. The processor loads the `tid` of the running transaction into its `transaction id` register.

Each segment register in the 801 architecture has an `S` protection bit. If the `S` bit was 0 the conventional page protection is in effect and if the `S` bit was 1, the transaction locking mechanism was applicable to the segment.

The 801 extended each page table entry (PTE) with three additional fields: a `tid`, `W`, and `lockbit` fields. A lock bit in the `lockbit` field exists for each line of 128 bytes in the page. With 2 kB pages, each PTE entry has 16 `lockbit` fields. Each PTE contains the locks of one transaction in that page, and the `tid` field identifies the transaction. The `W` bit determines whether the lock bits represent write locks or read locks. The hardware allowed write access to line only if both, the line's lock bit in the `lockbit` field and `W` are 1. Read access to a line is allowed if the line's lock bit in the `lockbit` field is 1 or if `W` is 1. The `tid` in the `transaction id` register must match the `tid` in the PTE for access to the segment. The hardware triggered
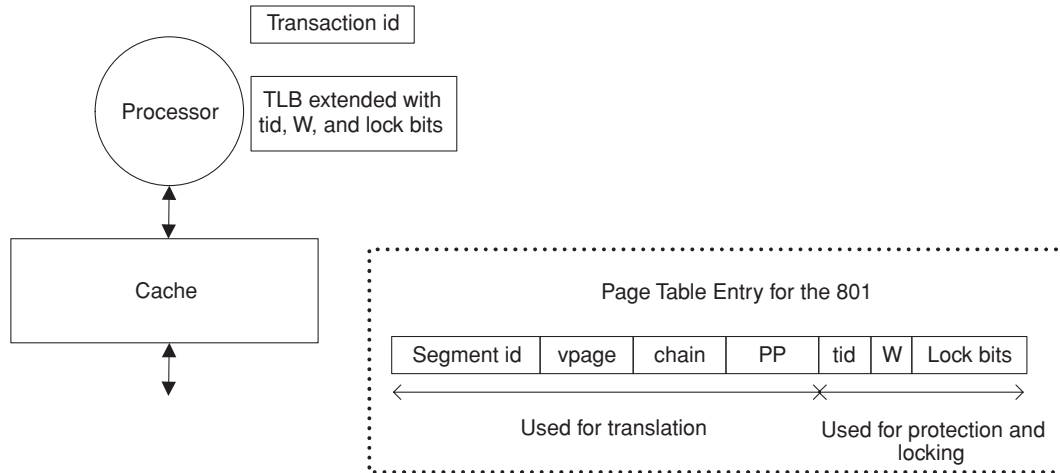
FIGURE 4.2: The 801 transaction locking extensions

a lock interrupt fault when a transaction accesses a segment to which it does not have access permissions. This redirects control to a software function.

Fig. 4.2 shows the architectural extensions. All transaction management is implemented in software.

The system maintains information about locks in a separate software structure called the lock table. Each entry in the lock table, called `lockwords`, has fields similar to the PTE. A `lockword` is allocated when a transaction first accesses a page following a commit, and it is freed on the next commit. Each page accessed in the transaction since the last commit allocates a `lockword`. Two lists are maintained to allow fast accesses to this table. One list is accessed using the `tid`, to identify all `lockword` entries belonging to the `tid`. Commit operations use this list to find all the locks held by the committing transaction in all pages. The other list is accessed in a the `segment id` and `vpage`. The lock-fault handler uses these lists to find all locks of any transaction in the referenced page, to detect conflicts, and to remember locks granted or released.

The hardware triggers a lock-fault interrupt if the locks in the PTE for a page are not those of the current transaction, and control transfers to the lock-fault handler. The handler searches the lock table and makes the transaction wait if there are conflicting locks, or grants and adds the requested lock in the table.

The lock table sits in pageable memory and is accessed in a critical section. The lock table is similar to the ownership tables in modern word-based software transactional memory systems (Chapter 3).

The system provides two options to open a file. If the `locks` option is used, then the file segment register's S bit is set to 1. A `journal` option allows reads without read locks and avoids certain deadlocks under careful use.

CHANG AND MERGEN 1988	
Strong or Weak Isolation	N/A
Transaction Granularity	128 bytes
Direct or Deferred Update	N/A
Concurrency Control	Function of database
Conflict Detection	Early
Inconsistent Reads	Optional
Conflict Resolution	Software
Nested Transaction	Nested

4.3.2   Knight, LFP 198

Overview

Knight describes a hardware system to parallelize a single thread program speculatively, and to execute it on a multiprocessor system. A compiler divides a program into a series of code blocks called transactions. For doing the division, the compiler assumes that these transactions do not have memory dependencies. These blocks then execute optimistically on the processors. The hardware enforces correct execution and uses caches to detect when a memory dependence violation between threads occurs [27].

In Knight's proposal, each processor has two caches, one to track memory dependences and the other to buffer temporary updates. The serial order of transactions in the sequential program determines the order in which transactions commit. When a processor is ready to commit its transaction, it waits for its transaction to become the next to commit. When this happens, the processor broadcasts the transaction's cache updates to all other processors. Other processors then match the incoming writes to their transaction's reads and writes. If a processor detects a conflict, it aborts and restarts the transaction. The processor performing the broadcast successfully commits.

This is the first paper, to the best of our knowledge, that proposed to use caches and cache coherence to maintain ordering among speculatively parallelized regions of a sequential code in the presence of unknown memory dependences. While the paper did not directly address explicitly parallel programming, it set the groundwork for using caches and coherence protocols for future transactional memory proposals.
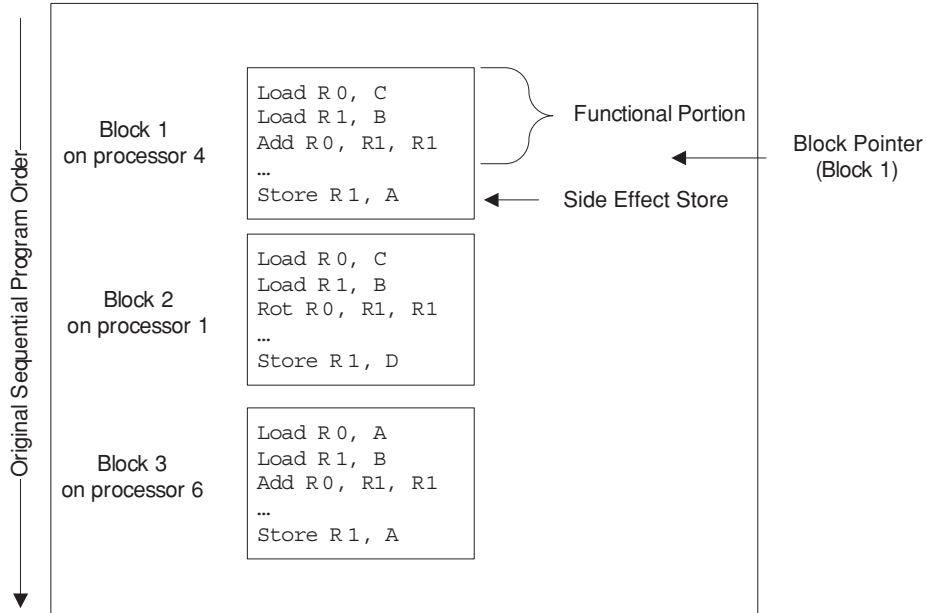
FIGURE 4.3: The compiler divides a sequential program into a series of transactions

Programming Interface

Knight describes his system in the context of a LISP application. The compiler transforms a sequential LISP program into a series of transactions. The compiler uses many heuristics for the transformation. These include traversing a call-tree to determine the order of execution, allowing dual-path execution at conditional branches and discarding the wrong path, using loops as transactions, and predicting whether a value of a memory location changes at commit time. The programmer or compiler was not responsible for checking memory dependencies.

Each block manages its own register state and no dependencies are carried through registers between transactions. However, dependencies between transactions could exist through memory. Each block terminated at one *final side-effecting* store. A side-effecting store means that another transaction can potentially read the location modified by the store. A transaction could have multiple side-effecting stores but only one final side-effecting store. The order of committing transactions was captured by the *block pointer* and was based on sequential program order. Fig. 4.3 shows a sequential program divided into a series of transactions.

Implementation

The proposal assumes a shared-memory multiprocessor and a network supporting broadcasts and snooping [17]. Each processor executes a transaction until the final side effect store. At
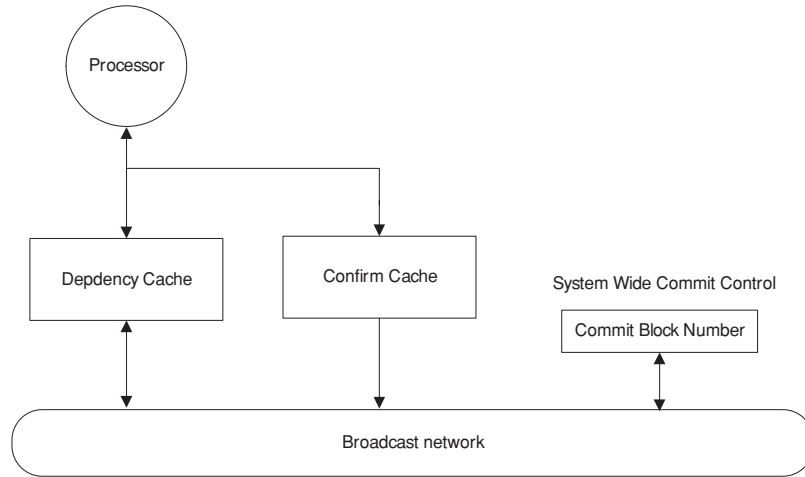
FIGURE 4.4: Knight's hardware organization

that point, the processor waits until it can commit. During execution, the processor maintains a dependency list recording all memory locations accessed in the block and updates its cache, but does not make the updates visible to other processors. Commit, called *confirming*, occurs when the system's transaction block counter identifies this processor's transaction block as the next to commit. The processor then broadcasts all writes to other processors and other processors detect conflicts. Fig. 4.4 shows the hardware organization.

The processor is augmented with two fully associative caches. The *dependency cache* holds data read from memory. It also watches for side-effecting writes from other processors. The *confirm cache* temporarily holds data written in the transaction block. On read operations, the processor first accesses the confirm cache and on a miss in accesses the dependency cache. The cache lines in the confirm cache are discarded when a transaction is initialized. The confirm cache is scanned at commit and modified lines are broadcast to other processors.

The dependency cache has three basic states (INVALID, VALID, and DEPENDS) and three heuristic-driven states (PREDICT/VALID, PREDICT/INVALID, and PREDICT/ABANDON). The INVALID state means the cache-line data and state are not valid, the VALID state means the line has up-to-date value from memory, and the DEPENDS state means the cache line has the correct value from memory but the correctness of the execution depends on the continued correctness of this value. This latter concept is similar to a processor speculating on dependence through memory. The PREDICT/VALID state means that the transaction block correctly predicted the value for this line. The processor writes the predicted value in the local cache and, at commit, compares this value to the value in memory, and if correct, transitions the cache-line state into the PREDICT/VALID state. The PREDICT/INVALID state is similar to the PREDICT/VALID

state except the predicted value was different from the value in memory at the time of commit.

The dependency cache *snarfs* values on the bus/network written as part of a confirm operation by other processors. A mismatch of the new content from the old content implies that a prediction was incorrect. *Snarfing* is a technique supported in some coherence protocols where a processor can observe bus activity and view/copy data exchanges between two other processors.

The architecture provides a `Cons` operator (`Cons` constructs memory objects holding two values or pointers). A program uses this to update an independent free pointer and restore its value following a transaction end. The `Cons` operator updates memory using a write-through technique. The Depends cache must not contain stale copies of data written with the `Cons` operator. The architecture provides an *unsafe* load to allow incoherence of iterative algorithms in explicitly parallel applications. The dependency list does not track unsafe operations.

An important difference between Knight's work and subsequent transactional memory works was the concept of a known total order of transactions. Since program order in a sequential program determines correctness, it accommodates situations in which the hardware cannot perform optimistic execution. This occurs if the local caches are insufficient to record the footprint of the transaction block or if IO or system calls occur. The system can wait until such an operation is the oldest in the system. However, a parallel application using transactional memory does not typically have a prior known commit order and thus cannot take advantage of such order. This difference has led to many challenges in transactional memory implementations.

KNIGHT 1986	
Strong or Weak Isolation	Strong
Transaction Granularity	Cache line
Direct or Deferred Update	Deferred (in cache)
Concurrency Control	Optimistic Commit serialized globally
Conflict Detection	Late write–write conflict Late write–read conflict
Inconsistent Reads	No
Conflict Resolution	Program order (sequential program)
Nested Transaction	N/A

4.3.3   Jensen, Hagensen, and Broughton, UCRL 1987

Overview

This paper describes architectural support for optimistic synchronization using a single memory location. Programmers could use such support for writing lock-free programs, thereby avoiding performance overheads associated with locking. The paper describes the idea as implemented in the S-1 AAP multiprocessor [25].

The paper replaces the complex `conditional_store` synchronization instruction (similar to a `compare&swap` instruction [7]) with a series of simpler instructions. This change exposes the latency of the simpler instructions to a compiler. The compiler could then schedule these instructions appropriately and thereby tolerate the latency inherent in a complex coherence protocol. The simpler instructions perform a load operation, set an appropriate condition to watch for, and then perform a store if the condition is still held.

The paper observed that a processor could use the cache coherence protocol to optimistically monitor a memory location for conflicts and conditionally perform operations if the location did not experience a conflict. This was the first proposal to split the `conditional_store` instruction into two instructions. The idea found its way into numerous commercial microprocessor instruction sets, including the MIPS [26], the PowerPC architecture [13], and the Alpha architecture [12].

Programming Interface

The paper introduced three new memory instructions. The programmer must take care that these instructions do not interact with regular load and store instructions:

1. `sync_load`. This instruction computes an address (called `xa-address`), indicates that the processor requires exclusive access to a synchronization block containing this address, and loads the accessed data into a general register. The computed address is treated as a sentinel for the synchronization block.

2. `sync_store`. This instruction is used for two actions:
    a. *Conditional stores*. If the `sync_store` succeeds, it conditionally stores data to memory. If the presumption of exclusive access in (1) above holds, the `sync_store` completes.
    b. *Selecting alternate path*. If the `sync_store` fails, the program's execution follows an alternate code path. In the S1-AAP implementation, if the instruction succeeds, the next instruction is skipped and a condition code is returned. If a failure occurs, then the next instruction is executed, allowing a retry path.

3. `sync_clear`. This instruction ends presumed exclusive access to a memory region. It does not store data and does not change program flow; it serves merely as a semantic

convenience. A common use is to execute this instruction as part of a context switch. This would abort any optimistic synchronization in progress by forcing a subsequent `sync_store` to fail.

The `sync_load` and `sync_store` form the read and write phases of the optimistic read–modify–write sequence of a `conditional_store` instruction. However, in a subtle but key difference, these load and store instructions need not be to the same data address. This allows optimistic monitoring of data itself and allows programmers to monitor one location and write another. For example, if the algorithm computes only one data item from a set of data, a programmer convention can establish one data item to define exclusive access over the entire data set. With this functionality, a programmer can write powerful synchronization constructs. However, the burden of correctness lies with the programmer and writing powerful synchronization constructs requires a level of sophistication from the programmer.

The code sequences shown below demonstrate the use of these new instructions to acquire a lock. In the S1-AAP, the `jump_q` instruction executes its following instruction only if the branch condition is true. If the `sync_store` instruction executes successfully, the processor skips executing the following instruction:

```
// if (lock == 0) { lock = ProcessID; }    %perform this atomically
// else goto LockHeld...                    %lock was held

Retry: sync_load R10, lock                  ; declare exclusive intent
   jump_q .neq (R10,0), LockHeld            ; test for zero
   sync_clear                               ; lock non-zero, hence abort
   load R10, ProcessID                      ; prepare to update lock
   sync_store R10, lock                     ; update lock if not aborted
   goto Retry                               ; try the update again
MyLock:
```

Implementation

The paper required coherence protocol support to implement optimistic synchronization. To detect when a conflict occurs on a monitored location, cache coherence protocols must detect when a processor is attempting to write to a memory location. In an invalidation-based protocol, the protocol simply invalidates other copies as part of obtaining exclusive permissions when a write occurs. However, the S1-AAP did not have such a protocol. It used a write update protocol in which the system sends the write to all interested caches without requesting exclusive
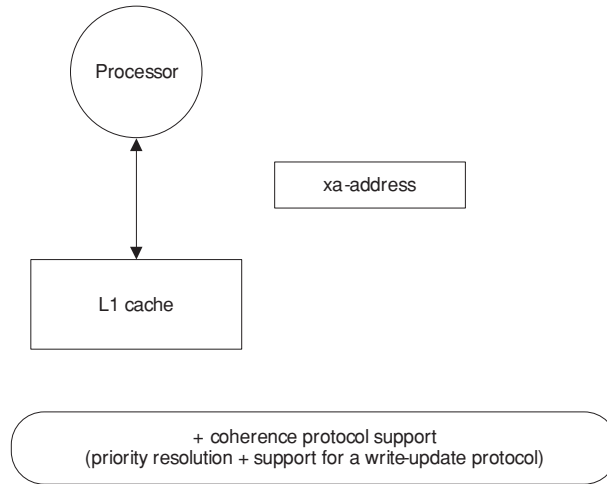
FIGURE 4.5: Hardware organization for optimistic synchronization

permissions first. To implement optimistic synchronization, the protocol required the S1-AAP hardware to perform the following sequence:

1. When a processor executes a `sync_load` instruction to an address, it sends an explicit query to remote caches that have the address.

2. Remote caches respond with either an `ACK` response, which means the remote cache is also trying a presumed exclusive region to the same address, or a `NACK` response. When the processor receives an `ACK` response, its subsequent `sync_store` instruction fails since another processor also required exclusive access. A regular store to the same address from other processors forces the `sync_store` instruction to fail.

The S1-AAP optimistic synchronization implementation (abstractly shown in Figure 4.5) performed conflict resolution in hardware. The paper describes two policies. In one, the requestor that receives an `ACK` response fails in its attempt and retries. In the other, an arbitrary conflict resolution (such as the processor identifier) is used to ensure that one processor always gets exclusive access. The S1-AAP used the latter conflict resolution. While this prevents livelock, static conflict resolution introduces the possibility of starvation.

The paper influenced synchronization mechanisms in numerous commercial microprocessors. Commercial microprocessor instruction sets, including MIPS [12, 26], Alpha [12], and PowerPC [1], implemented variants of the optimistic synchronization proposed in this paper. Sometimes it was the sole synchronization primitive. Optimistic synchronization was different

from a `conditional_store` in one significant aspect. The instruction `conditional_store` guaranteed forward progress at an instruction level because it was a single instruction (this does not imply forward progress for the synchronization itself as a process may spin on a location arbitrarily long). Optimistic synchronization, on the other hand, splits the instruction, and thus places the burden of forward progress on programmer discipline and system-wide support, especially if it is the sole synchronization primitive. Doing so introduces several challenges.

