Transactions as the Foundation of a Memory Consistency Model*

Luke Dalessandro Michael L. Scott
Computer Science
University of Rochester

Michael F. Spear
Computer Science and Engineering
Lehigh University

URCS Technical Report 959
July 2010

Abstract

We argue for transactions as the synchronization primitive of an ordering-based memory consistency model. Rather than define transactions in terms of locks, our model defines locks, conditions, and atomic/volatile variables in terms of transactions. A traditional critical section, in particular, is a region of code, bracketed by transactions, in which certain data have been *privatized*. Our memory model, originally published at OPODIS'08, is based on the database notion of *strict serializability* (SS). In an explicit analogy to the DRF0 of Adve and Hill, we demonstrate that SS provides the appearance of *transactional sequential consistency* (TSC) for programs that are *transactional data-race free* (TDRF).

We argue against relaxation of the total order on transactions, but show that selective relaxation of the relationship between program order and transaction order (*selective strict serializability*—SSS) can allow the implementation of transaction-based locks to be as efficient as conventional locks. We also show that condition synchronization (in the form of the transactional retry primitive) can be accommodated in our model without explicit mention of speculation, *opacity*, or aborted transactions. Finally, we compare SS and SSS to the notion of *strong isolation* (SI), arguing that SI is neither sufficient for TSC nor necessary in programs that are TDRF.

1 Introduction

Transactional Memory (TM) [16,19] attempts to simplify synchronization by raising the level of abstraction. Drawing inspiration from databases, it allows the programmer to specify that a block of code should execute atomically, without specifying how that atomicity should be achieved. (The typical implementation will be based on speculation and rollback.) In comparison to lock-based synchronization, TM avoids the possibility of deadlock, and—at least to a large extent—frees the programmer from an unhappy choice between the simplicity of coarse-grain locking and the higher potential concurrency of fine-grain locking.

Unfortunately, for a mechanism whose principal purpose is to simplify the programming model, TM has proven surprisingly resistant to formal definition. Difficult questions—all of which we address in this paper—include the following. ▷ Does the programmer need to be aware of speculation and rollback? What happens if a transaction attempts to perform an operation that cannot be rolled back? ▷ What happens when the same data are accessed both within and outside transactions? Does the answer depend on races between transactional and nontransactional code? ▷ Can transactions be added to a program already containing locks—that is, can the two be used together? ▷ How does one express condition synchronization, given that activities of other threads are not supposed to be visible to an already-started transaction?

*This work was supported in part by NSF grants CNS-0615139, CCF-0702505, and CSR-0720796; and by financial support from Intel and Microsoft.

Answers to these questions require a *memory model*—a set of rules that govern the values that may be returned by reads in a multithreaded program. It is generally agreed that programmers in traditional shared-memory systems expect *sequential consistency*—the appearance of a global total order on memory accesses, consistent with program order in every thread, and with each read returning the value from the most recent write to the same location [20]. We posit that transactional programmers will expect *transactional sequential consistency* (TSC)—SC with the added restriction that accesses of a given transaction be contiguous in the total execution order. However, just as typical multiprocessors and parallel programming languages provide a memory model weaker than SC (due to its implementation cost), typical transactional systems can be expected to provide a model weaker than TSC. How should this model be defined?

Several possibilities have been suggested, including *strong isolation* (SI) (a.k.a. strong atomicity) [7,35], *single lock atomicity* (SLA) [16, 1st edn., p. 20] [26], and approaches based on ordering-based memory models [12], linearizability [13,32], and operational semantics [1,27]. Of these, SLA has received the most attention. It specifies that transactions behave as if they acquired a single global mutual exclusion lock.

Several factors, however, make SLA problematic. First, it requires an underlying memory model to explain the behavior of the equivalent lock-based program. Second, it leads to arguably inappropriate semantics for programs that have transactional-nontransactional data races, that mix transactions with fine-grain locks, or that contain infinite loops in transactions. Third—and perhaps most compelling—it defines transactions in terms of the mechanism whose complexity we were supposedly attempting to escape.

We have argued [37] that ordering-based memory models such as those of Java [23] and C++ [8] provide a more attractive foundation than locks for TM. Similar arguments have been made by others, including Grossman et al. [12], Moore and Grossman [27], Luchangco [21], Abadi et al. [1], and Harris [14]. Our model is based on the *strict serializability* (SS) of database transactions. We review it in Section 2. We also show, in a manner analogous to the work of Gharachorloo, Adve, et al. [11], but omitted from our previous study, that SS provides the appearance of TSC for programs that are *transactional data-race free* (TDRF).

In Section 3 we show how locks and other traditional synchronization mechanisms can be defined *in terms of* transactions, rather than the other way around. By making atomicity the fundamental unifying concept, SS provides easily understood (and, we believe, intuitively appealing) semantics for programs that use a mix of synchronization techniques. In Section 4 we note that *selective strict serializability* (SSS), also from our previous work, can eliminate the need for "unnecessary" fences in the implementation of lock and volatile operations, allowing those operations to have the same code—and the same cost—as in traditional systems, while still maintaining a global total order on transactions. In section 5 we show how to augment SS or SSS with condition synchronization (specifically, the retry primitive of Harris et al. [17]) without explicit mention of speculation or aborted transactions. Finally, in Section 6, we compare SS to the notion of *strong isolation* (SI), arguing that SI is both insufficient to guarantee TSC for arbitrary programs, and unnecessary in programs that are TDRF. We conclude in Section 7.

