Deconstructing Transactional Semantics: The Subtleties of Atomicity

Colin Blundell E Christopher Lewis Milo M. K. Martin
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, Pennsylvania USA
{*blundell,lewis,milom*}*@cis.upenn.edu*

Abstract

Researchers have recently proposed software and hardware support for transactions as a replacement for the traditional lock-based synchronization most common in multithreaded programs. Transactions allow the programmer to specify a region of the program that should appear to execute atomically, while the hardware and runtime system optimistically execute the transactions concurrently to obtain high performance. The transactional abstraction is thus a promising approach for creating both faster and simpler multithreaded programs.

Although transactions have great potential for simplifying multithreaded programming due to their strong atomicity guarantees, this work shows that these same guarantees can have unexpected and potentially serious negative effects on programs that were written assuming weaker synchronization primitives. We make three contributions: (1) we show that a direct translation (statically or dynamically) of lock-based critical sections into transactions can introduce deadlocks into otherwise correct programs, (2) we define an atomicity model for transactions, in which we introduce the terms strong and weak atomicity, and (3) we show that the decision to enforce strong atomicity as opposed to weak atomicity can also result in deadlock. These results invalidate the intuitive idea that transactions are strictly safer than lock-based critical sections and strong atomicity is strictly safer than weak atomicity. We assert that the research community must confront these subtle issues of transactional semantics by exploring the design space and deciding upon the most appropriate semantics for future transactional systems.

1 Introduction

Synchronization has a first-order impact on the correctness and performance of multithreaded programs, and locks are currently the prevailing synchronization mechanism. Locks guard regions of code called critical sections, preventing concurrent access to shared data structures. Unfortunately, effective programming with lock-based synchronization is a delicate balancing act between achieving high performance and maintaining correctness. Coarse-grained locking (*e.g.*, one lock for an entire binary tree data structure) gives rise to simple programming paradigms. However, the resulting lock contention can significantly limit scalability and performance. Fine-grained locking (*e.g.*, a lock per node in a binary tree) can reduce lock contention but adds programming complexity, increasing the potential for deadlock and subtle data-race bugs. Furthermore, fine-grained locks may actually increase overhead due to more frequent acquire and release operations, which are slow on modern processors. To further complicate matters, programmers must often choose between various lock implementations (*e.g.*, simple spin locks for uncontended locks, contention-robust queue-based locks, per-thread reentrant locks, and reader/writer locks).

In response to the performance and complexity challenges of locks, researchers have proposed hardware support for synchronization via *transactions* [2, 6, 7, 10, 12, 14, 15]: segments of code that are atomic with respect to each other. Like lock-based critical sections, transactions are a mechanism for mutual exclusion, but transactions are simpler (specifying atomicity without naming a lock) and more efficiently implemented (optimistically executing concurrently, rolling back on dynamically detected inter-transaction conflicts). This combination of intuitive interface and efficient implementation has the potential to solve many lock-related problems. Returning to our binary tree example, encapsulating an entire binary tree operation in a transaction has (1) the simplicity of an implementation that uses coarse-grained locking, (2) the scalable performance of an implementation that uses fine-grained locking, and (3) the efficiency of avoiding fine-grained locking overheads. In addition, efficient implementations of transactions have the potential to replace all the various types of locks (*e.g.*, spin locks, queue-based locks, reader/write locks) with a single powerful primitive. Finally, transactions can provide a simpler approach for creating wait-free and non-blocking data structures [10]. We elaborate on the various transactional proposals in Section 2.

Although transactions have great potential, this work un-

covers subtle issues and common misconceptions about their semantics. In particular, we investigate the implications of different forms of atomicity. We show that in several circumstances, correct programs created assuming one form of atomicity (*e.g.*, that of lock-based critical sections) can deadlock when run on a system supporting a stronger form of atomicity (*e.g.*, that of transactions). We make three main contributions:

- **We show that a direct translation (statically or dynamically) of lock-based critical sections into transactions can introduce deadlock into an otherwise correct program.** That is, it is unsafe to simply begin a transaction at lock acquisition and end it at lock release. At least one proposal advocates this direct conversion of lock-based programs to transaction-based programs. In Section 3 we use an example to show that such a transformation is not safe in general.
- **We define two atomicity models for transactional systems.** Researchers have implemented differing policies for handling interactions between transactional and non-transactional code; however, current definitions of transactional semantics do not always explicitly express these differences. We define strong atomicity as a transactional semantics that guarantees atomicity between transactions and non-transactional code, and we define weak atomicity as a transactional semantics that guarantees atomicity only among transactions. We assert that a transactional system should specify its atomicity model explicitly, much as a shared memory multiprocessor specifies a memory consistency model to define memory ordering [1]. In Section 4 we further discuss these atomicity models.
- **We show that a program that is correct under the weak atomicity model may deadlock under the strong atomicity model.** The intuitive view that a stronger atomicity model will correctly execute a superset of the code that is correct under a weaker atomicity model is false. In Section 5 we show by example that strong atomicity can negatively affect program correctness.

This work invalidates the intuitive and commonly-held view that transactions are strictly safer than lock-based critical sections and that strong atomicity is strictly safer than weak atomicity. Stronger atomicity restricts the set of legal program interleavings, which would seem to only help the programmer by removing (potentially) buggy interleavings. However, programmers may intentionally or unintentionally exploit the non-atomicity of critical sections guarded by different locks or the non-atomicity between transactions and non-transactional statements, producing programs that *require* concurrent execution of these regions to avoid deadlock.

2 Background on Transactions

Transactions have been used both as an efficient synchronization primitive and as an approach for optimistic execution of lock-based critical sections.

2.1 Synchronization Primitives

Researchers have proposed both hardware and software support for transactions and transaction-like synchronization primitives. These proposals are distinguished by several aspects: their policy for mediating conflicts between transactions and non-transactional code, whether they have at-most-once or exactly-once execution semantics, the maximum size of a transaction, and whether the programmer must access memory differently within a transaction than outside one. Below, we survey hardware and software transactional systems, focusing on the above aspects.

Supporting Transactions in Hardware. Herlihy and Moss [10] propose transactional memory as a means of supporting lock-free data structures. Inspired by database transactions, they define a transaction as a sequence of instructions that are atomic and serializable with respect to other transactions (*i.e.*, transactions see each other's changes atomically and there is some serial commit order observed by all processors). In their system, transactions execute speculatively and roll back upon detecting a conflict (*i.e.*, an inter-transaction data race). Speculative transactional state is buffered in caches, and conflict detection is implemented as an extension to standard multiprocessor cache coherence protocols. This general implementation strategy has been followed by most subsequent hardware proposals. Herlihy and Moss do not prescribe a semantics to conflicts between a transaction and non-transactional code. Another notable aspect of their programmatic interface is that transactions have at-most-once semantics: although the system causes transactions to abort on conflicts, the programmer is responsible for detecting the abort and retrying the transaction. The maximum size of a transaction is implementation specific, because fixed-size hardware caches buffer speculative memory updates until transactions are committed.

A major limitation of Herlihy and Moss' original proposal is that transactional state is buffered in caches, so programmers must be aware of cache size and application cache behavior when constructing transactions. Resent work has addressed this limitation. Ananian et al.'s Unbounded Transactional Memory (UTM) [2] supports transactions that can be as large as virtual memory and persist through interrupts; this system, however, requires substantial changes to the processor and memory system. Therefore, they also propose Large Transactional Memory (LTM), which supports transactions that can be as large as physical memory but cannot survive interrupts;

LTM requires only minor changes to existing cache designs and coherence protocols. Interestingly, UTM allows transactions to interleave arbitrarily with non-transactional code, while LTM forces a transaction to abort on a conflict with non-transactional code. Virtual Transactional Memory (VTM) [15] and Thread-level Transactional Memory (TTM) [12] go a step further and locate memory resident speculative-state buffers in the application's virtual address space, thus tying transactions to threads/applications instead of processors. The hardware and operating system requirements for these systems are far from modest.

In Stanford's Transactional Coherence & Consistency (TCC) [6, 7], the transaction is the basic unit of parallel work, communication, memory coherence, and memory reference consistency. Unlike the above systems, all code must reside in some transaction. Furthermore, rather than leverage the cache coherence protocol, TCC defines a transaction-grained coherence and synchronization protocol. TCC supports transactions of unbounded size not by spilling overflowed hardware buffers to memory, but instead an overflowing transaction acquires permission to commit before it has actually completed; once it has this permission, it no longer needs to buffer. The same technique allows I/O to appear in transactions. A unique aspect of TCC's programming model is that it allows the programmer to explicitly specify an ordering among transactions, supporting a form of thread-level-speculative-style parallelization.