While the idea of optimistic synchronization is simple, designing optimistic synchronization into an instruction set was a difficult task. For example, the Alpha Architecture Handbook [12] listed various restrictions on the use of the `load-linked/store-conditional` instructions. The manual recommended against using memory-accessing instructions between the `load-linked` and `store-conditionals`, required branches be taken immediately preceding `load-linked` and `store-conditional` instructions, and limited the number of instructions that could be executed between the `load-linked` and `store-conditional` instructions. Such restrictions seriously limited the general applicability of these primitives.

These difficulties arise because the behavior of these new instructions was also dependent on their interactions with other events happening in the system and processor. For example, the `store-conditional` depends on the `load-linked` and events that occurred since the `load-linked` instruction's execution. This kind of behavior is different from conventional instructions, such as an `add` instruction, in which the behavior is specified in terms of the instruction's fixed known inputs when the instruction executes. The `add` instruction output does not change if a timer tick or an invalidation occurs. The above difference points to the challenge in design of such an instruction set extension. Some architecture specifications of optimistic synchronization, such as the PowerPC [1], only specify legal behaviors and provide instruction sequences guaranteed to work, but leave the results of all other instruction sequences as undefined.

Many hardware proposals for transactional memory propose similar new instructions. These instructions also face similar specification challenges. Unfortunately, these proposals do not consider how these instructions behave in the presence of existing events in the processor, and interactions with other instructions. It is not clear whether these new instruction proposals will end up with restricted behaviors that limit their broad usability.

Some optimistic synchronization implementations would also require broader system support. The SGI Origin 2000 [31], which used MIPS processors, provided special support in its directory cache coherence protocol to ensure that a failing store conditional does not cause other executing store-conditional operations to fail spuriously.

Modern implementations of complex instruction set architectures such as x86, however, implement a single `compare&swap` instruction as a series of micro-operations to make them suitable for out-of-order pipelines.

JENSEN ET AL. 1987	
Strong or Weak Isolation	Strong
Transaction Granularity	Cache line
Direct or Deferred Update	Conditional direct store (single word)
Concurrency Control	Optimistic (single word)
Conflict Detection	N/A for a single word
Inconsistent Reads	None
Conflict Resolution	Processor id or first to request ownership
Nested Transaction	N/A

4.3.4   Stone et al., IEEE Concurrency 1993

Overview

This paper describes the Oklahoma Update protocol, a hardware proposal to implement atomic read–modify–write operations on a bounded number of memory locations [47]. It proposed an alternative to critical sections and was aimed at simplifying the construction of concurrent and nonblocking code sequences for managing shared data structures (such as queues and linked lists) and atomic updates on multiple shared variables.

The proposal augmented a processor with new instructions and special registers, called reservation registers. The new instructions read and wrote these reservation registers. These registers specified addresses and buffered updates to these addresses. The processor used the cache coherence protocol to monitor updates to these addresses from other processors. When a processor performed a write as part of the Oklahoma Update, it did not immediately request write permission for the address and buffered the updates locally in the reservation registers. Only when it was ready to commit, the processor would use the coherence protocol to arbitrate for write permissions for the addresses in the reservation registers. On a successful arbitration, the processor would commit its updates to memory.

The Oklahoma Update protocol was an extension of Jensen et al. (Section 4.3.3) to multiple memory locations. The paper introduced hardware features to improve performance, such as restarting a failed update early, exponentially backing off when a conflict occurs, and providing forward progress through address-based conflict resolution.

Programming Interface

The proposal introduced three new instructions: `read-and-reserve`, `store-contingent`, `write-if-reserved`:

1. `read-and-reserve`. This instruction reads a memory location into a specified general-purpose register, places a *reservation* on the location's address in the reservation register, and clears the reservation register's data field.

2. `store-contingent`. This instruction locally updates the reservation register's data field without obtaining write permissions.

3. `write-if-reserved`. This instruction specifies a set of reservation registers and updates the memory locations reserved by those registers. It is used to initiate the commit process. It attempts to obtain exclusive ownership for each of the addresses in the reservation registers. If the reservations remain valid during this process, the instruction updates memory with the modified data from the reservation registers. The instruction returns an indication whether the update succeeded or not.

The following code demonstrates the use of these new instructions to insert an item into a linked list queue (from [47]):

```
// Usage of new instructions to construct data structure
// Memory[] contains the linked list queue
// head and tail are pointers to the head and tail of the list
// reservation1 and reservation2 signify the hardware reservations

void Enqueue(newpointer) {
   Memory[newpointer].next = NULL;
   status = 0;

   while (!status) {
      last_pointer = Read_and_Reserve(Memory[tail].next, reservation1);

      if (last_pointer == NULL) {
          // this is an empty queue
          first_pointer =
             Read_and_Reserve(Memory[head].next, reservation2);

         Store_Contingent(newpointer, reservation1);
```

```
        Store_Contingent(newpointer, reservation2);
        status = Write_If_Reserved(reservation1, reservation2);
    }


    else {
        // non-empty queue
        temp_pointer =
            Read_and_Reserve(Memory[last_pointer].next, reservation2);
        Store_Contingent(newpointer, reservation1);
        Store_Contingent(newpointer, reservation2);
        status = Write_If_Reserved(reservation1,  reservation2);
        }
    } // repeat until successful
    return;
}
```

Implementation
Fig. 4.6 shows the hardware organization necessary for the Oklahoma Update protocol.
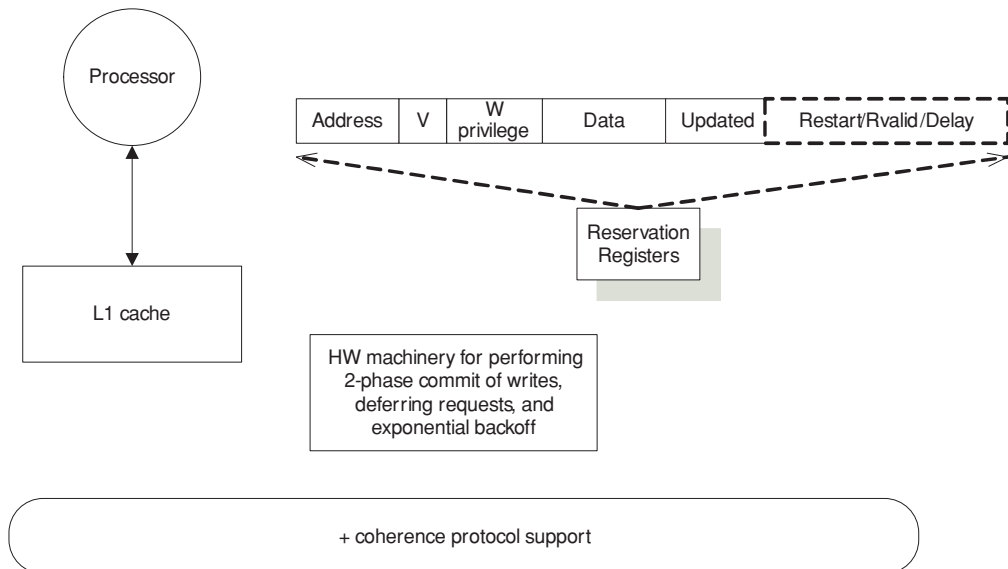The proposal assumes a standard cache-coherent shared-memory multiprocessors and adds

FIGURE 4.6: Oklahoma Update protocol support

reservation registers and hardware support for two-phase commits. The hardware optimistically executes the transaction assuming no conflicts.

The reservation registers have five primary fields. The `address` field records the address of the memory location. The `V` field records whether the reservation is valid. The `W` privilege field in the reservation registers is set if the processor already has permissions to write the address before executing the `Read-and-Reserve` instruction or received the permissions to write the address during its execution. The `Updated` field records whether the `Data` field has received the modified value of the location. The remaining fields are for performance. The `Restart/Rvalid/Delay` fields in the reservation registers provide information to the hardware for implementing an early restart capability and exponential backoff.

The Oklahoma Update protocol divided the implementation of the `write-if-reserved` instruction into two phases. In the first phase, the processor requests write permission for locations in reservation registers that did not have these permissions. Deadlocks may arise during the permissions acquisition phase. To avoid deadlocks, the hardware obtains write permissions in ascending order of address [11]. If the incoming request address is larger than the least reserved address for which the processor does not have write permissions, the processor releases its reservation. If the incoming request address is smaller than the least reserved address for which the processor does not have write permission, the processor defers the request in a buffer and services it later. This prevented livelock but did not provide starvation freedom or wait freedom.

Once all permissions have been obtained, the second phase starts and the processor commits the data values. During this phase, the processor does not abort and is uninterruptible. The two-phase implementation was similar to two-phase locking from database transaction-processing systems [4].

The Oklahoma Update implemented its conflict resolution policy in hardware. The processor deferred external requests for addresses in the reservation registers and serviced these external requests on commits and aborts, servicing the first request to a given address and then forcing a retry of the remaining requestors queued to the same address.

To improve performance, the processor implemented eager restart where, instead of waiting until the end to detect a conflict, execution would abort and restart when a reservation was lost due to a data conflict.

Interactions between the new instructions and regular store operations introduce forward-progress concerns. Regular stores do not participate in the new instructions' conflict resolution mechanism. If a regular store from one processor conflicted with an address specified in a reservation register of another processor, this processor would abort its update. This processor might have a forward-progress problem, as the regular store does not participate in the conflict resolution of the Oklahoma Update. While programmer convention can sometimes eliminate

stores to the same address, it is more difficult to eliminate false sharing. For example, a regular store may reference the same cache line as an address in a reservation register. Since cache coherence (and thus conflicts) is resolved at line granularity, the reservations would be lost. The authors propose implementing exponential backoff in hardware to ensure forward progress in such situations.

STONE ET AL. 1993	
Strong or Weak Isolation	Strong
Transaction Granularity	Cache line
Direct or Deferred Update	Deferred (in reservation registers)
Concurrency Control	Optimistic Commit initiates acquiring ownership
Conflict Detection	Late write–write conflict (if not a regular store) Late write–read conflict (if not a regular store)
Inconsistent Reads	None
Conflict Resolution	Address-based two-phase commit
Nested Transaction	N/A

4.3.5   Herlihy and Moss, ISCA 1993

Overview

This paper describes Transactional Memory, a hardware proposal to implement atomic read–modify–write operations that operate on a bounded number of arbitrary memory locations [23]. The motivation was similar to that of the Oklahoma Update proposal (Section 4.3.4), to develop lock-free data structures that avoid performance degradation and priority, inversion, deadlocks, and convoying in locking.

The proposal augmented a processor with new instructions and a transactional cache to monitor and buffer transactional data. The new instructions operated on the transactional data. The transactional cache buffered transactional state until a transaction committed and an ownership-based cache coherence protocol detected memory access conflicts among concurrently executing processors. The transactional memory proposal was for short-lived instruction sequences that accessed a relatively small number of memory locations. The footprint of the transaction could be limited to an architecturally specified number. For transactions that overflow the local cache space, the authors sketched a scheme for using software support to handle

overflow. The programmer was responsible for ensuring forward progress through software mechanisms.

This paper coined the term *transactional memory*, and identified the use of cache mechanisms for performing optimistic synchronization to multiple memory locations in a shared-memory multiprocessor. This was in contrast to the Oklahoma Update proposal that used reservation registers instead of a transactional cache.

Programming Interface

The proposal introduced six new instructions: `load-transactional`, `load-transactional-exclusive`, `store-transactional`, `commit`, `abort`, and `validate`. The programmer used these instructions for lock-free data structures and was responsible for saving register state and for ensuring forward progress. Transactions were expected to be short lived and complete in one scheduling quantum.

A transaction that experiences a data conflict does not abort immediately and may continue to execute. To prevent accessing nonserializable data (Section 2.3.5, "Detecting and Tolerating Conflicts"), a transaction must regularly use the `validate` instruction to check if a conflict had occurred. The authors considered using an asynchronous abort mechanism that traps to an abort handler before a transaction reads any inconsistent data. However, at the time designers deemed this overhead expensive [22]:

1. `load-transactional`. This instruction reads a value into a private register.
2. `load-transactional-exclusive`. This instruction reads a value into a private register and generates an ownership request if it misses the cache.
3. `store-transactional`. This instruction writes a value from a private register into a memory location in the cache but does not make the value visible to others until the transaction successfully commits. It generates an ownership request as appropriate.
4. `commit`. This instruction attempts to make memory changes permanent. The instruction succeeds only if no other transaction conflicted. The instruction returns a success or failure condition. Instruction failure discards all tentative memory changes. The commit operation does not require communication with other processors nor does it require writing data to memory.
5. `abort`. This instruction discards all updates in the write set.
6. `validate`. This instruction tests the current executing transaction's status. A return value of `true` implies that the transaction has not yet aborted. A return value of `false`

implies that the transaction aborted (from the hardware's perspective) and discards the write set.

The following code sequence demonstrates the use of the new instructions to insert an element into a doubly linked list (taken from [23]):

```
// Usage of new instructions to construct data structure
typedef struct list_elem {
      struct list_elem *next;
      struct list_elem *prev;
      int value;
} entry;


entry *Head, *Tail;


void Enqueue(entry* new) {
   entry *old_tail;
   unsigned backoff = BACKOFF_MIN;
   unsigned wait;
   new->next = new->prev = NULL;
   while (TRUE){
        old_tail = (entry*) LOAD_TRANSACTIONAL_EXCLUSIVE(&Tail);
                        // load pointer transactionally


        if (VALIDATE()) { // ensure transaction still valid
        STORE_TRANSACTIONAL(&new->prev, old_tail);
        // store pointer transactionally
        if (old_tail == NULL){


        STORE_TRANSACTIONAL(&Head, new);
        // store pointer transactionally
   }
        else {
        STORE_TRANSACTIONAL(&old_tail->next, new);
        // store pointer transactionally
   }
```

```
        STORE_TRANSACTIONAL(&Tail, new);
        // store pointer transactionally
        if (COMMIT()) // try to commit
        return;
    }


        wait = random() % (01 << backoff);
        while (wait--);
        if (backoff < BACKOFF_MAX)
        backoff++;
    }
}
```

Implementation

Fig. 4.7 shows the hardware organization for transactional memory. The hardware optimistically executes the transaction assuming no conflicts. The processor is augmented with two flags. The `tactive` flag signals whether the transaction is in progress. The flag is implicitly set when the first transactional memory operation executes and marks the beginning of a transaction; no explicit start instruction is used. The `tstatus` flag indicates whether the transaction in progress is active or aborted. For example, if another processor invalidates a transactionally accessed line, then the `tstatus` flag is set. This flag has meaning only if the `tactive` flag is set.
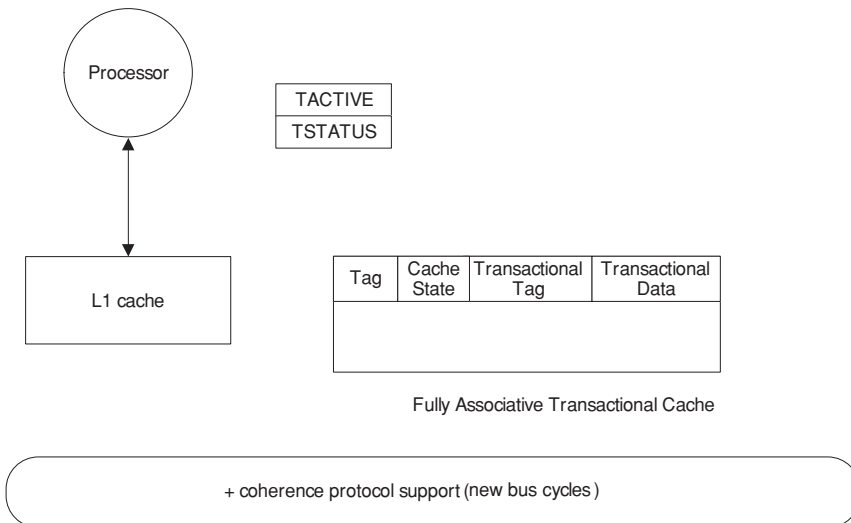

FIGURE 4.7: Herlihy and Moss Ttransactional Mmemory support

Events such as interrupts and cache overflows abort a transaction by setting the appropriate flag. Transactional operations performed while a transaction is aborted do not cause any bus traffic but may return arbitrary values.

Each processor has two caches: the regular cache to service normal loads and stores, and a transactional cache to buffer transactional memory updates and monitor accesses. This separation was to avoid affecting the performance of regular loads and stores. The *regular cache* is a direct-mapped cache with four coherence states: INVALID, SHARED (and not modified), DIRTY (not shared and is modified), and RESERVED (not shared and not modified, but valid).

The *transactional cache* is a fully associative cache that holds all transactional writes without propagating their values to other processors or to main memory until the transaction commits. The transactional cache has additional tags with each line that add special meaning to the regular cache states. If tag is empty, the line has no data. If tag is normal, the line has committed data. An xcommit tag means the contents must be discarded on commit, and an xabort tag means the contents must be discarded on an abort.

The cache coherence protocol is augmented by three new bus cycles. The t_read bus cycle is for a transactional read request that goes across the bus. This request can be refused (NACK) by a busy cycle. The t_rfo bus cycle is for a transactional read-for-exclusive request that goes across the bus. This can be refused (NACK) by a busy cycle. The busy bus cycle prevents too many transactions from aborting one another too often. This approach may starve some transactions but a queuing mechanism can address starvation. A busy response does not cause the transaction execution itself to abort immediately but records hardware state to allow the transaction to check for whether the transaction has aborted from the hardware's perspective. Until this check, the transaction may continue to execute without aborting.

The allocation policy in the transactional cache is changed such that first an empty line is replaced, followed by normal, and then xcommit. The cache writes back the data first if the line was dirty and had the xcommit tag. We step through a load-transactional instruction example (the paper discusses others scenarios). The processor probes the transactional cache for an address-matching entry with the xabort tag, and returns its value if there is one. If a normal tag entry exists, then the tag changes to xabort and the cache allocates a second entry with the xcommit tag and the same data. Otherwise, the processor issues a bus request for the data. When the data returns, the transactional cache allocates two entries, one with the xcommit tag and another with the xabort tag. A busy response from the bus results in transaction abort. This involves setting tstatus to false, invalidating all xabort tag entries, and setting all xcommit tag entries to normal tag.

To allow transaction state to spill over from the transactional cache, the authors suggest the use of a LimitLESS directory scheme [9], in which software can emulate a large directory. This allows using hardware for the common case and software for the exceptional case.

HERLIHY AND MOSS 1993	
Strong or Weak Isolation	Strong
Transaction Granularity	Cache line
Direct or Deferred Update	Deferred (in cache)
Concurrency Control	Optimistic
Conflict Detection	Early
Inconsistent Reads	Yes
Conflict Resolution	Receiver NACKs/Requestor software backoff
Nested Transaction	N/A

4.3.6   Rajwar and Goodman, MICRO 2001

Overview

The paper describes speculative lock elision (SLE), a hardware mechanism that permits programmers to use frequent and conservative lock synchronization to write correct multithreaded code easily but achieves the same performance as well-tuned synchronization [38]. SLE improves the performance and programmability of lock-based multithreaded applications while retaining the lock-based programming model.