2 The Basic Transactional Model

As is customary [11], we define a *program execution* to be a set of *thread histories*, each of which comprises a totally ordered sequence of *reads*, *writes*, and other *operations*—notably *external actions* like input and output. The history of a given thread is determined by the program text, the semantics (not specified here) of the language in which that text is written, the input provided at run time, and the values returned by reads (which may be set by other threads). An execution is said to be *sequentially consistent* (SC) if there exists a total order on reads and writes, across all threads, consistent with program order in each thread, such that each read returns the value written by the most recent preceding write to the same location.

An *implementation* maps source programs to sets of *target executions* consisting of instructions on some real or virtual machine. The implementation is correct only if, for every target execution, there exists an

equivalent program execution—one that performs the same external actions, in the same order. (Neither the program execution nor the target execution necessarily imposes a total order on external actions, but if action a_1 precedes action a_2 in the program execution, it must also precede it in the target execution.)

Given the cost of sequential consistency on target systems, *relaxed* consistency models differentiate between *synchronization* operations and *ordinary* memory accesses (reads and writes). Operations within a thread are totally ordered by *program order* $<_p$. Synchronization operations across threads are partially ordered by *synchronization order* $<_s$, which must be consistent with program order. The irreflexive transitive closure of $<_p$ and $<_s$, known as *happens-before order* ($<_{hb}$), provides a global partial order on operations across threads.

Two ordinary memory accesses, performed by different threads, are said to *conflict* if they access the same location and at least one of them is a write. An execution is said to have a *data race* if it contains a pair of conflicting accesses that are not ordered by $<_{hb}$. A program is said to be *data-race free* (DRF) with respect to $<_s$ if none of its sequentially consistent executions has a data race.

Relaxed consistency models differ in their choice of $<_s$ and in their handling of data races. In all models, a read is permitted to return the value written by the most recent write to the same location along some happens-before path. If the program is data-race free, any topological sort of $<_{hb}$ will constitute a sequentially consistent execution.

For programs with data races, arguably the simplest strategy is to make the behavior of the entire program undefined. Boehm et al. [8] argue that any attempt to define stronger semantics for C++ would impose unacceptable implementation costs. For managed languages, an at-least-superficially attractive approach is to allow a read to return either (1) the value written by the most recent write to the same location along some happens-before path or (2) the value written by a racing write to that location (one not ordered with the read under $<_{hb}$). As it turns out, this strategy is insufficiently strong to preclude circular reasoning. To avoid "out of thin air" reads, and ensure the integrity of the virtual machine, the Java memory model imposes an additional *causality* requirement, under which reads must be incrementally explained by already-justified writes in shorter executions [23].

2.1 Transactional Sequential Consistency

Our transactional memory model builds on the suggestion, first advanced by Grossman et al. [12] and subsequently adopted by others [1, 27, 37], that $<_s$ be defined in terms of transactions. We extend thread histories to include begin_txn and end_txn operations, which we require to appear in properly nested pairs. We consider inner pairs to be subsumed by surrounding pairs, and use the term "transaction" to refer to the contiguous sequence of operations in a thread history beginning with an outermost begin_txn and ending with the matching end_txn.

Given experience with conventional parallel programs, we expect that (1) races in transactional programs will generally constitute bugs, and (2) the authors of transactional programs will want executions of their (data-race-free) programs to appear sequentially consistent, with the added provision that transactions occur atomically. This suggests the following definition:

A program execution is *transactionally sequentially consistent* (TSC) iff there exists a global total order on operations, consistent with program order in each thread, that explains the execution's reads (in the sense that each read returns the value written by the most recent write to the same location), and in which the operations of any given transaction are contiguous. An *implementation* (system) is TSC iff for every realizable target execution there exists an equivalent TSC program execution.

Similar ideas have appeared in several previous studies. TSC is equivalent to the *strong semantics* of Abadi et al. [1], the *StrongBasic* semantics of Moore and Grossman [27], the sequentially consistent model of Shasha and Snir [33] (with "high-level" atomic operations), and the transactional memory with *store*

atomicity described by Maessen and Arvind [22]. TSC is also equivalent to what Larus and Rajwar called *strong isolation* [16, 1st edn., p. 27], but stronger than the usual meaning of that term, which does not require a global order among nontransactional accesses [7, 9]. Adve and Hill [5] muse about transactional serialization (essentially TSC) as a possible foundation for their memory models, but dismiss it as too costly.

2.2 Strict Serializability

For TM, the natural relaxed consistency model defines an order on transactions, which then combines with program order to produce a global order (the transactional analogue of happens-before) that determines the values that may be returned by an execution's reads. The principal questions then are: what is the transaction order, and how does it combine with program order?

In the database world, the standard ordering criterion is *serializability* [29], which requires that the result of executing a set of transactions be equivalent to some execution in which the transactions take place one at a time, and any transactions executed by the same thread take place in program order. *Strict serializability* (SS) imposes the additional requirement that if transaction A completes before B starts (in the underlying implementation), then A must occur before B in the equivalent serial execution. The intent of this definition is that if external (non-database) operations allow one to tell that A precedes B, then A must serialize before B. We adopt strict serializability as the synchronization order for transactional memory, equating non-database operations with nontransactional memory accesses, and insisting that such accesses occur between the transactions of their respective threads, in program order. In our formulation:

Program order, $<_p$, is a union of per-thread total orders, and is specified explicitly as part of the execution. In a legal execution, the operations performed by a given thread are precisely those specified by the sequential semantics of the language in which the source program is written, given the values returned by the execution's input operations and reads. Because (outermost) transactions are contiguous in program order, $<_p$ also orders transactions of a given thread with respect to one another and to the thread's nontransactional operations.