Supporting Transactions in Software. Researchers initially investigated transactional support in software for the purpose of building lock-free and non-blocking data structures. For this purpose, Shavit and Touitou [16] propose Software Transactional Memory (STM). STM supports only transactions whose data sets are statically known. Herlihy et al. [9] propose Dynamic Software Transactional Memory (DSTM), which supports transactions that access dynamic sets of memory locations. In DSTM, however, the programmer must access any object in a transaction by explicitly "opening" a transactional version of that object. In contrast to these proposals, Harris and Fraser [8] propose an STM to support a general-purpose atomic construct, similar in spirit to transactions. A programmer can make ordinary memory references within an atomic region, which has exactly-once semantics. Unfortunately, the performance overheads appear to be high when compared to hardware proposals. In a complementary line of research, Flanagan et al. [3, 4, 5] propose type systems that can statically verify atomicity of lock-based critical sections. These type systems could potentially reduce the work that a transactional memory system must do to dynamically guarantee atomicity.

2.2 Optimistic Execution of Critical Sections

Transactions may also be used as a basis for improving the implementation of lock-based critical sections. Transactional Lock Removal (TLR) [14] is a system that increases performance by dynamically converting lock-based critical sections to transactions. TLR is an extension of Speculative Lock Elision [13], which dynamically elides lock acquires by executing critical sections speculatively (buffering the results in hardware) and acquiring locks on conflicts. The goal of TLR is to avoid lock acquisition even on conflicts. To this end, TLR uses timestamps to provide an ordering that is used to resolve conflicts without acquiring locks. TLR has no explicit policy for mediating conflicts between transactions and non-transactional code, but the designers suggest extensions that make transactions atomic with respect to non-transactional code. Although TLR tries to avoid acquiring the lock, it must revert to acquiring the lock when buffer resources are exhausted or a transaction exceeds a scheduling quantum.

In a similar spirit, Welc et al. [17] define transactional monitors for Java. However, the name is somewhat of a misnomer, because transactional monitors have the same semantics as ordinary Java monitors, with the difference being that they execute speculatively (buffered in software) and roll back on conflicts between critical sections guarded by the same monitor. This per-monitor conflict detection behavior is an important difference between the two proposals, as TLR converts critical sections into transactions that are atomic with respect to all other transactions.

3 Critical Sections $\neq$ Transactions

As transactions are a promising replacement for lock-based critical sections, some transactional proposals [2, 12, 14] have extended the benefits of transactional systems to legacy lock-based programs by directly converting lock-based critical sections to transactions (replacing lock acquires and releases with transaction begin and end operations, respectively). This conversion changes the program's semantics: a critical section that was previously atomic only with respect to other critical sections guarded by the same lock is now atomic with respect to *all* other critical sections. In this section, we show that this semantic change can cause some correct lock-based programs to deadlock. As a result, such direct conversations may be acceptable for architectural studies, but a system that indiscriminately applies this direct conversion will not be backward-compatible for all legacy programs.

The assumption that lock-based critical section can be transparently translated to transactions is a natural one; the conversion simply disallows (previously legal) interleavings that contain perceivably concurrent execution of critical sections guarded by different locks, and it does not introduce any new interleavings. The disallowed interleavings would seem to be those most unintuitive to the programmer. For example, they may produce data races due to incorrect locking, in which case the conversion could actually remove

```
bool flagA = false, flagB = false;
mutex m1, m2;

proc1(){
  acquire(m1);
  while(!flagA){}
  flagB = true;
  ✪...
  release(m1);
}

proc2(){
  acquire(m2);
  flagA = true;
  while(!flagB){}
  ✪...
  release(m2);
}

P = proc1() || proc2()
```

(a)

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

proc1(){
  begin_trans();
  while(!flagA){}
  flagB = true;
  ✪...
  end_trans();
}

proc2(){
  begin_trans();
  flagA = true;
  while(!flagB){}
  ✪...
  end_trans();
}