SLE used hardware support for optimistic execution of critical sections (including the use of register recovery mechanisms), to convert a lock variable dynamically, and in a binary transparent manner, from an active serialization mechanism to a passive one invoked only when it was required. The key observation in SLE was that a processor did not have to acquire (modify the value of) a lock, but needed only to monitor it for correct execution. The processor predicts a lock as unnecessary and elides the lock acquire and release operations.

The processor executes the critical section optimistically as if it is lock-free. This makes the program behave as if locks were not present. If the memory operations between the lock acquire and release occur atomically, then the processor can elide the two writes corresponding to acquire and release. This is because the second write (lock release) undoes the changes of the first write (lock acquire). By doing so, the lock remains free and critical sections can execute concurrently. The coherence protocol detects data conflicts. On a data conflict, execution aborts, register state is recovered and execution restarts. The processor retries the critical section on a conflict, and after some threshold acquires the lock. This allows the proposal to provide the same forward-progress guarantees as the underlying locking algorithm used by the program. On repeated data conflicts, the processor explicitly acquires the lock. If the execution exceeded hardware resources and speculation could not continue, the processor acquires the lock and

ends speculation. SLE did not require identification of locks since hardware used prediction. This allowed SLE to be programming model and binary compatible without requiring new semantic instructions. However, it required the program to have a lock pattern SLE could predict.

SLE demonstrated how to execute and commit lock-based program execution without acquiring locks or requiring write permissions to them. Two threads contending for the same lock could execute and commit in parallel without communication if their execution was data independent. SLE treated all operations within the critical section as transactional unlike prior proposals where special instructions identified transactional locations.

Programming Interface

SLE assumes that programmers use lock patterns such as `test&test&set` and `test&set`. This allows the hardware to infer them. Beyond this, SLE does not require software support. SLE treated existing atomic read–modify–write operations such as a `load-linked/store-conditional` pair or a `compare&swap` instruction as possible lock acquires for critical sections, and does not require new instructions. The C language sequence below shows a hash table protected by a lock. The Alpha assembly language sequence below shows the lock acquire and release operations using the `ldl_l` (load-linked) and `stl_c` (store-conditional) instructions. A regular store operation releases the lock:

```
LOCK (hash_tbl.lock)         LOCK(hash_tbl.lock)
var = hash_tbl.lookup(X);    var = hash_tbl.lookup(Y);
if (!var)                    if (!var)
   hash_tbl.add(X);             hash_tbl->add(Y);
UNLOCK (hash_tbl.lock)       UNLOCK(hash_tbl.lock)


L1: ldl      t0,      0(t1)   # t0 = hash_tbl.lock
    bne      t0,      L1:     # if not free, goto L1
    ldl_l    t0,      0(t1)   # load locked, t0 = lock
    bne      t0,      L1:     # if not free, goto L1
    lda      t0,      1 (0)   # t0 = 1
    stl_c    t0,      0 (t1)  # conditional store, lock = 1
    beq      t0,      L1:     # if stl_c failed, goto L1

... <hash table access critical section> ...

    stl 0, 0 (t1)                # lock = 0, release lock
```
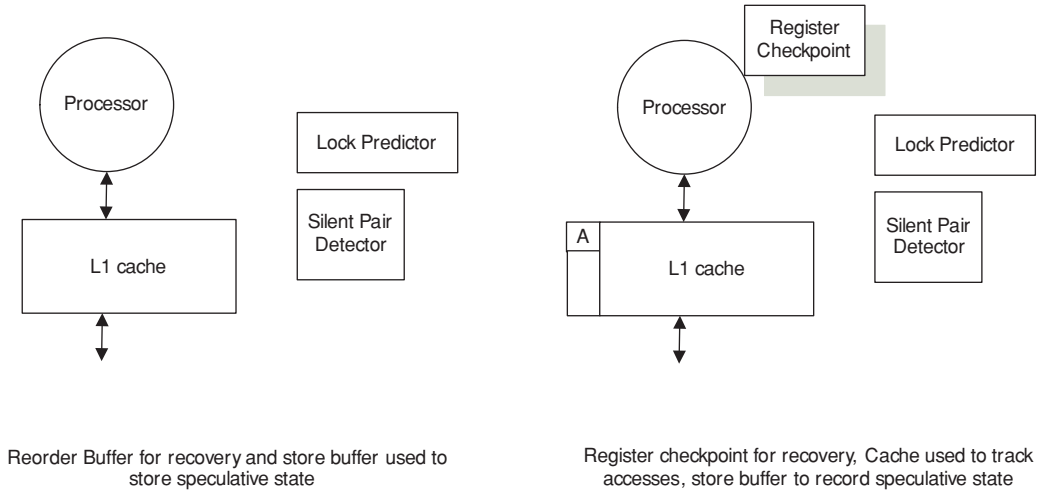
Reorder Buffer for recovery and store buffer used to store speculative state

Register checkpoint for recovery, Cache used to track accesses, store buffer to record speculative state

FIGURE 4.8: Speculative lock elision microarchitecture organization

Implementation

Fig. 4.8 shows the hardware organization for SLE. The hardware predicts a critical section, records register recovery state, tracks memory accesses and buffers updates, and performs the lock elision.

SLE uses one of two techniques to maintain register recovery state. In the first approach, it executes using the existing reorder buffer. This allows SLE to use existing recovery mechanisms for branch mispredictions, but limits the critical sections to those that fit in the reorder buffer. The second approach allows critical section size to be larger than the reorder buffer. It uses either a flash copy of the architectural register file or a checkpoint of the alias table.

SLE associates an `access` bit with each cache line to track addresses accessed during optimistic execution. These bits interact with the cache coherence protocol to detect data conflicts. SLE used a merging store buffer to record speculative data. This data was not exposed to the other processors until after commit. For commit, a processor requires all cache lines accessed in the optimistic critical section to be in the cache. Then, the write buffer drains the buffered writes into the cache. During this duration, the processor stalls snoops from other processors to the cache to prevent them from observing stale data until commit completes.

The processor starts SLE by predicting that the read of an atomic read–modify–write operation is part of the lock acquire of a critical section. A predictor and confidence estimator determines candidates for lock elision by inspecting the synchronization instructions used to construct locks. The processor issues this read operation as a regular load, and records the load's address and data value it returned. The processor also records the data of the write of the read–modify–write operation but does not make this write operation visible to other processors or

request write permissions for it. This has two effects. First, it leaves the lock variable in a free state. Since the processor did not write the value to the cache, the cache line remains in shared state. This allows other processors to see the lock as free. Second, the write value allows detection of a lock release and elision by a value and address comparison. This provides program order semantics: subsequent accesses to the lock variable from the processor performing the elision will return the last written value in program order. The sequence below demonstrates this. External threads see the lock variable as free while the speculating thread sees it as HELD:

```
Program Semantic     Instruction Stream            Value of hash_tbl.lock
                                                  as seen by    as seen by
                                                    self           others


TEST _lock_       L1: i1 ldl t0, 0(t1)              FREE          FREE
                      i2 bne t0, L1:
TEST _lock            i3 ldl_t t0, 0(t1)            FREE          FREE
   &                  i4 bne t0, L1:
SET _lock_            i5 lda t0, 1 (0)
                      i6 stl_c t0, 0 (t1)            HELD          FREE
                      i7 beq t0, L1:


     ... <hash table access critical section> ...
RELEASE _lock_        i56 stl 0, 0 (t1)             FREE          FREE
```

The data cache or the load and store buffers track memory accesses. If another processor makes a conflicting request to a line that has been speculatively accessed, the processor aborts execution and acquires the lock (possibly after a retry).

During optimistic execution, the processor inspects stores to determine if they write to the same address as the earlier write of the atomic read–modify–write operation and write the same value as the read of the atomic read–modify–write operation. If an address and value match occurs, the processor has identified a lock release operation. Execution commits once all earlier speculatively written lines are ordered. If the value does not match but the address matches, the processor makes the earlier buffered store value of the lock acquire visible to the coherence protocol, followed by this store. Since the algorithm relies on observed value- and address-patterns and does not change program semantics, it does not need to identify a lock accurately.

If optimistic execution cannot continue (e.g., due to lack of cache resources or IO operations), the buffered write data of the atomic read–modify–write is made visible to the coherence protocol without triggering a misspeculation. If the coherence protocol orders this request without any intervening data conflicts to either the lock or the speculatively accessed data, then the execution is committed. Here, the execution transitions from a lock elision mode into an acquired lock mode without triggering a misspeculation.

SLE did not provide transactional memory semantics all the time because in the presence of conflicts and resource constraints, execution reverts to the original lock-based execution. SLE was beneficial only if programmers employed the assumed lock patterns.

RAJWAR AND GOODMAN 2001	
Strong or Weak Isolation	Strong if hardware resources sufficient
Transaction Granularity	Cache line
Direct or Deferred Update	Deferred (in cache)
Concurrency Control	Optimistic
Conflict Detection	Early
Inconsistent Reads	No
Conflict Resolution	Receiver aborts/Heuristically acquires lock
Nested Transaction	Flattened

4.3.7   Rajwar and Goodman, ASPLOS 2002

Overview

This paper described transactional lock removal (TLR), an extension to SLE (Section 4.3.6), that uses timestamp-based fair conflict resolution to provide transactional semantics and starvation freedom [39]. In SLE, the processor acquired the lock if repeated conflicts occurred. In TLR, the lock is acquired only due to resource constraints, thus achieving lock-free execution in the presence of data conflicts. TLR maps a critical section to a transaction and treats all operations inside the critical section as implicitly transactional. It converts lock-based critical sections transparently and dynamically to lock-free optimistic transactions and enables the concept of a transaction to be reflected in common multithreaded programs while allowing the continued use of the familiar lock-based critical section paradigm.

TLR used hardware support in the cache coherence protocol and cache controllers for conflict resolution. TLR employed a fair timestamp-based conflict resolution using Lamport's *logical clock* construction [29] and Rosenkrantz et al.'s *wound–wait* algorithm [41]. In the
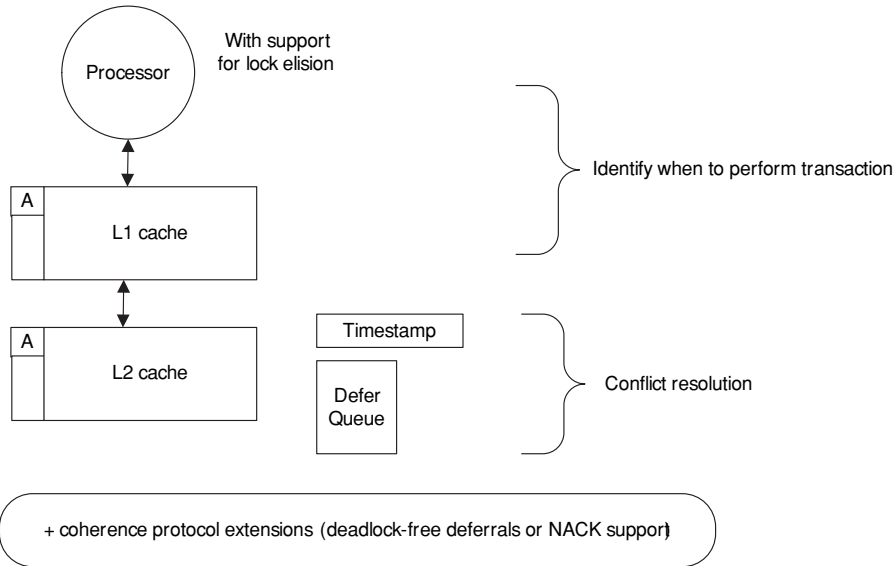
FIGURE 4.9: Transactional lock removal system organization

algorithm, a transaction with higher priority never waits for a transaction with lower prior-ity. A conflict forces the lower priority transaction to restart or wait. The algorithm provided starvation freedom.

Programming Interface
The programmer interface is the same as SLE (Section 4.3.6).

Implementation
TLR uses hardware to perform conflict resolution in the presence of data conflicts, and without having to fall back on the lock. Fig. 4.9 shows the hardware organization. The new hardware required is the addition of a timestamp and support to defer incoming requests based on times-tamps. TLR can ensure atomic execution of the critical section by obtaining all required cache lines accessed within the transaction in an appropriate ownership state, retaining such ownership until the end of the transaction, executing the sequence of instructions forming the transaction, speculatively operating on the cache lines if necessary, and making all updates visible atomically to other processors at the end of the transaction.

TLR uses the existing cache coherence protocol to implement the conflict resolution algorithm. The algorithm is similar to that proposed by Rosenkrantz et al. [41]. In the algorithm, a transaction with higher priority never waits for a transaction with lower priority. A conflict

forced the lower priority transaction to restart or wait. The priorities in the TLR algorithm are based on Lamport's logical clock construction.

A timestamp has two components: a local logical clock and a processor identifier. The local logical clock assigns a number to a successful TLR execution and captures the logical time at which the execution occurred. The local logical clock is incremented by 1 or higher value on a successful TLR execution. A processor identifier appended to the logical clock breaks ties among locally generated clocks. This forms a globally unique but locally generated timestamp. All requests from within a transaction on a processor are assigned the same timestamp. Drift among various processors is resolved by ensuring that on a commit, the processor sets its local logical clock to a value larger than an incoming conflicting request received. A processor retains and reuses its timestamp following a misspeculation. This allows the processor to retain its position, eventually have the lowest timestamp in the system, and avoid starvation.

The conflict resolution algorithm allows a higher priority transaction to retain ownership of its cache lines when it conflicts with a lower priority transaction. TLR uses coherence protocols to retain cache-line ownership either by using NACKs or by deferring requests. With NACK-based techniques, a processor does not process an incoming request but sends a negative acknowledgement to the requestor. The requestor then retries at a future time. This requires the baseline coherence protocol to support NACKs. With deferral-based techniques, a processor defers processing an incoming request by buffering the request and masking any conflict. However, the protocol transition has indeed occurred from the perspective of the coherence protocol. The paper discusses a deferral-based scheme because it does not require a special coherence protocol substrate (such as support for NACKs). Such deferrals support implicit construction of coherence requests to allow high-performance data transfers.

With deferrals, the conflict-winning processor with an exclusively owned cache line delays processing the incoming request for a bounded time, preferably until the processor has completed its transaction. The coherence state transitions as seen by the "outside world" are assumed to have occurred but the processor does not locally apply the incoming request. Deferrals introduce the possibility of deadlock. TLR uses new messages, called marker messages, to ensure deadlock-free execution. These messages do not affect the coherence state transitions and only implement the deadlock-free conflict resolution. The marker messages are required only when the processor is doing TLR and receives a conflicting request for an exclusively owned line.

Deadlock is not possible if only one cache line is under conflict within the transaction because a cyclic wait is impossible; the head node of the coherence chain is always in a stable coherence state. If the processor requests only a single cache line on which it depends, then the processor can ignore timestamp-based ordering. If an additional cache line is accessed, then

deadlock may occur if the access generates a cache miss. Here, the timestamp-based order must be enforced.

TLR also performs another optimization for coherence protocols that do not require explicit acknowledgement of invalidations when a shared cache line is written. Typically, in a critical section, a processor reads a shared location, operates on the value, and writes a new value to the same location, thus resulting in an upgrade request. TLR uses an instruction-based predictor to identify such situations where the processor should issue a read for ownership request itself as part of the original read of the shared location. The processor completes the sequence in a single memory request instead of the typical two. This obtains a similar effect to Herlihy and Moss's `load-transactional-exclusive` instruction (Section 4.3.5).

RAJWAR AND GOODMAN 2002	
Strong or Weak Isolation	Strong if hardware resources sufficient
Transaction Granularity	Cache line
Direct or Deferred Update	Deferred (in cache)
Concurrency Control	Optimistic
Conflict Detection	Early
Inconsistent Reads	No
Conflict Resolution	Timestamps (Lamport clocks) on conflicts
Nested Transaction	Flattened

4.4   BOUNDED/LARGE HTMS

The remaining papers in this chapter allow a transaction to exceed the footprint of the level one data cache but do not support context switches and thread migration during a transaction. We refer to these proposals as bounded/large HTMs. The proposals, however, significantly differ from each other in their implementations and characteristics.

Hammond et al. use deep write buffers and a two-level cache to track transactional state (Section 4.4.1). Ananian et al. spill transactional state into a local uncached memory table (Section 4.4.2). Moore et al. allow transactional state to escape out of the local cache and into the memory system, and use coherence protocol extensions to perform conflict detection (Section 4.4.3). Ceze et al. do not describe a TM system but describe a new implementation of HTM mechanisms that does not use the cache coherence protocol to perform conflict detection (Section 4.4.4). It uses signatures to capture read and write sets, including efficient tracking of nested transactions.

These papers used hardware mechanisms (caches, cache coherence mechanisms, and signatures) to track transactional reads and writes, to detect conflicts among transactions, and to commit transactions. Of these papers, only Moore et al. maintained undo information in a software-accessible log (hardware wrote this log directly) of and used software to perform recovery following an abort. This also gave software visibility into aborted state of a transaction because undo was not automatic, thus exposing this typically hidden HTM transaction state. The other papers abort and automatically undo using hardware. Since these papers maintain state in hardware structures tightly linked to the processor on which the transaction was executing, they do not support descheduling or moving an active transaction from one processor to another without aborting.

4.4.1   Hammond et al., ISCA 2004
Overview

This paper describes transactional coherence and consistency (TCC), a shared-memory model in which all operations execute inside transactions. A transaction is the basic unit of work, interprocessor communication, cache coherence, and memory consistency [20]. TCC was an alternative to conventional synchronization to simplify parallel software development.

In TCC, the programmer or compiler divides a program into a series of transactions. These transactions can have dependences through memory. For an explicitly parallel program, the programmer has to identify the transaction boundaries. The TCC hardware then executes these transactions in parallel.

TCC did not implement a cache-line ownership protocol. Instead, TCC broadcasts memory changes at transaction boundaries. A processor executes a transaction speculatively without requesting cache-line ownership and buffers updates locally. Multiple transactions can write the same locally cached memory location concurrently. When a transaction is ready to commit, its processor arbitrates for a global token. This token determines which transaction commits; only one transaction can commit system-wide at a time. Once the processor obtains the token, its transaction commits and it broadcasts its writes to all other processors. All other processors compare the incoming writes to their own transaction read and write sets. If a match occurs, a processor aborts and reexecutes its transaction. Interprocessor communication takes place at transaction boundaries only. If the transaction exceeds local cache buffers, then the transaction enters a nonspeculative mode. Here, the processor executing the transaction acquires the global commit token to prevent any other processor in the system from committing. The transaction executes nonspeculatively directly updating memory.

TCC unified two known techniques: speculative parallelization of sequential programs (`ordered transactions`) and optimistic synchronization in parallel programs (`unordered`

`transactions`). Knight in his paper (Section 4.3.2) outlined a similar approach for speculative parallelization but the paper did not explicitly connect his approach to optimistic synchronization in explicitly parallel programs.

Programming Interface

The TCC programming model divides all programs into a series of transactions. A transaction may be a single instruction. To divide a sequential program, these transactions can be determined heuristically. To divide an explicitly parallel program, the programmer must identify these transactions correctly so as not to break any atomicity or mutual exclusion requirements in the program. The programmer can specify the order in which various transactions commit; a useful mechanism in speculative parallelization.