Transaction order, $<_t$, is a total order on transactions, across all threads. It is consistent with $<_p$, but is not explicitly specified. For convenience, if $a \in A$, $b \in B$, and $A <_t B$, we will sometimes say $a <_t b$.

Strict serial order, $<_{ss}$, is a partial order on memory accesses induced by $<_p$ and $<_t$. Specifically, it is a superset of $<_t$ that also orders nontransactional accesses with respect to preceding and following transactions of the same thread. Formally, for all accesses a and c in a program execution, we say $a <_{ss} c$ iff at least one of the following holds: (1) $a <_t c$; (2) $\exists$ a transaction A such that $(a \in A \ \wedge \ A <_p c)$; (3) $\exists$ a transaction C such that $(a <_p C \ \wedge \ c \in C)$; (4) $\exists$ an access b such that $a <_{ss} b <_{ss} c$.

We say a memory access b *intervenes* between a and c iff $a <_p b \vee a <_{ss} b$ and $b <_p c \vee b <_{ss} c$. Read r is then permitted to return the value written by write w if r and w access the same location l, $w <_p r \vee w <_{ss} r$, and there is no intervening write of l between w and r. Depending on the choice of programming language, r may also be permitted to return the value written by w if r and w are incomparable under both $<_p$ and $<_{ss}$. Specifically, in a Java-like language, a read should be permitted to see an incomparable but causally justifiable write.

An execution with program order $<_p$ is said to be *strictly serializable* (SS) if there exists a transaction order $<_t$ that together with $<_p$ induces a strict serial order $<_{ss}$ that (together with $<_p$) permits all the values returned by reads in the execution. A TM implementation is said to be SS iff for every realizable target execution there exists an equivalent SS program execution.[1]

In a departure from nontransactional models, we do not include all of program order in the global $<_{ss}$ order. By adopting a more minimal connection between program order and transaction order, we gain the opportunity (in Section 4) to relax this connection as an alternative to relaxing the transaction order itself.

[1] Our TRANSACT'09 paper used the term *transactionally happens-before consistent* in place of SS.

2.3 Transactional Data-Race Freedom

As in traditional models, two ordinary memory accesses are said to *conflict* if they are performed by different threads, they access the same location, and at least one of them is a write. A legal execution is said to have a *data race* if it contains, for every possible $<_t$, a pair of conflicting accesses that are not ordered by the resulting $<_{ss}$. A program is said to be *transactional data-race free* (TDRF) if none of its TSC executions has a data race.

It is easy to show that any execution of a TDRF program on an SS implementation will be TSC. By definition, any such execution E will be SS. **Case 1:** Suppose that all E's reads see writes that are ordered with the read under $<_p$ or $<_{ss}$. Then we can construct an equivalent TSC execution by choosing operations from each thread history up to the beginning of the first transaction, and then repeatedly choosing, in $<_t$ order, the next transaction and subsequent nontransactional operations of the same thread. This construction preserves $<_p$ and $<_{ss}$, and thus the values seen by all reads. **Case 2:** Suppose on the other hand that at least one read in E sees an incomparable write. We obtain a contradiction as follows. Let w be an earliest write that is seen by an incomparable read, and let r be an earliest read that sees w. Employ the construction from Case 1, up to the point at which w or r appears. Complete the transaction, if any, in which w or r appears, but proceed no farther in its thread. Then continue the construction up to the point at which the other of w and r appears (again completing the current transaction, if any). This construction cannot require operations to be chosen from the first thread, or w and r would be ordered. The resulting execution prefix is clearly TSC. Moreover it can be extended to include conflicting accesses w and r while remaining TSC, meaning that the program could not have been TDRF.

3 Modeling Locks and Other Traditional Synchronization

As noted in Section 1, one often sees attempts to define transactions in terms of locks. Given a memory consistency model based on transactions, however, we can easily define locks in terms of transactions. This avoids any objections to defining transactions in terms of the thing they're intended to supplant. It's also arguably simpler, since we need a memory model anyway to define the semantics of locks.

Our approach stems from two observations. First, any practical implementation of locks requires some underlying atomic primitive(s) (e.g., test-and-set or compare-and-swap). We can use transactions to model these, and then define locks in terms of a reference implementation. Second, a stream of recent TM papers has addressed the issue of *publication* [26] and *privatization* [16, 24, 42], in which a program uses transactions to transition data back and forth between logically shared and private states, and then uses nontransactional accesses for data that are private. We observe that privatization amounts to locking.

3.1 Reference Implementations

Figure 1 shows reference implementations for locks and condition variables. Similar implementations can easily be written for volatile (atomic) variables, monitors, semaphores, conditional critical regions, etc. Note that this is *not* necessarily how synchronization mechanisms would be implemented by a high-quality language system. Presumably the compiler would recognize calls to acquire, release, etc., and generate semantically equivalent but faster target code.