P = proc1() || proc2()
```

(b)

Figure 1. A program with benign data races that executes correctly using locking (a) but deadlocks when directly converted to transactions (b).

bugs. However, some correct program might *require* one of these disallowed interleavings to make progress. Such a program would deadlock after the direct conversion, indicating that this conversion is not always safe.

Figure 1(a) presents a short (admittedly contrived) program that has this property. In this code, the programmer intends that neither `proc1` nor `proc2` can reach the lines marked by ✪ until the other can also reach this line (*i.e.*, effecting a barrier); the locks `m1` and `m2` each guard a (different) shared variable that is accessed in this line by `proc1` and `proc2`, respectively. Unprotected references to `flagA` and `flagB` give rise to benign data races, but one may choose to protect these variables with locks as in Figure 2. In either case, these programs operate as intended because `proc1` and `proc2` are protected by different locks, so their execution can be interleaved (assuming pre-emptive thread scheduling). Suppose we directly convert these critical sections to transactions, as shown in Figure 1(b). Now the transactions in `proc1` and `proc2` must execute atomically with respect to each other, meaning that one transaction must appear to execute before the other. This restriction allows either `proc1` to observe `proc2`'s update of `flagA` or `proc2` to observe `proc1`'s update of `flagB`, *but not both*. As a result, the program will deadlock because one or both of the transactions will be unable to make progress beyond the `while` loop. This example shows that the direct method of converting lock-based programs to use transactions may result in deadlock in legal lock-based programs by disallowing an interleaving that is necessary for progress.

This observation impacts some (but not all) previous proposals. For example, Ananian et al. [2] propose a tool that "simply replace[s] all locks with transactions," and use this tool to convert lock-based C programs for evaluation of LTM; our example shows that their tool is unsound in general. In contrast, TLR has a policy that the system reverts to acquiring the lock when a transaction exceeds its scheduling quantum. This fall-back case will result in correct execution on this example (and, we believe, in general), because the transactional deadlock will eventually cause the system to revert to lock-based execution.

We emphasize that our intent is not to exhibit a real or even necessarily realistic program on which the direct conversion is unsafe, but rather to show that it is theoretically possible for this conversion to cause deadlock. In practice, such a conversion may almost always be safe. Nonetheless, any system that translates lock-based critical sections into transactions cannot assume that this translation is always safe; it must either determine for which lock-based critical regions such a conversion is safe or have a fallback method (such as that of TLR). Determining when this direct translation can be safely applied is now an open research issue. An interesting first question is whether the direct translation of a correct program preserves partial correctness, *i.e.*, the translated program has the property that it will give a correct answer if it gives any answer. If this is true (as seems likely), then researchers can focus on detecting and preventing deadlock and livelock situations.

4 Strong versus Weak Atomicity

Transactions should clearly be atomic with respect to each other, but their relationship to non-transactional code is less clear. This ambiguity would at first appear to be merely an implementation detail, because we would expect all references to shared data to be contained within transactions. However, legal programs may contain unprotected refer-

```
bool flagA = false, flagB = false;
mutex m1, m2, mA, mB;

proc1(){                    proc2(){
  acquire(m1);                acquire(m2);
  while(true){                acquire(mA);
    acquire(mA);              flagA = true;
    if (flagA){               release(mA);
      release(mA);            while(true){
      break;                    acquire(mB);
    }                           if (flagB){
    release(mA);                  release(mB);
  }                               break;
  acquire(mB);                  }
  flagB = true;                 release(mB);
  release(mB);                }
  ✪...                        ✪...
  release(m1);                release(m2);
}                           }