Implementation

TCC uses hardware to recover register state, track read and write sets, and to buffer transactional updates until commit. It also performs commits and aborts in hardware. Fig. 4.10 shows the hardware organization.
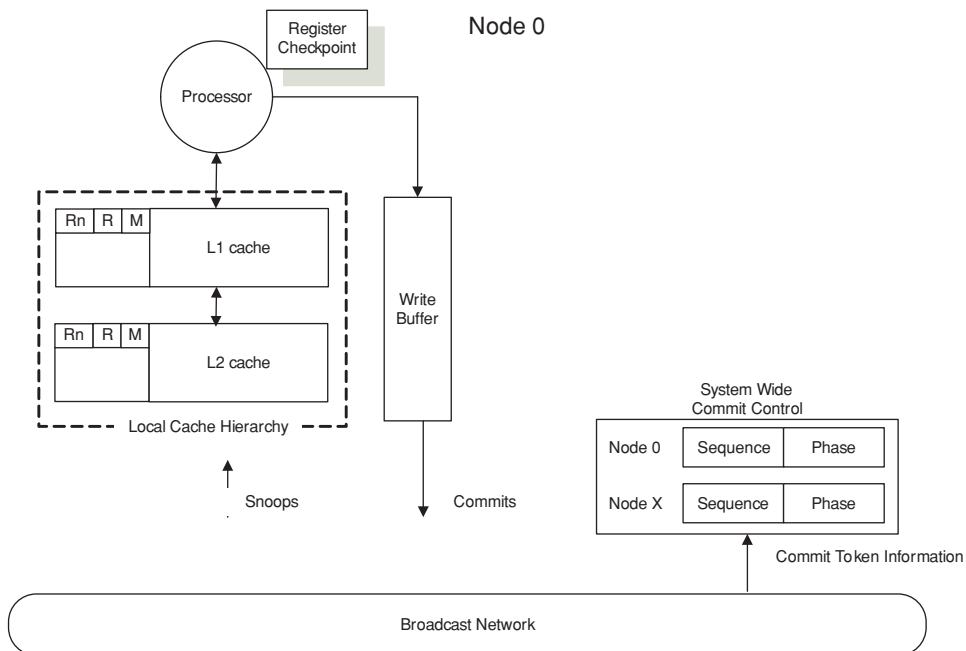

FIGURE 4.10: TCC hardware organization

The paper describes various schemes to manage register state: a shadow checkpoint of the entire register file, a checkpoint of only the register rename table, or software support to save register state. TCC extends hardware caches with `rename` bits (Rn), `read` bits (R) and `modified` (M) bits. The write buffer stores all updates until a commit or abort. The `read` bits in the cache maintain the read set. Multiple bits per line mitigate the problem of false sharing. The `modified` bits in the cache track the write set. The `Renamed` bits, one for each word/bytes in a cache line, are set if a store writes all parts of the word/byte. Subsequent loads to such words/bytes do not set the `Read` bit because the processor previously generated the read data and therefore cannot depend on another processor.

To commit a transaction, a processor arbitrates globally for a hardware commit token and broadcasts the transaction's write set to all other processors and to memory. The hardware collects all memory updates by a transaction into a commit packet and broadcasts it to other processors. TCC uses this to order transactions without a software construct.

Similar to the transaction block order in Knight's proposal (Section 4.3.2), the commit token provides TCC with the capability to serialize execution in the system. When the processor executes an operation such as IO or exceeds local buffers, it arbitrates for and acquires the global commit token. This way the processor prevents other processors in the system from committing and interfering with this processor's execution. The processor updates memory directly and nonspeculatively. The token is released only when the processor completes and commits its transaction. Other processors then resume arbitration for the token and commit. The paper describes double buffering to prevent unnecessary stalls while a processor waits for a prior transaction to commit. Double buffering provides extra write buffers and extra set of read and modified bits for the caches.

HAMMOND ET AL. 2004	
Strong or Weak Isolation	Strong
Transaction Granularity	Cache line
Direct or Deferred Update	Deferred (in cache)
Concurrency Control	Optimistic Commit serialized globally
Conflict Detection	Late write–write conflict Late write–read conflict
Inconsistent Reads	None
Conflict Resolution	Via global commit coordination
Nested Transaction	Flattened

4.4.2   Ananian et al./LTM, HPCA 2005

Overview

This paper describes an HTM implementation called large transactional memory (LTM) that allows transactional cache lines to spill into a reserved region in local memory without aborting the transaction [3]. This allows the transaction to access data sets in excess of local cache sizes. LTM does not allow a transaction to survive context switches. This paper also describes UTM (Section 4.5.1), an unbounded transactional memory system. However, the authors viewed UTM as too complex to implement, and proposed LTM as an alternative.

LTM allocates a special uncached region in local memory to buffer transaction state that spills from the cache. This region is maintained as a hash table. Each cache set has an associated `overflow` bit. This bit is set when a transactional cache line is evicted and moved into the hash table. When a request from another processor accesses a cache set with its `overflow` bit set but does not find a match for its request's tag, a hardware state machine walks the hash table to locate the line with a matching tag.

Programming Interface

LTM assumed a simple transaction usage and provided two new instructions: xbegin <pc> and xend. The xbegin instruction specifies a user handler. This handler is executed whenever a transaction aborts. This provides programmers an ability to specify actions on aborts. A typical usage of these new instructions is shown below:

```
XBEGIN <PC of user handler>
... <transaction code> ...
XEND
```

Implementation

LTM is a hardware proposal in which all transactional state is maintained and managed by hardware. LTM can relax the UTM design requirements because LTM does not require a transaction to survive context switches and therefore does not require persistent transactional state management.

Fig. 4.11 shows the LTM hardware organization. LTM assumes an HTM system where a processor executes transactions in hardware. It uses register recovery mechanisms and uses the cache to track and buffer transactional accesses (T bit per cache line). LTM assumes that the cache coherence protocol can respond with a NACK to incoming requests from other processors.

To support spilling transactional state safely, LTM extends each set in the cache with an O bit. When the processor executing a transaction overflows the cache, the LTM hardware
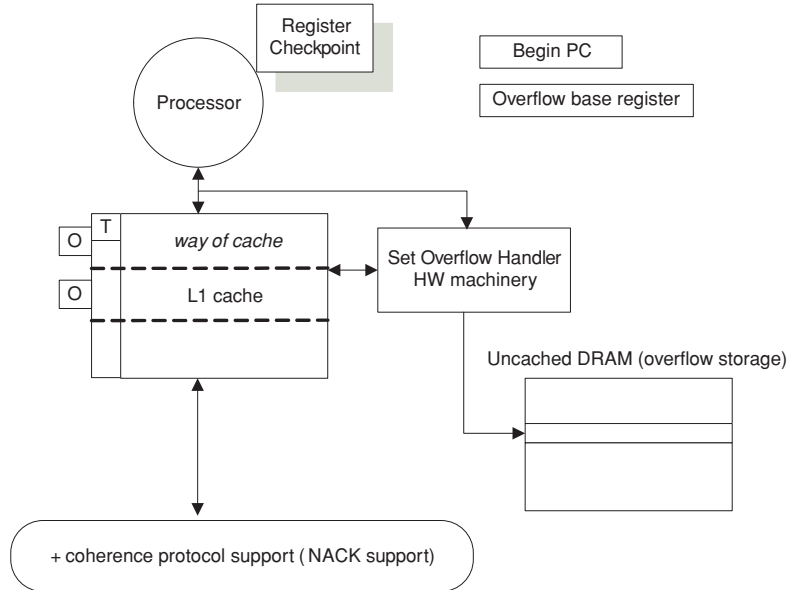
FIGURE 4.11: LTM hardware organization

moves the evicted block into a hash table in memory, and sets the `O` bit for the corresponding set in the cache. This signals an overflow for the cache set. During this process, the processor responds to incoming snoops with a negative acknowledgement (`NACK`). The processor does not service incoming snoops until it has checked both the cache and the memory hash table for a conflict.

If a processor's request hits in the cache and the `O` bit is not set, the cache treats the reference as a regular cache hit. If the processor request misses in the cache and the `O` bit is not set, the cache treats this request as a regular cache miss. If the `O` bit is set, even if the request does not match a tag in the cache, the line may be in the hash table in memory. The processor has to walk the hash table in memory. If the walk finds the tag in the hash table, the walker swaps the entry in the cache set with the entry in the hash table. Otherwise, the walker treats this as a conventional miss.

LTM implements its conflict resolution policy in hardware. If a processor executing a transaction receives a snoop request that conflicts with a transactional line (the line's `T` bit is set), then the transaction is aborted.

In LTM, any incoming snoops that hit a set with the `O` bit set but do not match the tag interrupt the processor and trigger a walk of the hash table. Consequently, two independent programs that do not share any data and are in their own virtual address spaces may interfere with each other. Because a transaction in one program can continually interrupt another transaction

in another program by virtue of simply running on the same system, the system does not offer performance isolation [50].

ANANIAN ET AL./LTM 2005	
Strong or Weak Isolation	Strong
Transaction Granularity	Cache line
Direct or Deferred Update	Deferred (in cache and in overflow memory)
Concurrency Control	Optimistic
Conflict Detection	Early
Inconsistent Reads	None
Conflict Resolution	Processor receiving conflict request aborts
Nested Transaction	Flattened

4.4.3   Moore et al., HPCA 2006

Overview

The paper describes LogTM, an HTM implementation that seamlessly overflows transactional data from the local data cache and into the rest of the memory hierarchy [35]. This allows a transaction to have footprints greater than the local cache. LogTM, like LTM (Section 4.4.2), did not allow a transaction to survive a context switch.

LogTM had two key features. First, LogTM supported eviction of transactionally accessed cache lines during a transaction by retaining ownership of the cache line. These evicted cache lines are treated as `sticky` by the cache coherence protocol; even though the lines were evicted, the evicting cache continued to be the owner. This way, transactions use existing cache coherence mechanisms to detect conflicts when a transaction overflows its cache.

Second, LogTM used a software log to record values of memory locations updated in a transaction, so they could be restored if an abort occurs. In earlier bounded HTMs, before a transactional write, processors write back the earlier copy of the line to the next level of the memory hierarchy and keep the transactionally updated line in the cache until commit. Instead of writing to lower levels of the hierarchy, LogTM copies the earlier value into a software log and allows the transactional write to propagate to the lower levels. When an abort occurs, a software handler walks the log and restores the values.

The LogTM implementation was optimized for commit operations and it assumed that most transactions commit successfully. LogTM performed transactional updates directly to memory while keeping an earlier copy of the updates in a log; it does not copy data on commits.
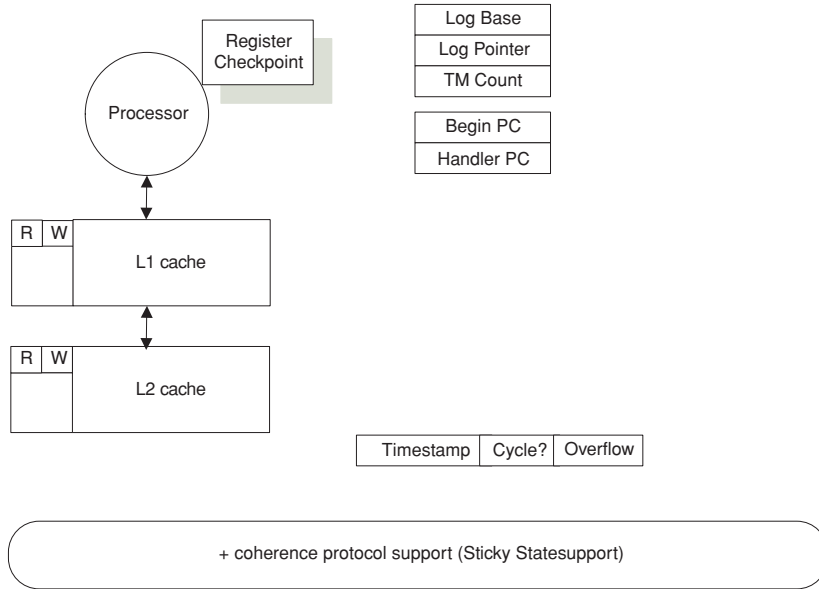
FIGURE 4.12: LogTM hardware organization

Programming Environment and Interface

LogTM assumed a simple transactional memory usage model where a thread begins, commits, and aborts a transaction. It assumes that threads are running in user mode and in a single virtual address space. Every thread is allocated its own log in virtual memory. The LogTM system software has an interface to initialize a thread's transactional state, log space, per-thread conflict handlers, and to perform rollback and aborts. The LogTM system communicates the bounds of the per-thread software log to hardware.

Implementation

In LogTM, the processor recovers register state following an abort. The hardware cache and cache coherence protocol track addresses accessed in the transaction and participate in conflict detection. Fig. 4.12 shows the LogTM hardware organization. Each cache line is extended with read (R) and write (W) bits. The R bit tracks whether this line has been read from in the transaction and the W bit tracks whether this line has been written to in the transaction.

The processor has hardware support to add entries to the executing thread's software log. When a store operation occurs inside a transaction, the LogTM hardware appends the earlier value of the cache line and its virtual address to the thread's software log. The W bit is set to filter redundant log operations. Since the log resides in software, the log's virtual address is pretranslated to a physical address for the hardware to directly add to the log. This is done using a single-entry micro-TLB. When a transaction aborts, control transfers to a software handler. This

handler then walks the software log backwards and restores memory state. When a transaction commits, the pointer to the log is reset. The log is maintained even if the transaction fits in the local cache. The paper describes various optimizations for reducing logging overhead. The hardware could buffer the updates to the log locally and not write the software log itself unless it is necessary. If a transaction commits without overflowing the cache, then LogTM never writes the software log. The write to the log can be delayed until either an abort or an overflow.

The processor has a nesting depth counter and an overflow bit. LogTM flattens nested transactions. Eviction of a transactional cache line sets the processor's overflow bit, and a commit or abort resets it.

The paper describes LogTM behavior using a directory-based MOESI cache coherence protocol. When a transactional cache line is evicted from the cache, the directory state is not updated and the cache continues to be associated with the line; it becomes `sticky`. This is unlike a conventional protocol where a writeback of a modified line results in the directory assuming ownership of the line.

Consider eviction scenarios for the `M, S, O`, and `E` cache-line states. Eviction of a trans-actionally written cache-line results in a transactional writeback instead of a conventional write-back. The directory entry state does not change and the processor remains the owner of the line. When the directory receives a request for this line, it forwards the request to the owner. If the owner receives the request for line that is not present in the cache and the owner has its over-flow bit set, then the owner signals a potential conflict by sending a negative acknowledgement (`NACK`) to the requestor. The requestor then invokes a conflict handler; it does not interrupt the processor that responded to the request.

A protocol that supports silent evictions of clean-shared cache lines works without special actions since the evicting processor will receive invalidation requests from the directory for the cache line. If the protocol does not support silent evictions, then a sequence similar to that for the `M` state ensures correct handling by preventing the directory from removing the processor from the sharing vector.

A cache line in the owned state means that the data value is not the same as in the main memory but the line is potentially shared. Here, the cache writes the line back to the directory and transitions it to an `S` state. LogTM treats a silent `E` eviction as if the line was in `M` state.

The processor must interpret incoming requests unambiguously; it can no longer simply ignore requests that do not match in the cache. This is because commit operations do not actively clean directory state; these states are cleaned lazily. Consider that processor `P` receives a request to an address not in its cache. If `P` is not in a transaction and receives a forwarded sticky state, it must be from an earlier transaction and thus is stale. If `P` is in a transaction but its overflow count is zero, it must be from an earlier transaction. In both cases, `P` responds to the directory with a `CLEAN` message.

The requestor performs conflict resolution after receiving responses from other processors. The requestor sends a request to the directory. The directory responds and possibly forwards it to one or more processors. Each processor then inspects its local cache state and responds with either an ACK (no conflict) or a NACK (a conflict). The requestor collects this information and then resolves the conflict. Instead of the requestor immediately aborting on receiving a NACK, it may reissue the request if the conflicting processor completes its transaction. However, to ensure forward progress and avoid unnecessary aborts, LogTM uses a distributed timestamp method, and invokes a software conflict handler if a possibility of a deadlock arises. Such a deadlock may arise because a transaction may be simultaneously waiting for an older transaction and may force an older transaction to wait.

The proposal does not require log or hash-table walks for detecting conflicts. As currently defined, the proposal requires the abort handler execution to be nonpreemptible.

MOORE ET AL. 2006	
Strong or Weak Isolation	Strong
Transaction Granularity	Cache line
Direct or Deferred Update	Direct (into cached memory)
Concurrency Control	Optimistic
Conflict Detection	Early
Inconsistent Reads	None
Conflict Resolution	Requestor invokes software
Nested Transaction	Flattened

4.4.4   Ceze et al., ISCA 2006

Overview

This paper describes an implementation of HTM that does not use caches or cache coherence to track transactional accesses and detect data conflicts. The new implementation, called *Bulk*, compresses address information for conflict detection into a signature and broadcasts the signature at commit time [8]. Other processors use this signature to detect conflicts. Other HTMs that detect conflicts at commit time are by Knight (Section 4.3.2), Stone et al. (Section 4.3.4), and Hammond et al. (Section 4.4.1). Bulk's signature mechanism does not require cache-line extensions to track transactional state, buffer transactional updates, and commit or discard this state. Bulk also does not require special coherence protocol support to implement late conflict detection.

Bulk represented transaction addresses into a compact signature. All transactional reads formed the *read signature* and all transactional writes formed the *write signature*. Hardware

generates, maintains, and operates on these signatures. Because of compression, the signature represents a superset of the addresses added to it. This can result in false conflicts between transactions. A transaction that is ready to commit requests arbitration for commit permission. Once it receives commit permissions, it broadcasts its write signature to other processors. Bulk supports word-level conflict detection without requiring the coherence protocol granularity to change. The paper also describes applying Bulk to speculative parallelization using transactions (similar to Sections 4.3.2 and 4.4.1).

Implementation

The paper assumes a programming interface where the hardware is informed of a begin operation and a commit operation. The begin starts transactional execution and the commit ends transactional execution. The paper focuses on the implementation that manages transactional state accessed during the transaction.

In Bulk, each transaction has a read and write signature. If transactions are nested, then each nesting level has a read and write signature. The signature has a fixed size, typically 2048 bits. Bulk introduces a new hardware component, called the Bulk disambiguation module (BDM), for managing these signatures.

When a processor performs a transactional read, the address is added to the read signature, and on a transactional write the address is added to the write signature. For adding an address, the hardware permutes the address bits and selects a subset of the permuted bits. These bits are then decoded and ORed into the signature. Compressing multiple addresses into 2048 bits of signature introduces aliasing.

Because these signatures are used to detect conflicts, the hardware supports additional operations on these signatures. For example, to test whether two signatures have any addresses in common, the hardware implements an intersection function. Similarly, a signature union combines the addresses from multiple signatures into a single signature. This is used when address sets of nested transactions have to be combined. Hardware supports testing an address for membership in a signature. This is used to allow the signature to coexist with a conventional cache coherence protocol and nontransactional read and write operations from other processors conflict with the transaction. The hardware implements a set extraction function—the signature can be expanded into a list of cache sets which the addresses map to. The exact list of addresses cannot be extracted since this information is lost during the construction of the signature. Instead, hardware uses the set list to identify a superset of addresses.