By defining traditional synchronization in terms of transactions, we obtain easy answers to all the obvious questions about how the two interact. Suppose, for example, that a transaction attempts to acquire a lock (perhaps new transactional code calls a library containing locks). If there is an execution prefix in which the lock is free at the start of the transaction, then acquire will perform a single read and write, and (barring other difficulties) the transaction can occur. If there is no execution prefix in which the lock is free at the

```
class lock {
    Boolean held := false
    void acquire( ) {
        while true {
            atomic {
                if not held {
                    held := true
                    return
                }
            }
        }
    }
    void release( ) {
        atomic {
            held := false
        }
    }
```

```
class condition {
    class token {
        Boolean ready := false
    }
    queue<token> waiting := [ ]
    void wait( ) {
        t := new token
        waiting.enqueue(t)
        while true {
            release( )
            acquire( )
            if t.ready {
                return
            }
        }
    }
```

```
    void signal( ) {
        t := waiting.dequeue( )
        if t != null {
            t.ready := true
        }
    }
    void signal_all( ) {
        while true {
            t := waiting.dequeue( )
            if t == null {
                return
            }
            t.ready := true
        }
    }
}
```

Figure 1: Reference implementations for locks and condition variables. Lock L is assumed to be held when calling condition methods.

start of the transaction, then (since transactions appear in executions in their entirety, or not at all) there is no complete (terminating) execution for the program. If there are some execution prefixes in which the lock is available and others in which it is not, then the one(s) in which it is available can be extended (barring other difficulties) to create a complete execution. (Note that executions enforce safety, not liveness—more on this in Section 5.) The reverse case—where a lock-based critical section contains a transaction—is even easier: since acquire and release are themselves separate transactions, no actual nesting occurs.

Note that in the absence of nesting, only acquire and release—not the bodies of critical sections themselves—are executed as transactions; critical sections protected by different locks can therefore run concurrently. Interaction between threads can occur within lock-based critical sections but not within transactions. (Concurrency within transactions [6] is not considered here.)

3.2 Advantages with Respect to Lock-Based Semantics

Several researchers, including Harris and Fraser [15] and Menon et al. [25, 26], have suggested that lock operations (and similarly volatile variable accesses) be treated as tiny transactions. Their intent, however, was not to merge all synchronization mechanisms into a single formal framework, but simply to induce an ordering between legacy mechanisms and any larger transactions that access the same locks or volatiles. Harris and Fraser suggest that it should be possible (as future work) to develop a unified formal model reminiscent of the Java memory model. The recent draft TM proposal for C++ includes transactions in the language's synchronizes-with and happens-before orders, but as an otherwise separate mechanism; nesting of lock-based critical sections within transactions is explicitly prohibited [4].

Menon et al., by contrast, define transactions explicitly in terms of locks. Unfortunately, as noted in a later paper from the same research group (Shpeisman et al. [34]), this definition requires transactions to mimic certain unintuitive (and definitely non-atomic) behaviors of lock-based critical sections in programs with data races. One example appears in Figure 2; others can be found in Luchangco's argument against lock-based semantics for TM [21]. By making transactions fundamental, we avoid any pressure to mimic the problems of locks. In Figure 2, for example, we can be sure there is no terminating execution. If, however, we were to replace Thread 1's transaction with a lock-based critical section (L.acquire( ); v = 1; while (w != 1) { }; L.release( );), the program could terminate successfully.

```
Initially v == w == 0
```

Thread 1	Thread 2
`1: atomic {`	
`2:  v = 1`	
	`3: while (v != 1) {}`
	`4: w = 1`
`5:  while (w != 1) {}`	
`6: }`	

Figure 2: Reproduced from Figure 2 of Shpeisman et al. [34]. If transactions have the semantics of lock-based critical sections, then this program, though racy, should terminate successfully.

3.3 Practical Concerns

Volos et al. [41] describe several "pathologies" in the potential interaction of transactions and locks. Their discussion is primarily concerned with implementation-level issues on a system with hardware TM, but some of these issues apply to STM systems as well. If traditional synchronization mechanisms are implemented literally *as transactions*, then our semantics will directly obtain, and locks will interact in a clear and well-defined manner with other transactions. If locks are implemented in some special, optimized fashion, then the implementation will need to ensure that all possible usage cases obey the memory model. Volos et al. describe an implementation that can be adapted for use with STM systems based on ownership records. In our NOrec system [10], minor modifications to the acquire operation would allow conventional locks to interact correctly with unmodified transactions.

4 Improving Performance with Selective Strictness

A program that accesses shared data only within transactions is clearly data-race free, and will experience TSC on any TM system that guarantees that reads see values consistent with some $<_t$. A program P that sometimes accesses shared data outside transactions, but that is nonetheless TDRF, will experience TSC on any TM system S that similarly enforces some $<_{ss}$. Transactions in P that begin and end, respectively, a region of data-race-free nontransactional use are referred to as *privatization* and *publication* operations, and S is said to be *privatization* and *publication safe* with respect to SS.

Unfortunately, many existing TM implementations are not publication and privatization safe, and modifying them to be so imposes nontrivial costs [24]. In their paper on lock-based semantics for TM, Menon et al. note that these costs are particularly egregious under single lock atomicity (SLA), which forces every transaction to be ordered with respect to every other [26]. Their weaker models (DLA, ALA, ELA) aim to reduce the cost of ordering (and in particular publication safety) by neglecting to enforce it in questionable cases (e.g., for empty transactions, transactions with disjoint access sets, or transactions that share only an anti-dependence).

We can define each of these weaker models in our ordering-based framework, but the set of executions for a program becomes much more difficult to define, and program behavior becomes much more difficult to reason about. As noted by Harris [16, Chap. 3] and by Shpeisman et al. [34], orderings become dependent on the precise set of variables accessed by a transaction—a set that may depend not only on program input and control flow, but also on optimizations (e.g., dead code elimination) performed by the compiler.

Rather than abandon the global total order on transactions, we have proposed [37] an optional relaxation of the ordering between nontransactional accesses and transactions. Specifically, we allow a transaction to be labeled as *acquiring* (privatizing), *releasing* (publishing), both, or neither.