      P = proc1() || proc2()
```

Figure 2. A race-free equivalent of the code in Figure 1.

ences to shared variables (*i.e.*, outside transactions) without creating malignant data races, so both transactional and non-transactional code can refer to the same data. To account for these cases, we present two *atomicity models*. We define *strong atomicity* to be a transaction semantics in which transactions execute atomically with respect to both other transactions *and* non-transactional code, and we define *weak atomicity* to be a semantics in which transactions are atomic only with respect to other transactions (*i.e.*, their execution may be interleaved with non-transactional code).

An atomicity model for a transactional system is analogous to a memory consistency model for a traditional shared memory multiprocessor. A memory consistency model defines the observable orderings of memory operations between threads [1]. A strong memory consistency model, which limits the observable reordering of memory operations, is easiest to reason about for programmers, but it is difficult to implement efficiently [11]. In contrast, a weak (or relaxed) memory consistency model, which allows for counter-intuitive reordering of memory operations, is more complex for programmers to reason about because it requires them to explicitly insert memory barriers to enforce ordering. However, weak ordering models are easier to implement efficiently. Similarly, strong atomicity provides a simple and intuitive view of transactional atomicity, which may be more difficult to implement efficiently (especially in software-based transactional systems). In contrast, weak atomicity provides a less intuitive model (as transactions may not appear atomic when interleaved with non-transactional code), but it may be easier to implement efficiently. Interestingly, as transactional systems are also shared memory systems, such systems must define *both* a transactional atomicity model and a memory consistency model, as well as any previously unconsidered interactions between the two.

Just as early work in shared memory multiprocessors did not explicitly address memory consistency issues, current work in transactional memory often does not explicitly consider the distinction between and implications of strong and weak atomicity. One model or the other is specified seemingly arbitrarily (*e.g.*, based on the published description of UTM/LTM [2], we believe that UTM provides weak atomicity, while LTM provides strong atomicity; TLR provides strong atomicity although lock-based critical sections can interleave arbitrarily with code not under a lock [14]) or the form of atomicity is left unspecified (*e.g.*, Herlihy and Moss' original definition of transactional memory semantics [10] does not fully specify how it resolves interactions between transactional and non-transactional code). TCC avoids this issue altogether because all code is contained within some transaction.

5 Code Assuming Weak Atomicity Can Break under Strong Atomicity

Another common and implicit assumption is that any program that executes correctly under weak atomicity will also execute correctly under strong atomicity. However, this assumption is not true; some programs that are correct under weak atomicity will deadlock under strong atomicity. A program executing under weak atomicity can interleave non-transactional code arbitrarily with transactional code, and such interleavings may be necessary for the program to make progress. If the system actually provides strong atomicity, these interleavings are not allowed and the program may deadlock as a result.

For example, consider the two concurrently executing procedures in Figure 3. The programmer intends that the two threads proceed in a coordinated way through the use of the shared variables `flagA` and `flagB`, effecting a barrier. Under weak atomicity, the program will execute correctly: the two threads' reads and writes can interleave arbitrarily, and the threads proceed as the programmer intended. However, consider what occurs if the program is executing under strong atomicity. The loop labeled ❶ in `proc1` will terminate only after the transaction in `proc2` propagates its update of `flagA` when the transaction commits; however, the transaction in `proc2` can commit only after the update to `flagB` (labeled ❷) executes (because of the loop labeled ❸). The resulting circular dependency causes this program to deadlock under strong atomicity, despite correctly executing under weak atomicity.

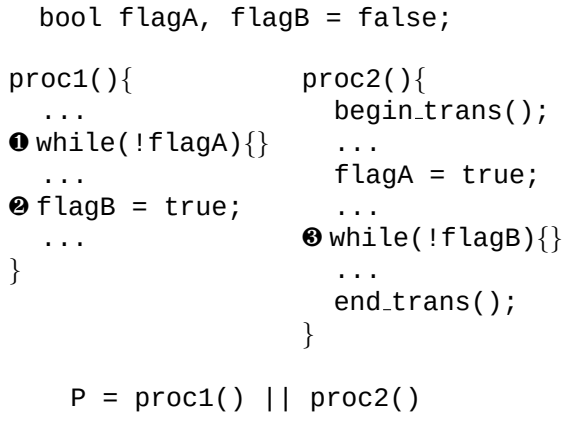

Figure 3. A program that executes correctly under weak atomicity but deadlocks under strong atomicity.

The above example illustrates the need for transactional memory systems to specify whether they are strongly atomic or only weakly atomic and then implement that semantics precisely. For example, consider that this program will deadlock on LTM but execute correctly on UTM, although the designers claim that the systems have "similar semantics." (TLR's fallback mechanism of reverting to lock acquires when a transaction exceeds its scheduling quantum will save it from deadlock on lock-based programs that are similar to this example.) If a programmer believes that a transactional system is strongly atomic and it is only weakly atomic, the programmer may write a buggy program due to race conditions between a transaction and non-transactional code (*e.g.*, a program that intermingles locks and transactions). Conversely, if a program is written with the assumption that a transactional system is weakly atomic and it in fact implements strong atomicity, the program may deadlock because it relies on transactions being non-atomic with respect to non-transactional code. As such, neither model can serve as a safe "least common denominator" target for programmers.

6 Conclusions and Open Questions