When a transaction aborts or commits, the processor has to either invalidate and discard transactional state or make state visible. To do so, it has to identify which lines in the cache are transactional. Since Bulk does not record this information in the cache lines, this information is determined from the signatures. The read signature determines which addresses have been

transactionally read and the write signature determines which addresses have been transactionally written. Note that the addresses determined from the signature are a superset of the actual addresses accessed in the transaction. Determining the addresses from a signature is a two-step process. First, the hardware extracts the cache sets corresponding to all the addresses in the signature. Then a membership function is applied to each valid address in the selected cache sets. A decode operation on a signature extracts an *exact* bitmask corresponding with the sets in the cache. For each selected set in the bitmask, Bulk reads the addresses of the valid lines in the set and applies the membership operation on each address. This way, the list of addresses is enumerated.

Before a processor can commit, it arbitrates system-wide for commit permissions. Once it obtains permission, it broadcasts its write signature to other processors. Unlike earlier proposals for conflict detection at commit, Bulk does not send out the list of addresses. When a processor receives the write signature, the processor performs an intersection operation between its local read signature and the incoming write signature. This operation detects conflicting addresses in the two signatures. If the receiving processor aborts, the processor uses its local write signature to identify and invalidate speculatively modified lines in the cache. It then clears the read and write signatures. To avoid extending each cache line with a bit to track speculatively written lines, Bulk requires that if a set has a speculatively modified cache line, then that set cannot have any nonspeculatively modified cache line. This restriction prevents an invalidation of a nonspeculatively modified cache line in the set. This could happen because a signature is an inexact representation of addresses and may alias to include nonspeculatively modified cache lines. The signature, however, expands to an exact representation of cache sets with speculatively modified cache lines.

If the receiving processor does not abort, all lines that are written to by the committing processor and are present in the receiving processor's cache must be invalidated. The write signature of the committing processor is used to invalidate these lines. If a cache line receives regular coherence invalidations, the hardware performs a membership operation of the incoming address on the local signature to check for a data conflict. Execution aborts if a match exists.

Bulk supports conflict detection at word granularity without changing the granularity of the cache coherence protocol. If word granularity is used for conflict detection, then Bulk must merge updates made to different parts of a cache line on different processors—one processor that is committing and another processor that receives the write signature (these updates do not conflict at a word granularity). For this, Bulk uses special hardware to identify a conservative bitmask of the words the receiving processor updated in the cache line. It then reads the latest committed version of the cache line and merges the local updates with the committed version. This way, it retains its updates while incorporating the updates of the committing processor. Bulk does not require cache-line bit extensions for this.

Bulk supports a nested transaction model where an inner transaction does not become visible to other threads until the outer transaction commits, but if a nested transaction experiences a conflict and aborts, then execution is rolled back only to the beginning of that transaction. A separate read and write signature is maintained for each nested level. Incoming write signatures or addresses are checked (via intersection or membership) with each of these nested signatures for conflicts. If any is found, then execution rolls back to the beginning of the aborting transaction. If an inner transaction commits, the signatures are not combined. When the outer transaction commits, then a union of all the write signatures is broadcast to other processors for conflict detection. Bulk can track multiple read and write sets for the nesting levels without requiring per-nesting level cache bits (Sections 4.7.1 and 4.7.2 describe alternatives). The paper suggests moving signatures into a memory location if more nesting levels are required than that provided by the processor.

Bulk does not describe an overflow scheme when transactional data does not fit the cache. It suggests that Bulk signatures to detect if an address is in the overflow space without a lookup of the overflow space. The signatures support addresses larger than the cache footprint.

Bulk uses approximate representations of physical addresses for conflict detection. The LTM had a similar approach using the 0 bit (Section 4.4.2). In such schemes, unrelated processes may alias and interfere with one another, and prevent performance isolation. Bulk records all signatures in hardware. Because Bulk does not provide a means to move signatures and transactional state from a processor's hardware structures to another processor, Bulk cannot migrate a transaction thread from one processor to another. The interaction with page mapping changes is undefined.

CEZE ET AL. 2006	
Strong or Weak Isolation	Strong
Transaction Granularity	Cache line/word
Direct or Deferred Update	Deferred (in cache and in overflow space)
Concurrency Control	Optimistic Commit serialized globally
Conflict Detection	Late write–write conflict Late write–read conflict
Inconsistent Reads	No
Conflict Resolution	Via global commit coordination
Nested Transaction	Flattened/Closed

4.5   UNBOUNDED HTMS

Unbounded HTMs allow a transaction to survive context switch events. Such support typically requires transactional state to be maintained in persistent space, preferably virtual memory. The three papers in this section do this. Ananian et al. (Section 4.5.1) present an unbounded HTM design that maintains all transactional metadata information in a table in the system's virtual address space, and architecturally extends each memory block with metadata information to detect conflicts without caches and cache coherence support. The table also maintains an undo log for aborts (direct-update TM system). UTM hardware operates on this table. Rajwar et al. (Section 4.5.2) present an unbounded HTM design that maintains metadata information for transactions that exceed hardware resources in a table in the application's virtual address space, and adds processor support to selectively intercept load and store operations, and invoke microcode routines to check for conflicts against the overflowed transactional state in the table. The table also maintains a buffer for transactional updates (deferred-update TM system). These two papers did not present implementation-specific details or an evaluation.

Zilles and Baugh (Section 4.5.3) extend the Rajwar et al. unbounded HTM (Section 4.5.2) in numerous ways. First, they modify it to support direct updates instead of deferred updates. This optimizes the system for commits, which are more common. Second, they extend the HTM interface beyond a simple begin and end, and integrate the HTM with software functions of compensation, waits, and retry. The system was prototyped on the Linux operating system.

In these systems, execution takes advantage of hardware caches for buffering and tracking transactional accesses, and for conflict detection if hardware resources are sufficient for all transactions. When a transaction overflows into a table in software-maintained space, then these systems also need to check for conflicts against this table.

4.5.1   Ananian et al./UTM, HPCA 2005

Overview

This paper describes unbounded transactional memory (UTM); a hardware transactional memory system that allows a transaction to survive context switches and exceed hardware buffer limitations [3]. The authors believe that such support is necessary for transactional memory to be widely used.

UTM proposed architectural extensions to decouple transactional state maintenance and conflict checking from the hardware caches and cache coherence protocol. To maintain transactional state outside hardware, UTM introduced a system-wide memory-resident data structure, called the XSTATE. The XSTATE maintained the transactional information, including read and write sets, for all transactions in the system. To make conflict checks independent of the

coherence protocol, UTM extended every memory block in the system with access bits and an owner pointer. This extended information was also stored in the page tables and on disk. A transaction could consult this extended information on memory accesses and determine conflicts. The UTM hardware operated on these extensions. UTM did not require a cache or cache coherence protocol for correct execution; the caches only accelerated UTM execution.

UTM was the first proposal to allow for transparent overflow of transaction state while maintaining transactional memory semantics and allowed a transaction to survive a context switch. The paper also describes a restricted implementation of UTM, called large transactional memory (LTM). We discussed LTM in an earlier section (Section 4.4.2).

Programming Interface
The UTM programming interface is the same as LTM (Section 4.4.2). The operating system (OS) manages the allocation of the UTM data structures, and these data structures are not visible to the application programmer.

Implementation
In UTM, the hardware operates on the UTM data structures. Hardware saves register state for recovery on aborts. UTM assumes a physical register file organization in the processor. It takes a snapshot of the processor's register rename table and prevents the release of the physical registers that are live in at the time the transaction starts. Nested transactions are flattened and this is captured in the nesting depth counter. UTM makes the abort handler, nesting depth, and the microarchitecture register rename table snapshot part of the architectural state. The operating system is responsible for saving and restoring these on context switches.

UTM proposes two key architectural extensions. The first extension supports persistent maintenance of transaction state. The second extension allows for checking conflicts without involving a cache coherence protocol.

Fig. 4.13 shows the architectural UTM extensions. The first architectural extension is the `XSTATE`; a single memory-resident data structure that represents the state of all transactions in the system. The OS manages the allocation of this data structure. This structure has three key components:

a) *Commit record.* The commit record tracks the status of the transaction, which may be in `pending`, `committed`, or `aborted` state.

b) *Log entry.* A log entry is associated with a memory block read or written in the transaction. The log entry provides a pointer to the block in memory, the old value of the block, and a pointer to the commit record. These pointers also form a linked list to all entries in the transaction log that refer to the same block.
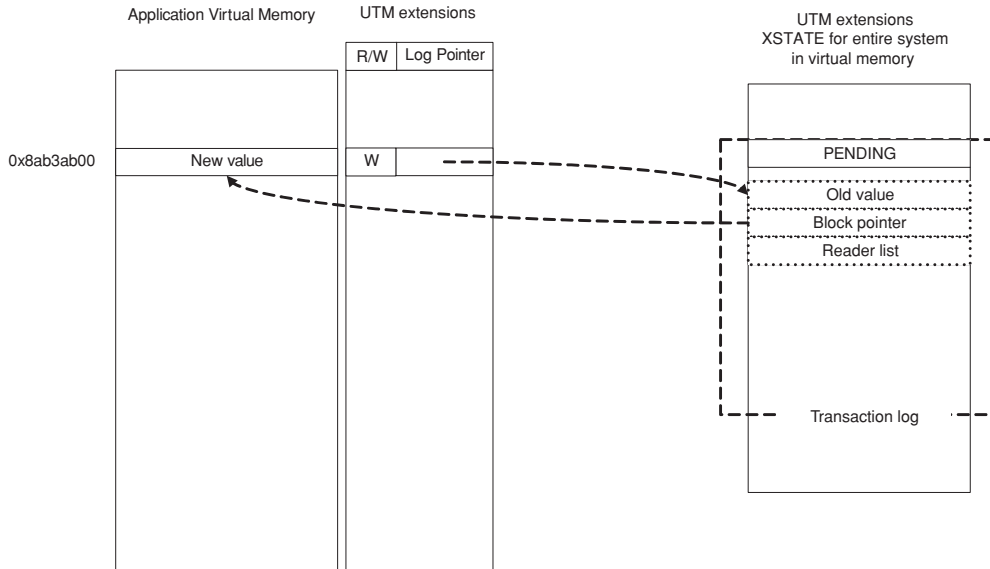
FIGURE 4.13: UTM data structures

c) *Transaction log.* The transaction log contains the commit record and a vector of log entries corresponding with the thread executing the transaction. Each active transaction in the system has its own transaction log. The OS allocates the transaction log and two hardware control registers record the range for the currently active threads. The transaction log also records a timestamp. UTM uses this timestamp to resolve conflicts between transactions.

The second architectural extension (for conflict checks) is the addition of a RW bit and a pointer, called the log pointer, for each block in memory and disk (when paging). If the RW bit is not set, then no active transaction is operating on that memory block. If the bit is set, then the pointer for the memory block points to the transaction log entry for that block. This check is performed by the UTM system on all memory accesses. The access may result in a pointer traversal to locate the transaction that owns this block.

The log pointer and the RW bit for each user memory block are stored in addressable physical memory to allow the operating system to page this information. Every memory operation (both, inside and outside a transaction) must check the pointer and bits to detect any conflict with a transaction. Page tables also record information about the locations. Since during a load or store operation, an XSTATE traversal may result in numerous additional memory accesses, the processor must support restart of a load or store operation in the middle of the traversal, or the operating system must ensure that all pages required by an XSTATE traversal are

simultaneously resident. One can inspect the log pointer and determine whether a transaction was in the XSTATE.

UTM had the option to store either the new values or the old values of the memory location in the log. The design sketched in the paper optimizes for commits and records old values of the memory location in the log; the transaction updates memory directly in place. An abort requires restoring the older values to the transactionally updated memory locations. Commit operations do not require data movement but require iterating the log entries and clearing the log pointers.

While UTM did not require caches for functional correctness, it used caching for performance. If a transaction fits in the local HTM cache, then UTM uses the HTM mechanisms of cache coherence and cache buffering. The log pointer and RW bits are not updated if the state fits in the cache. If a transactional line is evicted, then the UTM system updates the log pointer and RW bits, and creates its log entry. To ensure that the HTM can correctly interact with the XSTATE, the processor must always check the log pointer and RW bits for a memory block, even if no transaction has overflowed. All conflicting transactions can be identified by walking the log pointer. The paper describes optimizations to avoid unnecessary writebacks on aborts and commits.

ANANIAN ET AL./UTM 2005	
Strong or Weak Isolation	Strong
Transaction Granularity	Memory block/Cache line
Direct or Deferred Update	Deferred in cache/Direct in memory
Concurrency Control	Optimistic
Conflict Detection	Early
Inconsistent Reads	None
Conflict Resolution	Timestamp
Nested Transaction	Flattened

4.5.2   Rajwar, Herlihy, and Lai, ISCA 2005

Overview

The authors describe virtual transactional memory (VTM); an HTM system that shields programmers from hardware implementation details by virtualizing limited resources such as hardware buffer sizes and scheduling durations [40]. This allows transactions to survive context switches and exceed hardware buffer limitations. The virtualization is analogous to how virtual memory shields programmers from limited physical memory.

VTM decoupled transactional state maintenance and conflict checking from the hardware. VTM used software data structures resident in the application's virtual memory. The transaction address and data table (XADT) maintained information for transactions that overflowed hardware caches or exceeded scheduling durations, and served as the central coordination structure for overflowing transactions. Transactions that fit in the cache do not add state to the XADT, even if another transaction has overflowed. Processors continually monitored a shared memory location that tracked whether the XADT has overflowing transactions. To accelerate the conflict check and avoid an XADT lookup, VTM used a software conflict filter, the XADT filter (XF). The XF returned whether a conflict existed without requiring to look up the XADT.

VTM added a microcode/firmware capability in each processor to operate on the data structures, similar to hardware page walkers in today's processors. In the absence of any overflows, VTM executed transactions using HTM without invoking any VTM-related machinery or overhead. In the presence of overflows, the VTM machinery on each processor would take appropriate actions without interfering with other processors.

VTM ensured that transactions execute directly in hardware without overheads when hardware was sufficient. It was the first proposal to provide unbounded transaction using only processor-local support. In contrast, UTM required architectural extensions to memory blocks.

Programming Interface

VTM was an overflow management and conflict checking system for transactional memory state. The paper did not discuss a programming model, semantics, or software policy. The paper assumes a simple transactional memory programming model in which an application has multiple software threads running in a single shared virtual address space. Each thread has a transaction status word (XSW) monitored continually by the processor executing the thread. Each thread serially executes transactions explicitly delimited by the instructions begin_xaction and end_xaction. Nested transactions are flattened.

The XADT is common to all transactions sharing the address space. Transactional state can overflow into the XADT in two ways: a running transaction may evict individual transactional cache lines, or an entire transaction swaps out, evicting all its transactional cache lines. Each time a transaction issues a memory operation that causes a cache miss and the XADT overflow count signals an overflow in the application, it must check whether the operation conflicts with an overflowed address.

If the XADT signals a conflict with an overflowing transaction, the conflict is resolved based on a uniform but unspecified policy. Such a policy is maintained as part of the XADT and it may consider numerous factors, including a transaction's age and its operating system thread priority. The operating system handles the allocation of the VTM software data structures and the application programmers do not directly operate on these structures.
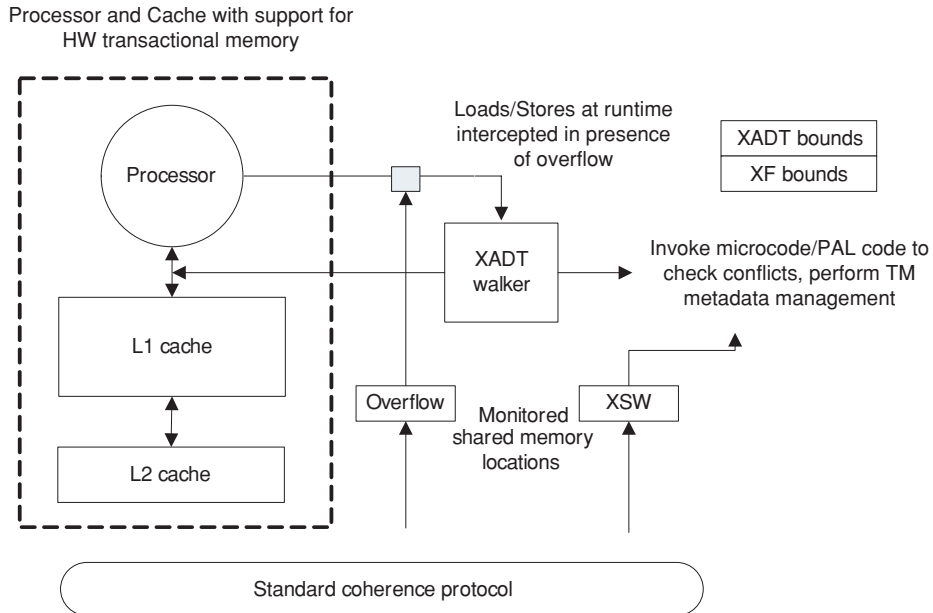
Processor and Cache with support for
HW transactional memory

FIGURE 4.14: VTM hardware organization

Implementation

Fig. 4.14 shows the hardware organization for a VTM system. VTM assumes that the processor has support for a typical bounded HTM; it has support to buffer tentative updates, track transactional accesses, and detect conflicts using hardware mechanisms. VTM does not dictate a type of HTM implementation. The HTM executes transactions that can fit within the hardware and scheduling resources. VTM extends the HTM system with hardware support in the form of continually monitored locations, selectively intercepting memory operations from the processor and invoking appropriate hardware assists, and microcode/firmware that implements these assists for operating on the VTM data structures.

VTM implements a deferred-update TM system. It leaves the original data in place and records new data in the XADT. When an overflow occurs, the VTM system moves the evicted address into a new location in the XADT. The transaction state for the overflowing transaction is split between caches and the XADT. In a deferred-update system, reads must return the last write in program order. For performance, reads must quickly know which location to access for the buffered update. The processor caches the mapping of the original virtual address to the new virtual address in a hardware translation cache, the XADC. This cache speeds subsequent accesses to the overflowed line by recording the new address of the buffered location.
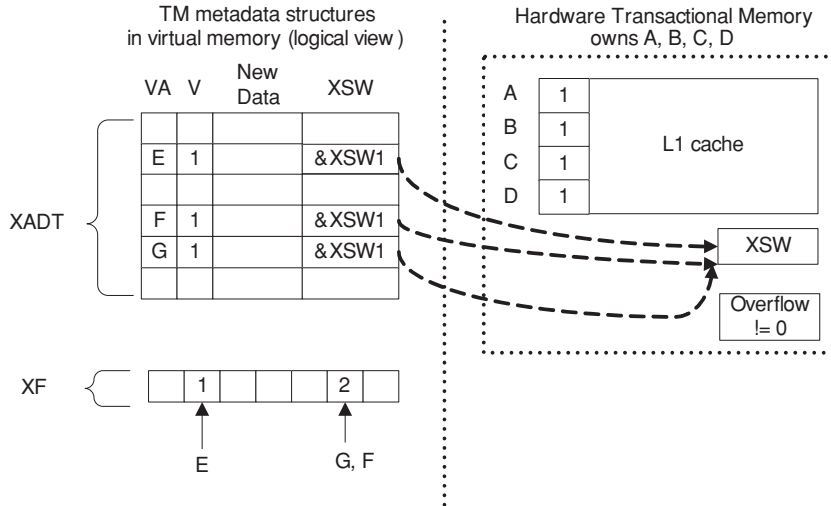
FIGURE 4.15: VTM HW and SW modes

VTM requires the virtual address of the evicted cache line to be available for moving it into the XADT. When a context switch occurs, VTM moves transactional state from the hardware cache to the XADT. This increases the context switch latency for such transactions.