Selective strict serial order, $<_{sss}$, is a partial order on memory accesses. Like strict serial order, it is consistent with transaction order. Unlike strict serial order, it orders nontransactional accesses only with respect to preceding acquiring transactions and subsequent releasing transactions of the same thread (and, transitively, transactions with which those are ordered). Formally, for all accesses a, c, we say $a <_{sss} c$ iff at

least one of the following holds: (1) $a <_t c$; (2) $\exists$ an acquiring transaction A such that $(a \in A \ \wedge \ A <_p c)$; (3) $\exists$ a releasing transaction C such that $(a <_p C \ \wedge \ c \in C)$; (4) $\exists$ an access b such that $a <_{sss} b <_{sss} c$.

Note that for any given program $<_{sss}$ will be a subset of $<_{ss}$—typically a proper one—and so a program that is TDRF with respect to SS will not necessarily be TDRF with respect to SSS. This is analogous to the situation in traditional ordering-based memory models, where, for example, a program may be DRF1 but not DRF0 [11].

A transactional programming language will probably want to specify that transactions are both acquiring and releasing by default. A programmer who knows that a transaction does not publish or privatize data can then add an annotation that permits the implementation to avoid the cost of publication and privatization safety. Among other things, on hardware with a relaxed memory consistency model, identifying a transaction as (only) privatizing will allow the implementation to avoid an expensive *write-read fence*. The designers of the C++ memory model went to considerable lengths—in particular, changing the meaning of trylock operations—to avoid the need for such fences before acquiring a lock [8]. Given SSS consistency in Figure 1, we would define the transaction in lock.acquire to be (only) acquiring, and the transaction in lock.release to be (only) releasing. Similarly, a get (read) operation on a volatile variable would be acquiring, and a put (write) operation would be releasing.

5 Condition Synchronization and Forward Progress

For programs that require not only atomicity, but also condition synchronization, traditional condition variables will not suffice: since transactions are atomic, they cannot be expected to see a condition change due to action in another thread. One could release atomicity, effectively splitting a transaction in half (as in the *punctuated* transactions of Smaragdakis et al. [36]), but this would break composability, and require the programmer to restore any global invariants before waiting on a condition. One could also limit conditions to the beginning of the transaction [15], but this does not compose.

Among various other alternatives, the most popular appears to be the retry primitive of Harris et al. [17]. The construct "if (!desired_condition) retry" instructs a speculation-based implementation of TM to roll the current thread back to the beginning of its current transaction, and then deschedule it until something in the transaction's read set has been written by another thread. While the name "retry" clearly has speculative connotations, it can also be interpreted (as Harris et al. do in their operational semantics) as controlling the conditions under which the surrounding transaction is able to perform its one and only execution. We therefore define retry, for our ordering-based semantics, to be equivalent to while (true) { }.

At first glance, this definition might seem to allow undesirable executions. If T_1 says atomic {f := 1} and T_2 says atomic {if (f != 1) retry}, we would not want to admit an execution in which T_2 "goes first" and waits forever. But there is no such execution! Since transactions appear in executions in their entirety or not at all, T_2's transaction can appear only if T_1's transaction has already appeared. The programmer may think of retry in terms of prescience (execute this only when it can run to completion) or in terms of, well, re-trying; the semantics just determine whether a viable execution exists. It is possible, of course, that for some programs there will exist execution prefixes[2] such that some thread(s) are unable to make progress in any possible extension; these are precisely the programs that are subject to deadlock (and deadlock is undecidable).

Because our model is built on atomicity, rather than speculation, it does not need to address aborted transactions. An implementation based on speculation is simply required to ensure that such transactions have no visible effects. In particular, there is no need for the *opacity* of Guerraoui and Kapałka [13]; it is acceptable for the implementation of a transaction to see an inconsistent view of memory, so long as the

[2] We assume that even in such a prefix, transactions appear *in toto* or not at all.

compiler and run-time system "sandbox" its behavior (as, for example, in Bartok STM [18], McRT [3], JudoSTM [28], Intel's SGLA [26], or DASTM [30]).

5.1 Progress

Clearly an implementation must realize only target executions equivalent to some program execution. Equally clearly, it need not realize target executions equivalent to *every* program execution. Which do we want to require it to realize?

It seems reasonable to insist, for starters, that threads do not stop dead for no reason. Consider some realizable target execution prefix M and an equivalent program execution prefix E. If, for thread T, the next operation in program order following T's subhistory in E is nontransactional, we might insist that the implementation be able to extend M to M^+ in such a way that T makes progress—that is, that M^+ be equivalent to some extension E^+ of E in which T's subhistory is longer.

For transactions, which might contain retry or other loops, appropriate goals are less clear. Without getting into issues of fairness, we cannot insist that a thread T make progress in a given implementation just because there exists a program execution in which it makes progress. Suppose, for example, that flag f is initially 0, and that both T_1 and T_2 have reached a transaction reading if (f $<$ 0) retry; f := 1. Absent other code accessing f, one thread will block indefinitely, and we may not wish to dictate which this should be.

Intuitively, we should like to preclude implementation-induced deadlock. As a possible strategy, consider a realizable target execution prefix M with corresponding program execution prefix E, in which each thread in some nonempty set $\{T_i\}$ has reached a transaction in its program order, but has not yet executed that transaction. If for every extension E^+ of E there exists an extension E^{++} of E^+ in which at least one of the T_i makes progress, then the implementation is not permitted to leave all of the T_i blocked indefinitely. That is, there must exist a realizable extension M^+ of M equivalent to some extension E' of E in which the subhistory of one of the T_i is longer.