The main contribution of this paper is the counter-intuitive observation that programs that execute correctly under certain guarantees of atomicity can break when executing under stronger guarantees. Therefore, further work on transactions should consider this observation when proposing any transparent strengthening of atomicity policies. We have illustrated this situation by showing two ways in which this phenomenon can occur.

First, transactions do not strictly subsume lock-guarded critical sections in the sense that any program that works correctly with locks will work correctly when directly converted to transactions. The stronger guarantees that transactions provide result in different requirements for correct execution: locks enforce atomicity only among segments of code that are guarded by the same lock, while transactions enforce atomicity among all concurrent transactions. Hence, a program that depends on non-atomicity between critical sections guarded by different locks may break when converted to transactions.

Second, introducing atomicity between non-transactional and transactional code can break a program that correctly executes when non-transactional code can interleave with transactions. Therefore, a system must specify its policy on atomicity among transactions and non-transactional code as part of its transactional semantics. We have introduced the definitions of two transactional atomicity models. We define strong atomicity as a transactional semantics that gives the guarantee of atomicity between transactions and non-transactional code; weak atomicity is a transactional semantics that makes no such guarantee. We assert that in the future, designers of transactional memory systems should state explicitly whether they are implementing strong atomicity or weak atomicity.

These subtleties of atomicity suggest two important implications. First, users of transactions—programmers or automatic conversion tools—must be aware of the exact semantics supported by the system they are using. Anything less can lead to incorrect programs. Second, that there does not yet exist a standard semantics for transactions threatens their utility as a synchronization mechanism. If different systems provide different transactional semantics, programs will not be portable and programmers will resort to more portable primitives (*e.g.*, locks).

This paper raises several questions. We have given theoretical program examples that give rise to the problems we describe, but how often (if ever) do these types of codes arise in practice? Is it possible to build tools that determine—either statically or dynamically—when it is safe to convert a lock-based critical section into a transaction? What are the benefits and drawbacks of strong atomicity and weak atomicity? Is a single transactional semantics appropriate for all applications and implementations? If not, how many different semantics are necessary? We hope that this work spurs researchers to investigate these questions.

Although this work presents additional challenges for designers of transactional systems, we continue to believe that transactional systems are a promising approach for addressing many difficulties of programming tightly-coupled multiprocessor and multithreaded systems. We hope that this and other critical treatments [18] of practical aspects of transactions will contribute to their successful implementation, evaluation, and use, and not their abandonment.

Acknowledgments. The authors thank Mark Hill, Christos Kozyrakis, Ravi Rajwar, Craig Zilles, and the anonymous reviews for their helpful comments on drafts of this paper; in particular, Mark Hill raised the question of whether the direct conversion from locks to transactions maintains partial correctness. This work is funded in part by NSF Award 0311199 and gifts from Intel Corporation. E Lewis is supported by NSF Career Award 0347290.

References

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

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *International Symposium on High-Performance Computer Architecture*, pages 316–327, 2005.

[3] Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. Type inference for atomicity. In *Types in Language Design and Implementation*, pages 47–58, 2005.

[4] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Programming Language Design and Implementation*, pages 338–349, 2003.

[5] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *Types in Language Design and Implementation*, pages 1–12, 2003.

[6] Lance Hammond, Brian D. Carlstrom, Vicky Wong, Ben Hertzberg, Mike Chen, Christos Kozyrakis, and Kunle Olukotun. Programming with transactional coherence and consistency (TCC). In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13, 2004.

[7] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *International Symposium on Computer Architecture*, pages 102–113, 2004.

[8] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.

[9] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Principles of Distributed Computing*, pages 92–101, 2003.

[10] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, pages 289–300, 1993.

[11] Mark D. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer*, 31(8): 28–34, August 1998.

[12] Kevin E. Moore, Mark D. Hill, and David A. Wood. Thread-level transactional memory. Technical Report 1524, Department of Computer Sciences, University of Wisconsin, March 2005.

[13] Ravi Rajwar and James R. Goodman. Speculative lock elision: enabling highly concurrent multithreaded execution. In *International Symposium on Microarchitecture*, pages 294–305, 2001.

[14] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, 2002.

[15] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *International Symposium on Computer Architecture*, 2005.

[16] Nir Shavit and Dan Touitou. Software transactional memory. In *Principles of Distributed Computing*, pages 204–213, 1995.

[17] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional monitors for concurrent objects. In *European Conference on Object-Oriented Programming*, pages 519–542, 2004.

[18] Craig Zilles and David H. Flint. Challenges to providing performance isolation in transactional memories. In *Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.