If no transactions overflow in the application, the Overflow monitored location retains the value zero. Consequently, transactions execute directly in hardware using the baseline HTM and do not invoke any VTM machinery. When a transaction overflows, then the Overflow monitored location changes to 1 and processor-local VTM machinery gets involved. It intercepts load and store operations from the processor and invokes assists to microcode/firmware. The processor running the overflowing transaction performs overflow transaction state management. The assists add metadata information about the overflowing address into the XADT. An XADT entry records the overflowed line's virtual address, its clean and tentative value (uncommitted state), and a pointer to the XSW of the transaction to which the entry belongs. The other processors perform lookups against overflow state. A hardware walker, similar to the page miss handler, performs the lookups.

Fig. 4.15 shows the coexistence of overflowed software and hardware state. The left shows the software-resident overflow state and the right shows the hardware-resident nonoverflow state for a transaction. The overflow entries have a pointer to the XSW, which allows other transactions to discover information about a given transaction. The XF is a software filter (implemented as a counting bloom filter [5, 14]) that helps a transaction to determine whether a conflict for an address exists. VTM uses this filter to determine quickly if a conflict for an address exists and avoid an XADT lookup. In the example shown, G and F map to the same entry in XF and

thus result in a value 2 for that entry. Strong isolation ensures that the committed lines copied from the XADT to memory are made visible to other non-transactional threads in a serializable manner. Other threads (whether in a transaction or not) never observe a stale version of the logically committed but not yet physically copied lines.

Conflicts between nonoverflowing hardware transactions are resolved using HTM mechanisms. For transactions that have overflowed, a processor requesting access to such an address will detect the conflict when it looks up the XADT. This localizes conflict detection, allows conflicts with swapped transactions to be detected, and avoids unnecessary interference with other processors.

The paper also describes criteria that transactional memory virtualization schemes must meet to integrate into existing systems. Virtualization must ensure that the performance of common-case hardware-only transactional mode is unaffected. Conflict detection between active transactions and transactions with overflowed state should be efficient, and should not impede unrelated transactions. Committing or aborting a transaction should not delay transactions that do not conflict. Context switches and page faults may impede transaction progress, but should not prevent transactions from eventually committing. Nontransactional operations may abort transactions but should not compromise any transaction's consistency (strong isolation). Lastly, this virtualization should be transparent to application programmers, much in the way virtual memory management is.

RAJWAR ET AL. 2005	
Strong or Weak Isolation	Strong
Transaction Granularity	Cache line
Direct or Deferred Update	Deferred (in cache and in XADT)
Concurrency Control	Optimistic
Conflict Detection	Early
Inconsistent Reads	None
Conflict Resolution	Software controlled
Nested Transaction	Flattened

4.5.3   Zilles and Baugh, TRANSACT 2006

Overview

This paper describes an alternate implementation of the VTM proposal (Section 4.5.2). First, it modifies VTM to perform direct instead of deferred updates to memory, and optimizes

the implementation for commit operations. Second, it describes software and instruction set extensions to allow VTM transactions to wait on events and retry efficiently by communicating with the scheduler, to pause themselves temporarily, and to perform compensation actions [49]. To avoid confusion with the original VTM paper (Section 4.5.2), we refer to the design in this paper as the Illinois VTM system.

The Illinois VTM system adds new instructions, new software exception support, and extends the VTM metadata structures. To allow a transaction to efficiently wait and retry, the paper introduces four new software exceptions that allow the hardware to communicate with the software scheduler. The `xact_wait` and `xact_completion` allow a transaction to deschedule and wait for another transaction and for the other transaction to wake up the waiting transaction. The `retry` and `retry_wakeup` software exceptions allow a transaction to deschedule and wait for a data conflict. Two new instructions `xact_pause` and `xact_unpause` allow a transaction to escape the transaction domain and execute nontransactionally. The system provides support for executing compensation actions when such a transaction later aborts. The system maintains information about waiting transactions and compensation functions in software data structures. The paper describes the use of pause and compensation for memory management inside transactions.

Programming Interface

The paper describes the software interface to the TM system. The Illinois VTM system has two key software metadata structures that maintain a transaction's software state. Because this state exists in software, programmers can operate on this state.

The local transaction state segment (LTSS) maintains per-transaction state. The LTSS is demand allocated. The LTSR, a new architectural register, points to the LTSS of the transaction executing on the processor. The LTSS fields are shown below. The XSW maintains the transaction's status. The `transaction_num` provides an identifier for conflict resolution and `reg_chkpt` stores the transaction's register state for recovery. The remaining fields provide support for compensation and waiting. A programmer can register a list of compensation functions (`comp_lists`) to execute when this transaction aborts. The programmer, via the software exception interface, can also record information about waiting transactions (`waiters`, `waiters_chain_prev`, `waiters_chain_next`):

```
typedef struct local_xact_State_s {
    xsw_type_t                 xsw;
    int                        transaction_num;
    x86_reg_chkpt_t            *reg_chkpt;
    comp_lists_t               *comp_lists;
    struct transaction_state_s *waiters;
```

```
    struct transaction_state_s *waiters_chain_prev;
    struct transaction_state_s *waiters_chain_next;
    ...
} local_xact_state_t;
```

The global transaction state segment (`GTSS`) maintains all transaction state for the address space in which the application is executing. The kernel allocates the `GTSS`. The `GTSR`, a new architectural register, points to the `GTSS`. The `GTSS` fields are shown below. The `overflow_count` records the number of transactions that have exceeded hardware resources and are executing with software support. The `XADT` points to the software data structure that maintains the address and data undo log for transactions executing with software support. Two locks exist for correctly operating on the shared data structures:

```
typedef struct global_xact_State_s {
    int                 overflow_count;
    xadt_entry_t        *xadt;
    spinlock_t          gtss_lock;
    spinlock_t          xact_waiter_lock;
    ...
} global_xact_state_t;
```

The paper describes a prototype implementation of the Illinois VTM system in a Linux operating system executing on the x86 architecture, primarily because of the lack of user-level exception handling support in x86. If user-level exception support is available, then kernel support for exceptions is not necessary. Programmers can write these exceptions into the transaction and use them for communication between the hardware and software.

Implementation

The paper does not describe the details of the unbounded HTM implementation beyond noting that the original VTM proposal was modified to support direct updates. Instead, it focuses on the key aspects that provide the HTM with enhanced software capability. The paper introduced new software exceptions for the HTM hardware and software schedulers to communicate. It does not require hardware support beyond its baseline HTM implementation:

- `xact_wait`, `xact_completion`. A transaction raises the `xact_wait` exception when it wants to wait. Consider two transactions T_1 and T_2 and T_2 wants to wait for T_1. T_2 raises the `xact_wait` exception. This transfers control to a software handler. The handler marks T_2 as unavailable for scheduling and informs the scheduler. The handler also records this information in T_1's `LTSS XSW` field and adds T_2 to the list of waiting transactions. This communicates waiter information to T_1. When T_1 aborts or commits,

it will raise the `xact_completion` exception. The software handler will scan the list of waiting transactions (T$_2$ in this example) and wake them.

- `retry, retry_wakeup`. A transaction that wants to deschedule and wait for a data conflict raises the `retry` exception. The handler marks the transaction as blocked, sets the transaction's priority to the lowest to make it abort on all conflicts. It marks its `LTSS XSW` to indicate that another transaction will wake this transaction. If a transaction on a list of waiters is aborted, then the `retry_wakeup` exception occurs. The handler wakes up the waiting thread.

Care is required to ensure that wakeup responsibility is properly transferred. The Illinois VTM system uses a `compare&swap` operation to ensure that the transaction to which a transfer of responsibility occurs has not yet aborted or committed.

The paper uses two new instructions, `xact_pause` and `xact_unpause` to allow a transaction to escape its transactional scope and execute nontransactionally. During a pause, accesses are not added to a transaction's read and write sets, and memory updates are performed directly to memory and to the undo log. Updating the undo log ensures that these updates are not lost when a transaction aborts. The system allows a thread to register a data structure that includes pointers to two linked lists, one for actions to perform on an abort and another to perform actions on a commit. Each transaction maintains a list of actions and tracks whether compensation code runs or a deallocation of the action occurs.

The paper describes the use of compensation for memory management including cases where an allocation occurs inside a transaction and the deallocation occurs outside the transaction. It presents wrappers to perform `malloc` and `free` operations nontransactionally and to allow compensation if necessary.

ZILLES AND BAUGH. 2006	
Strong or Weak Isolation	Strong
Transaction Granularity	Cache line
Direct or Deferred Update	Deferred in cache, direct in memory
Concurrency Control	Optimistic
Conflict Detection	Early
Inconsistent Reads	None
Conflict Resolution	Software controlled
Nested Transaction	Flattened

4.6   HYBRID HTM–STMS/HARDWARE-ACCELERATED STMS

An alternate approach to providing unbounded transactions in a HTM (Section 4.5) is to assume an STM as the baseline and use an HTM or HTM-like mechanisms to improve performance. This approach maintains flexibility while the STM is used, and achieves unbounded behavior with bounded HTM mechanisms. However, it gives up on various HTM benefits (See Section 3.1 and Section 4.1 for the tradeoffs of STM and HTM systems). Chapter 5 discusses possible ways unbounded TM systems may evolve.

Lie describes a hybrid HTM–STM proposal in which an object-based STM is the baseline (Section 4.6.1). The TM system, however, first executes the transaction in an HTM and on failure executes the transaction in an STM. The key here is to integrate an object-based software system with a line-based hardware system, and detecting conflicts between the two. Kumar et al. describe a similar scheme in which they integrate a line-based HTM and an object-based STM (Section 4.6.2). These approaches use the HTM to execute transactions in hardware directly, when possible, and enhance the software to perform appropriate conflict checks. Shriraman et al. describe an alternate approach where instead of directly integrating an HTM and STM they provide an instruction set interface for the STM to control individual HTM hardware mechanisms at a fine granularity (Section 4.6.3).

Proposals in this section are based on STM proposals discussed in Chapter 3 and the descriptions are not repeated here.

4.6.1   Lie, MIT ME Thesis 2004

Overview

Lie proposes a hybrid hardware–software transactional memory system (HSTM) in which a transaction first executes as a hardware transaction using a line-based HTM, and if unsuccessful, restarts and executes as a software transaction using an object-based STM [32]. This approach provides unbounded transactions with a simple HTM, but it has overheads and requires re-compilation for integration into existing execution environments.

In HSTM, two versions of transaction code are generated, one for an HTM and another for an STM. The HTM transaction code also contains software checks to allow the hardware transaction running in an HTM to detect conflicts with a software transaction running in an STM.

This was the first published work on a hybrid transactional memory model where a transaction first runs in an HTM and if unsuccessful, runs in an STM.

Programming Interface

The STM in HSTM is an object-based FLEX STM (Section 3.4.7). The compiler directly implements the STM and performs analysis to optimize checks. For example, if a transaction

reads from a field it previously wrote, it reads the value directly from the transactional version without checking the original version. In HSTM, if an STM transaction writes an object, it replaces the value with a special value FLAG and updates the object in a separate buffer. HTMs use this value to detect conflicts.

Implementation

The HTM is line based where caches track memory accesses, detect conflicts, and buffer updates. HSTM generates two versions of transaction code: one that executes as an HTM and another that executes in an STM. The HTM version of the transaction code performs checks in software to detect conflicts with a transaction executing in an STM. A special FLAG value is used for this. On each transactional load in a hardware transaction, the transaction checks the loaded value. If the value is FLAG, then a running software transaction may have written to the field. The HSTM aborts the hardware transaction. On each transactional store, HSTM checks to ensure that the readers pointer is NULL. If it is not NULL, then the HSTM aborts the hardware transaction. To allow the hardware transaction to abort itself, HSTM uses a xABORT instruction.

Lie discussed tradeoffs between HSTM [32] and UTM [3, 32]. HSTM incurs memory overhead due to additional software checks and the additions of the readers and versions pointer even when running hardware transactions. Since HSTM runs most of the transactions in hardware, this overhead is usually unused. HSTM can support unbounded transactions with minimal hardware modification. However, HSTM has overheads as compared to an HTM and cannot call into legacy functions.

4.6.2   Kumar et al., PPoPP 2006

Overview

This paper describes hybrid transactional memory, a hardware–software transactional memory system where a transaction executes first as a hardware transaction on a line-based HTM, and if it fails, it executes as a software transaction in an object-based STM, DSTM [28]. This way, the system can achieve the performance of a hardware transaction whenever possible, and fall back to a software transaction when hardware was insufficient. The paper describes extensions to a DSTM system to make it coexist with an HTM.

Hybrid transactional memory introduces new instructions, and extends the DSTM data structures to integrate with an HTM. Hybrid transactional memory requires two versions of a transaction, one for executing in an HTM and another for executing in a DSTM. The instruction extensions for the two versions are different because of differences in design of the DSTM and HTM systems. The DSTM system was object based and explicitly identified

transactional operations. The HTM system was line based and implicitly assumed all operations in the transaction as transactional. Hybrid transactional memory maintains DSTM semantics when the programmer explicitly aborts a transaction. When a transaction running as a hardware transaction aborts, the execution aborts and restarts as a software transaction in an STM. This maintains abort semantics for the DSTM.

Programming Interface

Hybrid transactional memory uses DSTM as its baseline STM and inherits its programming interface. The paper identifies two DSTM actions as contributing to significant overheads: versioning, and allocation and copying required when a transaction opens an object for writing. HTM is used to remove these overheads where possible. A transaction executes either in hardware mode or in software mode. During execution, the transaction opens all its objects in the same mode as its execution. The paper introduces the following instruction set extensions for the two modes:

1. `xba`. This instruction starts a transaction in hardware mode. All operations inside the transaction default to transactional.
2. `xbs`. This instruction starts a transaction in software mode. All operations inside the transaction default to nontransactional.
3. `xc, xa`. These instructions commit and abort the transaction.
4. `ldx/stx`. These are explicit transactional load and store operations and are used inside transactions in software mode.
5. `ldr/str`. These are explicit nontransactional load and store operations and are used inside transactions in hardware mode.
6. `sstate/rstate`. These instructions checkpoint and restore architectural register state.
7. `xhand`. This instruction sets up an exception handler for aborts.

The authors do not discuss the implications of the new instructions on processor implementations or on the instruction set architecture design.

Implementation

Hybrid transactional memory extends the objects in DSTM and extends the processor and its caches. Fig. 4.16 shows the original DSTM TMObject and the TMObject with extensions for Hybrid transactional memory.

A software mode transaction only uses the HTM hardware to monitor the `Locator State` object. Another hardware or software mode transaction can abort this transaction by
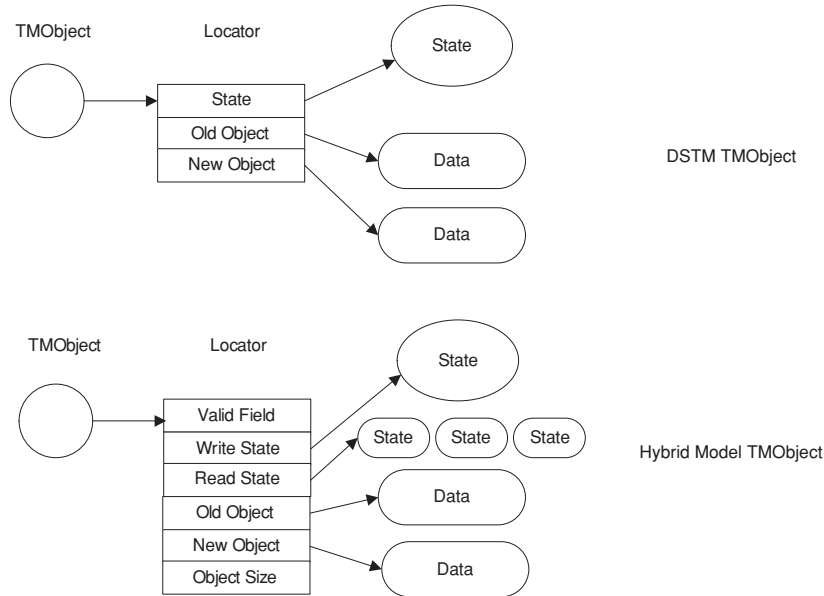
FIGURE 4.16: Hybrid transactional memory software meta data structures

writing the `State` object. Software mode transactions on opening a `TMObject` check the `read` and `write` fields of the locator for conflicts with other transactions. If the object is opened for writing, then the copy of the object occurs only after the transaction has switched the `TMObject`. This is because a concurrently executing hardware mode transaction might have modified the data in place. A hardware mode transaction does not have a locator and it updates data directly in the cache without making a software copy.

Fig. 4.17 shows the hardware implementation. The paper describes the hardware for a two-thread multithreaded processor. The HTM uses a register checkpoint for recovering register state. Two new structures are added. A hardware transactional buffer records two versions for a line, the transactionally updated value and the value before the update. A hardware transaction state table has two bits per hardware thread. These bits record if the thread is executing a transaction and if the transaction started in a software or hardware mode.

Each line in the cache has two bit-vectors recording reads and writes, one for each hardware thread. The HTM uses these vectors to detect conflicts. Conflicts between two hardware transactions are resolved by aborting the transaction that receives the conflicting request. A conflict between a hardware transaction and a nontransaction request aborts the transaction.

When a hardware transaction detects a conflict and aborts, the hardware invalidates all transactionally written lines in the transactional buffer and clears all read and write vectors. The abort sets an exception flag in the transaction state table but does not abort execution
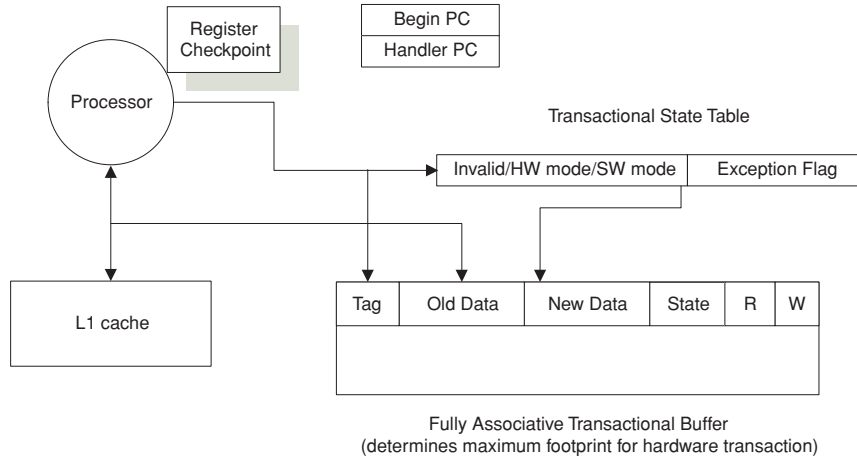
FIGURE 4.17: Hybrid transactional memory hardware organization

right away. The abort is triggered and a software abort handler invoked when the transaction performs another memory operation or tries to commit.