5.2 Inevitability

If transactions are to run concurrently, even when their mutual independence cannot be statically proven, implementations must in general be based on speculation. This then raises the problem of irreversible operations such as interactive I/O. One option is simply to outlaw these within transactional contexts. This is not an unreasonable approach: locks and privatization can be used to make such operations safe.

If irreversible operations are permitted in transactions, we need a mechanism to designate transactions as *inevitable* (*irrevocable*) [40,43]. This can be a static declaration on the transaction as a whole, or perhaps an executable statement. Either way, irreversibility is simply a hint to the implementation; it has no impact on the memory model, since transactions are already atomic.

In our semantics, an inevitable transaction's execution history is indistinguishable from an execution history in which a thread (1) executes a privatizing transaction that privatizes the whole heap, (2) does all the work nontransactionally, and then (3) executes a publishing transaction. This description formalizes the oft-mentioned lack of composability between retry and inevitability (see Appendix A).

5.3 orElse and abort

In the paper introducing retry [17], Harris et al. also proposed an orElse construct that can be used to pursue an alternative code path when a transaction encounters a retry. In effect, orElse allows a computation to notice—and explicitly respond to—the failure of a speculative computation.

Both basic transactions and the retry primitive can be described in terms of atomicity: "this code executes all at once, at a time when it can do so correctly." The orElse primitive, by contrast, "leaks"

information—a failure indication—out of a transaction that "doesn't really happen," allowing the program to do something else instead. We have considered including failed transactions explicitly in program executions or, alternatively, imposing liveness-style constraints across sets of executions ("execution E is invalid because transaction T appears in some other, related execution"), but both of these alternatives strike us as distinctly unappealing. In the balance, our preference is to leave orElse out of TM. Its effect can always be achieved (albeit without composability or automatic roll-back) by constructs analogous to those of Section 3.

In a similar vein, consider the oft-proposed abort primitive, which abandons the current transaction (with no effect) and moves on to the next operation in program order. Shpeisman et al. observe that this primitive can lead to logical inconsistencies if its transaction does not contribute to the definition of data-race freedom [34]. In effect, abort, like orElse, can be used to leak information from an aborted transaction. Shpeisman et al. conclude that aborted transactions must appear explicitly in program executions. We argue instead that aborts be omitted from the definition of TM. Put another way, orElse and abort are speculation primitives, *not* atomicity primitives. If they are to be included in the language, it should be by means of orthogonal syntax and semantics.

6 Strong Isolation

Blundell et al. [7] observed that hardware transactional memory (HTM) designs typically exhibit one of two possible behaviors when confronted with a race between transactional and non-transactional accesses. With *strong isolation* (SI) (a.k.a. *strong atomicity*), transactions are isolated both from other transactions and from concurrent nontransactional accesses; with *weak isolation* (WI), transactions are isolated only from other transactions.

Various papers have opined that STMs instrumented for SI result in more intuitive semantics than WI alternatives [31, 35], but this argument has generally been made at the level of TM implementations, not user-level programming models. From the programmer's perspective, we believe that TSC is the reference point for intuitive semantics—and SS and SSS systems provide TSC behavior for programs that are correspondingly TDRF. At the same time, for a language that assigns meaning to racy programs, SS and SSS permit many of the results cited by proponents of SI as unintuitive. This raises the possibility that SI may improve the semantics of TM for racy programs.

It is straightforward to extend any ordering-based transactional memory model to one that provides SI. We do this for SS in Appendix B, and explore the resulting model (SI-SS) for example racy programs. We note that SI-SS is not equivalent to TSC; that is, there are racy programs that still yield non-TSC executions. One could imagine a memory model in which the racy programs that *do* yield TSC executions with SI are considered to be properly synchronized. Such a model would authorize programmers to write more than just TDRF code, but it would be a significantly more complicated model to reason about: one would need to understand which races are bugs and which aren't.

An additional complication of any programmer-centric model based on strong isolation is the need to explain exactly what is meant by a nontransactional access. Consider Figure 3. Here x is an unsigned long long and is being assigned to nontransactionally. Is this a race under a memory model based on SI? The problem is that Thread 2's assignment to x may not be a single instruction. It is possible (and Java in fact permits) that two 32 bit stores will be used to move the 64 bit value. Furthermore, if the compiler is aware of this fact, it may arrange to execute the stores in arbitrary order. The memory model now must specify the granularity of protection for nontransactional accesses.

While SS and SSS do not require a strongly isolated TM implementation, they do not exclude one either. It may seem odd to consider using a stronger implementation than is strictly necessary, particularly given its cost, but there are reasons why this may make sense. First, SS with happens-before consistency for programs with data races is not trivially compatible with undo-log-based TM implementations [34].

Thread 1	Thread 2
1: atomic {	
2: r = x	x = 3*ull*
3: }	

Figure 3: Is this a correct program under an SI-based memory model?

These implementations require SI instrumentation to avoid out-of-thin-air reads due to aborted transactions. Indeed, two of the major undo-log STMs, McRT [3] and Bartok [18], are strongly isolated for just this reason. Second, as observed by Grossman et al. [12], strong isolation enables *sequential reasoning*. Given a strongly isolated TM implementation, all traditional single-thread optimizations are valid in a transactional context, even for a language with safety guarantees like Java. Third, SI hardware can make it trivial to implement volatile/atomic variables.

With these observations in mind, we would not discourage development of strongly isolated HTM. For STM, we note that a redo-log based TM implementation with a hash-table write set provides most of the properties Grossman et al. attribute to SI—specifically, constraints (1) and (3) from Appendix B, with the option of (2) as well if the read set is also hashed. Such an implementation permits many of the same compiler optimizations that SI does, and, as shown by Spear et al. [38], can provide performance competitive with undo logs. Ultimately, we conclude that SI is insufficient to guarantee TSC for racy programs, and unnecessary to guarantee it for TDRF programs. It may be useful at the implementation level for certain STMs, and certainly attractive if provided by the hardware "for free," but it is probably not worth adding to an STM system if it adds significant cost.

7 Conclusions

While it is commonplace to speak of transactions as a near-replacement for locks, and to assume that they should have SLA semantics, we believe this perspective both muddies the meaning of locks and seriously undersells transactions. Atomicity is a fundamental concept, and it is *not* achieved by locks, as evidenced by examples like the one in Figure 2. By making atomic blocks the synchronization primitive of an ordering-based memory consistency model, we obtain clear semantics not only for transactions, but for locks and other traditional synchronization mechanisms as well.

In future work, we hope to develop a formal treatment of speculation that is orthogonal to—but compatible with—our semantics for TM. We also hope to unify this treatment with our prior work on implementation-level sequential semantics for TM [32].

It is presumably too late to adopt a transaction-based memory model for Java or C++, given that these languages already have detailed models in which other operations (monitor entry/exit, lock acquire/release, volatile/atomic read/write) serve as synchronization primitives. For other languages, however, we strongly suggest that transactions be seen as fundamental.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *SIGPLAN Symp. on Principles of Programming Languages*, Jan. 2008.

[2] M. Abadi, T. Harris, and M. Mehrara. Transactional Memory with Strong Atomicity Using Off-the-Shelf Memory Protection Hardware. In *SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Feb. 2009.

[3] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *SIGPLAN Conf. on Programming Language Design and Implementation*, June 2006.

[4] A.-R. Adl-Tabatabai and T. Shpeisman, editors. *Draft Specification of Transaction Language Constructs for C++*. Transactional Memory Specification Drafting Group, Intel, IBM, and Sun, 1.0 edition, Aug. 2009.

[5] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *Intl. Symp. on Computer Architecture*, May 1990.

[6] K. Agrawal, J. Fineman, and J. Sukha. Nested Parallelism in Transactional Memory. In *SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Feb. 2008.

[7] C. Blundell, E. C. Lewis, and M. M. K. Martin. Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2), Nov. 2006.

[8] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *SIGPLAN Conf. on Programming Language Design and Implementation*, June 2008.

[9] L. Dalessandro and M. L. Scott. Strong Isolation is a Weak Idea. In *4th SIGPLAN Workshop on Transactional Computing*, Feb. 2009.

[10] L. Dalessandro, M. F. Spear, and M. L. Scott. NOrec: Streamlining STM by Abolishing Ownership Records. In *SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Jan. 2010.

[11] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15:399–407, 1992.

[12] D. Grossman, J. Manson, and W. Pugh. What Do High-Level Memory Models Mean for Transactions? In *SIGPLAN Workshop on Memory Systems Performance and Correctness*, Oct. 2006.

[13] R. Guerraoui and M. Kapałka. On the Correctness of Transactional Memory. In *SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Feb. 2008.

[14] T. Harris. Language Constructs for Transactional Memory (invited keynote address). In *SIGPLAN Symp. on Principles of Programming Languages*, Jan. 2009.

[15] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[16] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool, second edition, 2010. (first edition, by Larus and Rajwar only, 2007).

[17] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable Memory Transactions. In *SIGPLAN Symp. on Principles and Practice of Parallel Programming*, June 2005.

[18] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *SIGPLAN Conf. on Programming Language Design and Implementation*, June 2006.

[19] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Intl. Symp. on Computer Architecture*, May 1993.

[20] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):241–248, Sept. 1979.

[21] V. Luchangco. Against Lock-Based Semantics for Transactional Memory (brief announcement). In *ACM Symp. on Parallelism in Algorithms and Architectures*, June 2008.

[22] J.-W. Maessen and Arvind. Store Atomicity for Transactional Memory. *Electronic Notes in Theoretical Computer Science*, 174(9):117–137, June 2007.

[23] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *SIGPLAN Symp. on Principles of Programming Languages*, Jan. 2005.

[24] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *Intl. Conf. on Parallel Processing*, Sept. 2008.

[25] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. Hudson, B. Saha, and A. Welc. Single Global Lock Semantics in a Weakly Atomic STM. In *3rd SIGPLAN Workshop on Transactional Computing*, Feb. 2008.

[26] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *ACM Symp. on Parallelism in Algorithms and Architectures*, June 2008.

[27] K. F. Moore and D. Grossman. High-Level Small-Step Operational Semantics for Transactions. In *SIGPLAN Symp. on Principles of Programming Languages*, Jan. 2008.

[28] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, Sept. 2007.

[29] C. H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, Oct. 1979.

[30] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing Conflicting Transactions in an STM. In *SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Feb. 2009.

[31] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic Optimization for Efficient Strong Atomicity. In *Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2008.

[32] M. L. Scott. Sequential Specification of Transactional Memory Semantics. In *1st SIGPLAN Workshop on Transactional Computing*, June 2006.

[33] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Trans. on Programming Languages and Systems*, 10(2):282–312, Apr. 1988.

[34] T. Shpeisman, A.-R. Adl-Tabatabai, R. Geva, Y. Ni, and A. Welc. Towards Transactional Memory Semantics for C++. In *ACM Symp. on Parallelism in Algorithms and Architectures*, Aug. 2009.

[35] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *SIGPLAN Conf. on Programming Language Design and Implementation*, June 2007.

[36] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with Isolation and Cooperation. In *Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2007.