Hardware and software mode transactions can abort each other. A hardware mode transaction aborts another hardware mode transaction using cache coherence. The hardware mode transaction aborts a software mode transaction by reading the `State` field of the `Locator` and atomically replacing the value `ACTIVE` with `ABORTED`. Software mode transactions detect the abort because they cache the `Locator`. Software mode transactions abort hardware mode transactions by swapping the `TMObject` pointer to the new locator it had created. This aborts the hardware mode transaction because the HTM has transactionally read this pointer. Software mode transactions abort other software mode transactions using DSTM mechanisms.

The preferred execution mode for a transaction can be determined heuristically. The paper assumes that a transaction tries to run in HTM three times before switching to an STM. Profile-driven hints and distinguishing between conflict and capacity-induced restarts can help decide the default execution mode of a transaction.

In DSTM, a transaction may abort itself explicitly. When this happens, DSTM requires nontransactional object updates made in the transaction to be persistent and not rolled back. However, hardware mode transactions consider all updates as transactional and roll back all state. To maintain DSTM requirements, if an explicit abort is executed in hardware mode, then execution aborts and restarts in software mode and reexecutes the abort.

Context switches pose a problem for hybrid transactional memory as described. However, checking the transaction's state on rescheduling to ensure that it did not abort would address the problem.

4.6.3   Shriraman et al., TRANSACT 2006

Overview

The paper describes Rochester transactional memory (RTM), a hardware-accelerated software transactional memory system [43]. RTM uses HTM mechanisms only to accelerate an STM. RTM assumes an object-based STM system. In the hybrid approaches discussed so far, the behavior of the HTM component of the hybrid HTM–STM system was determined by the HTM implementation and the STM system did not have fine control over HTM mechanisms. RTM on the other hand provides the STM with direct and fine control of individual HTM mechanisms.

RTM introduces instruction set extensions through which an STM can control HTM mechanisms and perform STM-specific operations directly in hardware. RTM, in a key difference from prior HTM proposals, gives software control over which cached lines are exposed to the cache coherence protocol and which are not. RTM also provides software with control over which cache lines should be transactionally monitored and kept exposed to the cache coherence protocol.

Programming Interface

RTM assumes an object-based RSTM (Section 3.4.8) and inherits its programming interface. RTM assumes that each object in the RSTM is allocated to its own cache line. This avoids false sharing because hardware cache coherence works at the granularity of cache lines. It increases the data footprint of the program because otherwise multiple objects could reside on the same cache line. RTM only manages objects explicitly identified as `shared`; all other objects are treated as nontransactional.

A transaction must explicitly open a shared object for either read-only access or a read–write access before using the object. The `open_RO` function opens an object for read-only access. It returns a pointer to the current version of the object and maintains information for conflict tracking. The `open_RW` function opens an object for read–write access. It creates a clone of the object and returns a pointer to it. Other transactions use the original copy. A transaction abort discards the clone and a transaction commit installs this clone as the latest copy replacing the original copy using an atomic operation. A transaction uses a transaction descriptor to record whether it is `active,` `committed,` or `aborted.` Each object's header is extended with a pointer to the transaction descriptor of the last transaction that modified the object. The header also has pointers to the original and cloned data.

The sequence below shows insertion into a sorted linked list in C++ using the RTM programming interface:

```
void intset::insert (int val) {
   BEGIN_TRANSACTION;
```

```
      const node* previous = head->open_RO();
      const node* current = previous->next->open_RO();
      while (current != NULL){
         if (current->val >= val) break;
         previous = current;
         current = current->next->open_RO();
      }
      if (!current ||  current->val > val) {
         node *n = new node (val, current->shared());
         previous->open_RW()->next = new Shared<node>(n);
      }
   END_TRANSACTION;
}
```

RTM first executes a transaction by using hardware to buffer writes (eliminate clone overhead) and for conflict detection (eliminate validate overhead). If the transaction fails, RTM retries the transaction as a software-only transaction. Hardware state is cleared on every kernel and user-level thread switch.

RTM introduces the following instruction set extensions:

1. `SetHandler(HandlerAddress)`. This instruction, typically executed at the beginning of every transaction, specifies the address to which control transfers on an abort. This is similar to the `xhand` instruction in Kumar et al. (Section 4.6.2).

2. `TLoad(Address,Register)`, `TStore(Register,Address)`. RTM uses these instructions to access transactional data. While these instructions bring data into the cache, once in the cache, these cache lines do not participate in the coherence protocol. For example, exclusive ownership requests from other processors do not invalidate these cache lines. Other processors cannot see data stored in these cache lines using the `TStore` instruction. RTM relies on a software protocol to provide correct execution when invalidation requests to these lines are ignored.

3. `ALoad(Address,Register)`. This instruction specifies an address to monitor. If the address receives an invalidation request from another processor, RTM immediately calls an abort handler. A common use of this instruction is for a reader to monitor a transactional object's header for writes by other transactions. This is a similar use to the `ldx` instruction in the Kumar et al. hybrid TM proposal (Section 4.6.2).

4. `ARelease(Address)`. This instruction undoes the effect of the `ALoad` instruction and stops RTM from monitoring this address. This is a performance optimization and serves the same purpose as the `release` instruction in McDonald et al. (Section 4.7.2.)

5. `CAS-Commit(Address,O,N)`. This instruction performs a `compare&swap(CAS)` on `Address` and if successful, commits the hardware state of the transaction. If the CAS fails, then RTM discards the transactional hardware state and control transfers to the handler specified by the `SetHandler` instruction. This instruction operates on the transaction's status. The instruction combines the software commit operation of an STM with the hardware commit operation of an HTM, similar to how VTM performs its commit (Section 4.5.2).

6. `Abort`. A program uses this instruction to explicitly abort an execution.

7. `Wide-CAS(Address,O,N,K)`. This instruction supports a CAS on K adjacent words that sit in a single cache line. It is used to accelerate certain STM-specific operations.

The authors do not discuss the implications of these new instructions on modern processor implementations or their impact on instruction set design and instruction set validation.

The STM used in RTM implements a visible reader protocol (Section 2.3). This is to allow a software transaction and a hardware transaction to detect conflicts. Each software transaction reads its own transaction descriptor using the `ALoad` instruction. Since RTM is based on an object-based STM, conflict detection occurs only on the object header; data in the object does not participate in conflict detection.

Implementation

RTM extends the hardware with support to buffer transactional state in a cache; mechanisms to prevent a cache line from participating in a cache coherence protocol, and to mark address in the cache to be actively monitored for invalidation operations. When RTM executes in hardware mode, it uses write buffering and conflict detection for all accessed objects. If a hardware mode transaction aborts, it restarts completely in software.

When RTM executes a transaction with hardware support, the data cache buffers transactionally written data. This data is not exposed to other processors and the cache line does not participate in the cache coherence protocol. For example, invalidations from other processors to this cache line are ignored. This allows two transactions to write the same data at the same time. Software is responsible for providing correct execution when this happens. RTM immediately aborts a transaction when a monitored cache line receives an invalidation, and is similar to the asynchronous abort in many HTMs. This avoids the inconsistent read problem faced by some TM systems. RTM has hardware support for a wide compare&swap which atomically inspects

and updates adjacent locations of memory in the same cache line. This removes an STM pointer indirection.

RTM uses a "threatened" signal analogous to the existing "shared" signal. A transaction writing an address uses this signal to warn another transaction reading the address.

4.7   HTM SEMANTICS

Papers so far focused on implementation techniques and hardware mechanisms for HTMs. This section discusses two papers that describe possible interfaces for HTMs. The first paper is by Moss and Hosking (Section 4.7.1). It describes a framework for reasoning about nested transactions, both open and closed, and develops a set of semantic rules for nesting. The paper centers the discussion around a set of hardware structures to allow transactions to discover nesting relationships. The second paper is by McDonald et al. (Section 4.7.2). It describes a collection of instructions that provide software with greater control over the HTM policies and implementation. It also describes how these instructions help software to perform compensation, two-phase commit and coordination, and open and closed nesting. This paper also describes a cache implementation to support open and closed nesting directly in hardware.

These papers do not describe a complete TM system, and focus on few key aspects of a TM system.

4.7.1   Moss and Hosking, SCP 2006

Overview

In this paper, Moss and Hosking use a hardware model to describe a framework to reason about nested transactions [36]. HTMs so far supported flattened nesting where nested transactions were subsumed into the outermost transaction. The paper describes hardware extensions to support closed and open nested transactions. Closed nested transactions minimize roll back when a transaction conflicts and aborts. Open nested transactions increase concurrency when executing large transactions.

The paper does not describe a programming interface and focuses primarily on models to reason about nested transactions. While the model describes nesting for hardware, the concepts are also applicable to software systems. To increase concurrency for open nested transactions, the paper proposes an open nested semantic, in which an open nested transaction, when it commits, it removes from its parent's read or write set any address the open nested transaction updated.

Semantic Description

The paper describes a reference model for reasoning about nesting. We do not reproduce its formal descriptions of open nesting semantics, but informally touch upon it. The paper

differentiates open and closed nesting (see Chapter 2) only in terms of their behavior when a transaction commits. If an open transaction commits, then its read set is discarded and its writes are merged into the top-level (i.e., the outermost) transaction. It also removes the written data elements from the read and write set of all other transactions. Compensation action is necessary if a parent subsequently aborts.

Implementation Sketch

The paper does not describe an HTM system. Its focus primarily is on representing nested transactions and sketching a hardware implementation to support nesting. The paper frames the discussion around parent and child transactions and sketches various models. We describe one model and refer the reader to the paper for other models.

The paper uses three table structures to model transactional state, including nesting relationships. These tables also have an in-memory representation and the processor locally caches these structures. Processors use these tables to identify nesting relationships between transactions. The *transaction data cache* is fully associative and maintains the transactional state and data for memory locations accessed in the transactions. The fields are shown below:

TID	Address	Data	Valid	Dirty	Written	Ancestor Written	From TID	Write/ RCount

The `TID` identifies the transaction that owns this location and a zero value means no transaction owns this location. The `address` field records the address of the cache location and data field records the data. If the `dirty` field is set, then it means this entry does not exist in the in-memory representation of the table. The `written` field tracks if the transaction wrote this location. The remaining fields track information for nesting. The `ancestor written` field indicates if some ancestor of this transaction has a written copy of this location. The `from TID` field indicates the transaction from which this value was read. The `write` field records the `TID` of any descendant that read and wrote this location. The `rcount` field records the number of live descendants that read this copy.

The *transaction parent cache* tracks the child and parent pairs for live transactions. This table can be used to discover the `TID` of a transaction's parent. The cache can drop clean entries silently but must flush dirty entries to main memory. The fields are shown below:

Parent	Child	Valid	Dirty	OV	Open

The `parent` field tracks the TID of the transaction's parent and the `child` field tracks the child transaction's TID. The `valid` field marks the entry as valid. If the `dirty` field is set, then this entry does not exist in the in-memory representation of the table. If the transaction overflows the transaction data cache into memory, then the `OV` field is set. If the transaction is an open nested transaction, then the `open` field is set.

The *overflow summary table* is a memory-resident bit vector to track overflowed addresses. It is indexed using a hash of the address.

The paper describes many scenarios. We discuss one where a processor performs a read or write operation inside a transaction. A transaction first obtains a free TID on creation and allocates an entry in the transaction's parent cache, marking its entry `valid` and `dirty`. When the transaction reads or writes a location, the processor looks up the transaction data cache. If it misses, then it checks the overflow summary table. If the entry does not exist in the table, then the processor determines the parent of the transaction and looks up its entry. If the parent lookup also does not return an entry for this location, the system creates two new entries: one for a transaction with a TID as zero representing committed state for the location and another for the current transaction. If the parent lookup returns an entry, then the system only creates an entry for the current transaction in the transaction data cache with the data returned from the ancestor. The fields are initialized to reflect the ancestor information.

If a location has no ancestor or has an ancestor with only a reader count, then a read to this location will have no conflict. If the ancestor has a write TID field, then a read would conflict with the writer. A write access conflicts only if the cache has an unwritten entry and the ancestor has a reader count greater than 1, or if there is no entry and the youngest ancestor has a reader count greater than 0 or has a write TID associated.

The operations described are complex for hardware to implement. To simplify the implementation, the authors propose not tracking ancestors in the data cache. The paper also describes operations to abort and commit nested transactions. While the paper presents a sketch of various schemes, it provides a model to reason about nested transactions, both in hardware and in software.

4.7.2   McDonald et al., ISCA 2006

Overview

This paper describes instruction set extensions for HTMs. These extensions allow programmers to specify rich transactional memory semantics to an HTM [34]. The extensions include support to perform a two-phase commit of a transaction, to execute closed and open nested transactions, and to invoke software handlers when a transaction commits, aborts, or has a data conflict. The extensions are intended as a set of primitives for software to use flexibly, and the extensions do not dictate an implementation.

The paper introduces a transactional control block (TCB), a memory-resident data structure. The TCB captures information about an executing transaction. The paper also describes extensions to architectural register state and the implementation of nested transactions using extensions to hardware caches and the implementation of two-phase commit. The paper does not describe the implementation of all the new instructions. The paper describes the implementation challenges for supporting rich nesting in caches.

Programming Interface

The paper provides programmers with a transaction control block (TCB) and new architectural registers to maintain HTM transaction state. The TCB records a transaction's status word, the register checkpoint, the read and write set, and data state in the form of an undo log or a write buffer. The TCB also records a stack of software handlers for commits, aborts, and data conflict violations. Abort, commit, and violation handlers allow software to perform specialized actions. Software can register these handlers. The system invokes commit handlers in the order they were registered, and invokes violation handlers when a data conflict occurs. Violation handlers execute as part of the transaction but use open nesting to access shared state. The application uses these handlers to execute compensation code or to perform software contention management.

The location of the current TCB is stored in the `xtcbptr_base` and `xtcbtr_top` registers. The software handler code address is stored in the `xchcode` (for commit), `xvhcode` (for data conflict violation), and `xahcode` (for abort) registers. The `xstatus` register records a transaction's state, its identifier, if it is an open or closed transaction, if it has aborted or committed, and its nesting level. The `xvPC` and `xvaddr` registers record the program counter when a violation or abort occurred. Since violations may occur at any transaction level, the `xvcurrent` and `xvpending:` registers record a bit mask, one bit per nesting level, for the current violation and any pending violations.

The paper also provides an extensive set of new instructions:

1. `xbegin`, `xbegin_open`. These instructions begin a closed or open nested transaction.
2. `xvalidate`. This instruction checks the validity of the read set of the transaction, and if successful, marks the transaction as validated. After a successful validation, the transaction cannot abort due to a prior conflict. The transaction may however abort voluntarily.
3. `xcommit`, `xabort`. The `xcommit` instruction commits the current transaction. The `xabort` instruction aborts the transaction, jumps to the abort handler, and disables further violation reporting.

4. `xrwsetclear`. This instruction discards the current read and write sets. It also clears the bit in the violation bit mask for the current nesting level.

5. `xregrestore`. This instruction restores the register checkpoint.

6. `xvret, xenviolrep`. The `xvret` instruction returns from the abort or violation handler and enables violation reporting. The `xenviolrep` instruction enables violation reporting.

7. `imld, imst, imstid`. The `imld` and `imst` instructions perform loads without adding to the read set and stores without adding to the write set. It, however, maintains undo information for the store. The `imstid` instruction stores without adding to write set and without maintaining undo information.

8. `release`. This instruction removes an address from the current read set.

The `xvalidate` and `xcommit` instructions allow implementation of two-phase commit. The code sequence between these two instructions operates as part of the transaction (and has access to the transaction's speculative state) but is not protected against shared data conflicts. Open nesting is necessary to avoid data conflicts between these two phases. The authors propose using these instructions to perform IO and system calls and for coordinated commits.

The paper describes an alternative to Moss and Hosking's definition of open nesting (Section 4.7.1). Moss and Hosking suggest that when an open nested transaction commits, it removes from its ancestor's read or write set any address the transaction updated. The paper argues that this is a nonintuitive semantic, and that the address should not be removed from the ancestor's read or write set.

Implementation

The paper describes the behavior of the `xvalidate` instruction. If an HTM performs early acquisition of exclusive ownership and early conflict detection, the `xvalidate` instruction blocks execution until all prior loads and stores complete execution without conflicts. The hardware acquires all ownerships before the `xvalidate` instruction's completion. If an HTM performs late acquisition of exclusive ownership and late conflict detection, the `xvalidate` instruction starts to acquire ownerships for appropriate cache lines or to arbitrate for a global commit protocol.

The paper describes two cache implementations to support nesting directly in hardware. In the first implementation, the cache line is extended to track read and write sets for multiple nested transactions. In the described implementation, the cache line supports nesting depth of

four. The cache-line extensions for the first implementation are shown below. The read and write sets are identified by $R_i W_i$ bits for a nesting level i:
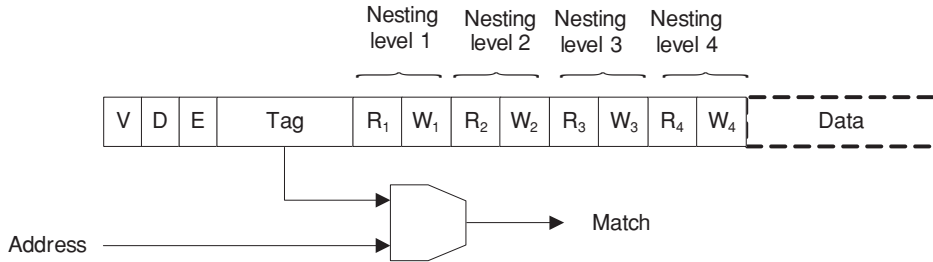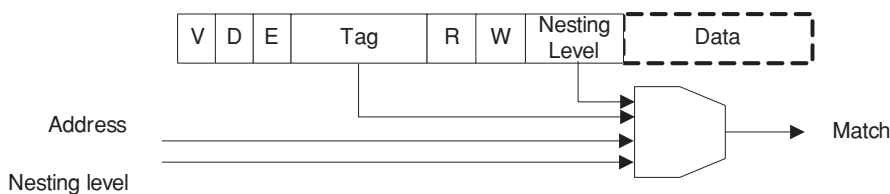

In this implementation, data in a cache line corresponds with the innermost nesting level. Therefore, before a nested transaction writes the line, it must copy the older value into an undo log. If the cache receives a read request from another processor, the cache checks all R and W bits in parallel and if a conflict is detected, it is propagated to the violation xvpending and xvcurrent bit masks.

When a closed nested transaction commits, the undo log entries of the committing nested transaction are appended to the parent transaction's undo log. The R and W bits for the committing transaction must be logically ORed into the R and W bits of the parent transaction. The paper describes a lazy scheme to implement this operation where the merge occurs when the processor next accesses the cache line. During this lazy update, the two unmerged bits are treated as one for conflicts. Another nesting level cannot start until the merge is completed. An open nested transaction commit clears the transaction's R and W bits in the cache.

When a closed nested transaction aborts, the transaction's R and W bits are cleared and the undo log is rolled back. If a transaction aborts and it had an open nested transaction that committed, then the parent's data might have been overwritten by the open nested transaction. In this case, care is needed to make sure that the parent does not roll back the open nested transaction's updates.

The second implementation uses different cache lines in the same cache set to track the same address for nested transactions. This scheme does not extend a cache line to have multiple R and W bits. The fields are shown below:

Here, each line is extended with one set of R and W bits and a nesting level field. The nesting level records the transaction nesting depth corresponding with this line. In this implementation, a cache lookup can return multiple hits to cache lines in the same set and requires an additional check to determine the most recent version. When a new nesting level accesses a line with a nesting level field set to some value, a new line is allocated, data from the old line is copied into the new line, and the nesting level field of the new line is updated.

A commit of a closed nested transaction at nesting level i changes all cache lines with level i to i − 1. If i − 1 entry already exists, its read-set and write-set information merges into the older entry. Then the cache discards this entry. For an open nested commit, all entries in i change to zero. If there are more versions, their data updates to that of nesting level i. On an abort operation at nesting level i, all R_i and W_i bits are flash cleared.

REFERENCES

[1]   Cathy May, Ed Silha, Rick Simpson, and Hank Warren, *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. San Francisco, CA: Morgan Kaufman, May 1994.

[2]   S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Comput.*, Vol. 29(12), pp. 66–76 Dec. 1996.

[3]   C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson and S. Lie, "Unbounded transactional memory," In *Proc. 11th Int. Symp. on High-Performance Computer Architecture*, pp. 316–327 Feb. 2005.

[4]   P. A. Bernstein, V. Hadzilacos and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Reading, MA: Addison-Wesley, 1987.

[5]   B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, Vol. 13(7), pp. 422–426 July 1970.

[6]   C. Blundell, E. C. Lewis and M. M. K. Martin, "Deconstructing transactions: The subtleties of atomicity," In *4th Annu. Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[7]   R. P. Case and A. Padegs, "Architecture of the IBM system/370," *Commun. ACM*, Vol. 21(1), pp. 73–96 Jan. 1978.

[8]   L. Ceze, J. Tuck, J. Torrellas and C. Cascaval, "Bulk disambiguation of speculative threads in multiprocessors," In *Proc. 33rd Int. Symp. on Computer Architecture*, pp. 227–238 2006.

[9]   D. Chaiken, J. Kubiatowicz and A. Agarwal, "Limitless directories: A scalable cache coherence scheme," In *Proc. 4th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pp. 224–234, 1991.

[10] A. Chang and M. Mergen, "801 storage: Architecture and programming," *ACM Trans. Comput. Syst.*, Vol. 6(1), pp. 28–50 Feb. 1988.

[11] E. G. Coffman, M. Elphick and A. Shoshani, "System deadlocks," *ACM Comput. Surv.*, Vol. 3(2), pp. 67–78 1971.

[12] *Compaq. Alpha Architecture Handbook*, version 4, Oct. 1998.

[13] IBM Corporation, *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. San Francisco, CA: Morgan Kaufman, 1998.

[14] L. Fan, P. Cao, J. Almeida and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, Vol. 8(3), pp. 281–293 2000.

[15] K. Gharachorloo, "Memory consistency models for shared-memory multiprocessors," Stanford University, 1995; also appears as Technical Report CSL-TR-95-685, Computer Systems Laboratory, Stanford University, Stanford, CA, Dec. 1995.

[16] K. Gharachorloo, A. Gupta and J. L. Hennessy, "Two techniques to enhance the performance of memory consistency models," In *Proc. 1991 Int. Conf. on Parallel Processing*, pp. 355–364 Aug. 1991.

[17] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," In *Proc. 10th Annu. Int. Symp. on Computer Architecture*, pp. 124–131 1983.

[18] J. R. Goodman and P. J. Woest, "The Wisconsin multicube: A new large-scale cache-coherent multiprocessor," In *Proc. 15th Annu. Int. Symp. on Computer Architecture*, pp. 422–431 May 1988.

[19] L. Hammond, M. Willey and K. Olukotun, "Data speculation support for a chip multiprocessor," In *Proc. 8th Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 58–69 Oct. 1998.

[20] 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 *Proc. 31st Annu. Int. Symp. on Computer Architecture*, pp. 102–113 June 2004.

[21] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Francisco, CA: Morgan Kaufman, 1995.

[22] M. Herlihy, Personal Communication, 2006.

[23] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," In *Proc. 20th Annu. Int. Symp. on Computer Architecture*, pp. 289–300 May 1993.

[24] M. D. Hill, "Multiprocessors should support simple memory consistency models," *IEEE Comput.*, Vol. 31(8), pp. 28–34 Aug. 1998.

[25] E. H. Jensen, G. W. Hagensen and J. M. Broughton, "A new approach to exclusive data access in shared memory multiprocessors," Lawrence Livermore National Laboratory, Technical Report UCRL-97663, Nov. 1987.

[26] G. Kane and J. Heinrich, *MIPS RISC Architecture*. Upper Saddle River, NJ: Prentice-Hall, 1992.

[27] T. F. Knight, "An architecture for mostly functional languages," In *Proc. ACM Lisp and Functional Programming Conference*, pp. 105–112 Aug. 1986.

[28] S. Kumar, M. Chu, C. J. Hughes, P. Kundu and A. Nguyen, "Hybrid transactional memory," In *Proc. 11th ACM* SIGPLAN *Symp. on Principles and Practice of Parallel Programming*, pp. 209–220 2006.

[29] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, Vol. 21(6), pp. 558–565 July 1978.

[30] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. Comput.*, Vol. 28(9), pp. 779–782 Sept. 1979.

[31] J. Laudon and D. E. Lenoski, "The SGI Origin: A ccNUMA highly scalable server," In *Proc. 24th Annu. Int. Symp. on Computer Architecture*, pp. 241–251 June 1997.

[32] S. Lie, "Hardware support for unbounded transactional memory," Masters Thesis, Massachusetts Institute of Technology, May 2004.

[33] J. F. Martinez and J. Torrellas, "Speculative synchronization: Applying thread-level speculation to explicitly parallel applications," In *Proc. 10th Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 18–29 Oct. 2002.

[34] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis and K. Olukotun, "Architectural semantics for practical transactional memory," In *Proc. 33rd Int. Symp. on Computer Architecture*, pp. 53–65 2006.

[35] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill and D. A. Wood, "LogTM: Log-based transactional memory," In *Proc. 12th Int. Symp. on High–Performance Computer Architecture*, pp. 254–265 Feb. 2006.

[36] J. E. B. Moss and A. L. Hosking, "Nested transactional memory: Model and architecture sketches," In *Science of Computer Programming,* Vol. 63(2), pp. 186–201, Dec. 2006.

[37] G. Radin, "The 801 minicomputer," In *Proc. 1st Int. Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 39–47 1982.

[38] R. Rajwar and J. R. Goodman, "Speculative lock Elision: Enabling highly concurrent multithreaded execution," In *Proc. 34th Int. Symp. on Microarchitecture*, pp. 294–305 Dec. 2001.

[39] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," In *Proc. 10th Symp. on Architectural Support for Programming Languages and Operating Systems*, pp. 5–17 Oct. 2002.

[40] R. Rajwar, M. Herlihy and K. Lai, "Virtualizing transactional memory," In *Proc. 32nd Annu. Int. Symp. on Computer Architecture*, pp. 494–505 June 2005.

[41] D. J. Rosenkrantz, R. E. Stearns and P. M. Lewis, "System level concurrency control for distributed database systems," *ACM Trans. Database Syst.*, Vol. 3(2), pp. 178–198 June 1978.

[42] P. Rundberg and P. Stenstrom, "Reordered speculative execution of critical sections," *Int. Conf. on Parallel Processing*, 2002.

[43] A. Shriram, 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 *1st ACM Sigplan Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[44] J. E. Smith and G. Sohi, "The microarchitecture of superscalar processors," *Proc. IEEE*, Vol. 83, pp. 1609–1624 Dec. 1995.

[45] G. S. Sohi, S. E. Breach and T. N. Vijaykumar, "Multiscalar processors," In *Proc. 22nd Annu. Int. Symp. on Computer Architecture*, pp. 414–425 June 1995.

[46] J. G. Steffan, C. B. Colohan, A. Zhai and T. C. Mowry, "A scalable approach to thread-level speculation," In *Proc. 27th Annu. Int. Symp. on Computer Architecture*, pp. 1–12 June 2000.

[47] J. M. Stone, H. S. Stone, P. Heidelberger and J. Turek, "Multiple reservations and the Oklahoma update," *IEEE Concurrency*, Vol. 1(4), pp. 58–71 Nov. 1993.

[48] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the IEEE Futurebus," In *Proc. 13th Annu. Int. Symp. on Computer Architecture*, pp. 414–423 June 1986.

[49] C. Zilles and L. Baugh, "Extending hardware transactional memory to support non-busy waiting and non-transactional actions," In *1st ACM Sigplan Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[50] C. Zilles and D. Flint, "Challenges to providing performance isolation in transactional memories," In *Proc. 4th Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

CHAPTER 5

Conclusions

Transactional memory (TM) is a popular research topic. The advent of multicore computers and the absence of an effective and simple programming model for parallel software have encouraged a belief (or hope) that transactions are an appropriate mechanism for structuring concurrent computation and facilitating parallel programming. The underpinning of this belief is the success of transactions as the fundamental programming abstraction for databases, rather than practical experience applying transactions to general parallel programming problems. Confirmation of this belief can come only with experience. The research community has started exploring the semantics and implementation of transactional memory, to understand how this abstraction can integrate into existing practice, to gain experience programming with transactions, and to build implementations that perform well.

This book surveys the published research in this area, as of early summer 2006. As is normal with research, each effort solves a specific problem and starts from strong, though often narrow, assumptions about how and where the problem arises. Software transactional memory (STM) papers offer a very different perspective and start from very different assumptions than hardware transactional memory (HTM) papers. Nevertheless, there are common themes and problems that run through all of this research. In this survey, we have tried to extract the key ideas from each paper and to describe a paper's contribution in a uniform context. In addition, we have tried to unify the terminology, which naturally differs among papers from the database, computer architecture, and programming language communities.

Looking across the papers in this book, a number of common themes emerge.

First, transactional memory is a new programming abstraction. The idea evolved from initial proposals for libraries and instructions to perform atomic multiword updates to modern language constructs such as `atomic{ }`, which allow arbitrary code to execute transactions. Moreover, researchers have elaborated the basic concept of a transaction with mechanisms from conditional critical sections such as `retry` and mechanisms to compose transactions such as `orElse`. These language features lift a programmer above the low-level mechanisms that implement TM, much as objects and methods build on subroutine call instructions and stacks.

Control over the semantics and evolution of the programming abstractions belongs in large measure to programming language designers and implementers. They need to invent new

constructs and precisely define their semantics; integrate transactional features into existing languages; implement the language, compiler, and run-time support for transactions; and work with library and application developers to ensure that transactions coexist and eventually support the rich environment of modern programming environments.

This level of abstraction is what we loosely refer to as the semantic layer of transactional memory. Programmers will think at this level and write their applications with these abstractions. A key challenge is to ensure that these constructs smoothly coexist with legacy code written, and perhaps even compiled, long before TM. This code is valuable, and for the most part, it will not be rewritten. Building a system entirely around transactions is a long-term endeavor. Transactions will coexist with nontransactional code for a long time. It is also important that transactional memory systems provide features and programming conventions suitable for multiple languages and compilers, so a construct written in one language and compiled with one compiler interoperates with code written in other languages and compiled with other compilers. A further challenge is to learn how to program with transactions and to teach the large community of programmers, students, and professors a new programming abstraction.

Second, high-performance software implementations of transactional memory will play an important role in the evolution of transactional memory, even as hardware support becomes available. STM offers several compelling advantages. Because of their malleability, STM systems are well suited to understanding, measuring, and developing new programming abstractions and experimenting with new ideas. Until we measure and simulate "real" systems and applications built around transactions, it will be difficult to make the engineering tradeoffs necessary to build complex HTM support into processors. Changing a processor architecture has implications on engineering, design, and validation costs. Further, instruction-set extensions need to be supported for an extended period, often the lifetime of an architecture.

Moreover, since STM systems run on legacy hardware, they provide a crucial bridge that allows programmers to write applications using TM long before hardware support or HTMs are widely available. If STM systems do not achieve an acceptable and usable level of performance, most programmers will wait until computers with HTM support are widely available before revamping their applications or changing their programming practice to take advantage of the new programming model.

Third, STM and HTM complement each other more than they compete. It is easy to view these two types of implementations as competitors, since they both solve the same problem by providing operations to implement transactional memory. Moreover, the respective research areas approach problems from opposite sides of the hardware–software interface and often appear to be moving in opposite directions. However, each technique has strengths that complement the weaknesses of the other. Together they are likely to lead to an effective TM system that neither could achieve on its own.

A key advantage of STM systems is their flexibility and ability to adapt readily to new algorithms, heuristics, mechanisms, and constructs. The systems are still small and simple (though they are often integrated into more complex platforms, such as Java or .NET run-times), which allows easy experimentation and rapid evolution. In addition, manipulating TM state in software facilitates integration with garbage collection, permits long-running, effectively unbounded transactions, and allows rich transaction nesting semantics, sophisticated contention management policies, natural exceptions handling, etc. Moreover, the tight integration of STMs with languages and applications allows cross-module optimization, by programmers and compilers, to reduce the cost of a transaction.

STMs also have limitations. When STMs manipulate metadata to track read and write sets and provision for rollback, they execute additional instructions, which increases the overhead in the memory system and instruction execution. Executing additional instructions also increases power consumption. Moreover, STMs have not yet discovered an economical way of providing strong isolation. Unmanaged and unsafe languages (such as C and C++) offer few safety guarantees, and thus constrain compiler analysis and limit STM implementation choices in ways that may lead to lower performance or difficult-to-find bugs. STMs also face difficult challenges in dealing with legacy code, third-party libraries, and calls on functions compiled outside of the STM.

HTM systems have a different collection of strengths. A key characteristic of most HTMs is that they are decoupled from an application. This allows an HTM to avoid the code bloat necessary for STMs. Consequently, HTMs can execute some transactions (those that fit hardware buffers) with no more performance or instruction execution overhead than that caused by sequential execution. In addition, most HTMs support strong isolation. Furthermore, HTMs are well suited to unmanaged and unsafe code, whose fixed data layouts and unrestricted pointers constrain STM systems. HTMs can also accommodate transactions that invoke legacy libraries, third-party libraries, and functions not specially compiled for TM.

However, HTMs have limitations. Hardware buffering is limited, which forces HTMs to take special action when a transaction overflows hardware limits. Some HTMs spill state into lower levels of the memory hierarchy, while others use STM techniques and spill to or log to software-resident TM metadata structures. Since hardware partially implements the mechanisms, it does not always offer the complete flexibility of an STM system. Early HTMs implemented contention management in hardware, which required the policy to be simple. Recent HTMs have moved contention management into software. Finally, HTMs have no visibility into an application. Optimization belongs to a compiler and is only possible through the narrow window of instruction variants.

While STMs and HTMs both have advantages and limitations, the strength of an STM is often a limitation in an HTM, and an HTM often excels at tasks STMs find

difficult to perform efficiently. We see three obvious ways in which these two fields can come together.

The STM community can identify instruction set extensions and hardware mechanisms to accelerate STM systems. Obvious areas are the high cost of tracking read, write, and undo sets and detecting conflicts between transactions. Such support must be general enough to aid a wide variety of STM systems. Hardware mechanisms identified in recent proposals (Section 4.6) retain much of the complexity of a typical HTM but provide finer control over the hardware. The challenge here is to design mechanisms that integrate into complex, modern processors and provide architecturally scalable performance over time. Other approaches are necessary to give STM the same transparency that HTM can achieve for strong isolation, legacy code, third-party libraries, and perhaps for unsafe code.

The hardware community can continue to develop self-contained HTM systems, which rely on software to handle overflows of hardware structures and to implement policy decisions. This approach preserves the speed and transparency of HTM, but may not have the flexibility of a software system, unless the interfaces are well designed. Experimentation with STM systems may help to divide the responsibilities appropriately between software and hardware and to identify policies that perform well in a wide variety of situations and that can be efficiently supported by hardware. The challenges are to ensure that aspects of the TM system that are not yet well understood are not cast into hardware prematurely and to ensure that the system can be integrated into a rich and evolving software environment.

The final approach is to combine the strengths of the two approaches into a hybrid hardware–software TM system that offers low overhead, good transparency, and flexible policy. The contributions of the STM community can include software definition of the metadata structures and their operations, software implementation of policy, and close integration into compilers and run-time. The HTM community can contribute strong isolation, support for legacy software, TM-unaware code, and low-performance penalties and overheads.

TM holds promise to simplifying the development of parallel software as compared to conventional lock-based approaches. It is yet unclear how successful it will be. Success depends on surmounting a number of difficult and interesting technical challenges, starting with providing a pervasive, efficient, well-performing, flexible, and seamless transactional memory system that can be integrated into existing execution environments. It is not clear what such a system would look like, but this book describes a fountain of research that may provide some of the ideas.

Biography

James Larus is a Research Area Manager for programming languages and tools at Microsoft Research. He manages the Advanced Compiler Technology, Human Interaction in Programming, Runtime Analysis and Design, and Software Reliability Research groups and co-lead the Singularity research project. He joined Microsoft Research as a Senior Researcher in 1998 to start and, for five years, lead the Software Productivity Tools (SPT) group, which was one of the most innovative and productive groups in the areas of program analysis and programming tools. Before joining Microsoft, Larus was an Assistant and Associate Professor at the University of Wisconsin-Madison, where he co-led the Wisconsin Wind Tunnel research project with Professors Mark Hill and David Wood. This DARPA and NSF-funded project investigated new approaches to simulating, building, and programming parallel shared-memory computers. In addition, Larus's research covered a number of areas: new and efficient techniques for measuring and recording executing programs' behavior, tools for analyzing and manipulating compiled and linked programs, programming languages, tools for verifying program correctness, compiler analysis and optimization, and custom cache coherence protocols. Larus received a PhD from the University of California at Berkeley in 1989, and an AB from Harvard in 1980.

Ravi Rajwar is a researcher in the Corporate Technology Group at Intel Corporation. His research interests include theoretical and practical aspects of computer architecture. Most recently, he has investigated resource-efficient microprocessors and architectural support for improving programmability of parallel software. Rajwar received a PhD from the University of Wisconsin-Madison in 2002, a MS from the University of Wisconsin-Madison in 1998, and a BE from the University of Roorkee, India, in 1994, all in Computer Science.