[37] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. In *Intl. Conf. on Principles of Distributed Systems*, Dec. 2008.

[38] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. A Comprehensive Contention Management Strategy for Software Transactional Memory. In *SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Feb. 2009.

[39] M. F. Spear, M. M. Michael, and M. L. Scott. Inevitability Mechanisms for Software Transactional Memory. In *3rd SIGPLAN Workshop on Transactional Computing*, Feb. 2008.

[40] M. F. Spear, M. Silverman, L. Dalessandro, M. M. Michael, and M. L. Scott. Implementing and Exploiting Inevitability in Software Transactional Memory. In *2008 Intl. Conf. on Parallel Processing*, Sept. 2008.

[41] H. Volos, N. Goyal, and M. Swift. Pathological Interaction of Locks with Transactional Memory. In *3rd SIGPLAN Workshop on Transactional Computing*, Feb. 2008.

[42] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Intl. Symp. on Code Generation and Optimization*, Mar. 2007.

[43] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable Transactions and Their Applications. In *ACM Symp. on Parallelism in Algorithms and Architectures*, June 2008.

A Inevitability as Privatization

In our semantics, if an inevitable transaction privatizes the shared heap and then enters an infinite loop, then no further progress can be made by any transaction. This provides a defined behavior for the clearly buggy program. Furthermore, if inevitability is an executable statement, then this description necessitates that inevitable transactions execute serially (that is, while no other transactions are executing, in contrast to proposals by Welc et al [43] and Spear et al. [39, 40]). Onerous though such a constraint may seem, it is necessary in both SS and throughout all but the weakest (ELA) of Menon et al.'s lock-based semantics. Figure 4 presents an example execution that can otherwise violate publication safety. While this particular example could be addressed by only blocking concurrent *writing* transactions, one can create more subtle examples, as suggested by Menon et al. [26, Fig. 12]) in which Thread 2 is a read-only transaction, and "publishes via anti-dependence." To handle such cases, concurrent *reader* transactions must also block.

```
Initially val == 42, flag == false, and r == 0
```

Thread 1	Thread 2
`atomic {`	
`  x = val`	
	`val = 0`
	`atomic { flag = true }`
`  become_inevitable`	
`  if (flag) r = x`	
`}`	

Figure 4: At the point where Thread 1 requests to become inevitable, `flag` has not been read, and there is (as yet) no conflict with the transaction in Thread 2. If Thread 2's transaction is permitted to commit, however, and Thread 1's transaction subsequently becomes inevitable, we are forced into a violation of publication safety as `r` will be set to 42.

B Strong Isolation As a Transactional Memory Model Mix-in

Grossman et al. [12, Sec. 2.2] enumerate three "questions about atomicity" that illuminate the difference between strong and weak isolation. We cast these as three constraints, provided by strong isolation, on the reads-see-writes relationship. We formalize them here for SS; extension to SSS is straightforward.

(1) If b is a transactional read, and there exists at least one write, w, within the same transaction, to the same location, such that $w <_p b$, then b must return the value of the most recent such write under $<_p$.

(2) If (1) does not hold, then if b is transactional, and if there exists at least one read, a, within the same transaction, to the same location, such that $a <_p b$, then b must see the same write seen by a.

(3) If neither (1) nor (2) holds, then b may see the most recent write to the same location along some $<_p$ or $<_{ss}$ path, or an unordered write to the same location (subject to causality) provided that it is not an *intermediate transactional write*. A transactional write a is said to be intermediate if there exists a write, b, within the same transaction, to the same location, such that $a <_p b$.

As defined by Blundell et al. [7], an SI implementation provides two properties, *containment* (a transaction is invisible until it commits) and *non-interference* (nontransactional writes do not appear to occur mid-transaction). Non-interference is provided by the combination of constraints (1) and (2) above; containment is provided by constraint (3). This mix-in definition can be integrated into any ordering-based transactional memory model by amending constraint (3) to account for the model's underlying reads-see-writes relationship.

We can now examine the implications of the resulting strongly isolated SS model (SI-SS) on a number of racy program examples.

Initially x == y == 0

Thread 1	Thread 2
1: x = 1	y = 1
2: r1 = y	r2 = x

Figure 5: Can r1 == r2 == 0? There is no TSC execution with this outcome, however both SS and SI-SS permit the result as they rely on the same reads-see-writes mapping for programs without transactions.

Initially x == 0

Thread 1	Thread 2
1: atomic {	
2: x = 1	
	3: r1 = x
4: x = 2	
5: }	

Figure 6: An example of containment. Can r1 == 1? SS allows the transactional read of x in thread 2 to see the value of any racing writes—permitting the result. SI-SS explicitly forbids seeing transactional intermediate writes in nontransactional reads.

Consider the well-known example of Figure 5. Reasoning about this code in an SC setting disallows the outcome of r1 == r2 == 0. TSC requires sequential consistency, so it too prohibits this outcome. SI-SS permits the ordering loop.

Now consider the code fragment in Figure 6. The two transactional writes of x in Thread 1 race with the nontransactional read of x in Thread 2. Under SS, the read in Thread 2 may return any of 0 (the initial value of x), 1, or 2. It is easy to see how a weakly isolated buffered-update STM system might realize the result r1 == 1. The implementation might keep its redo log as a time-ordered list of writes. Once committed, it could write this log back to memory in order, meaning that there exists a point in time where the write from line 2 has occurred and not yet been overwritten by the write from line 3. If Thread 2's read occurs during that period, it will see the partially committed results.

Strong isolation does not permit the result r1 == 1: if the read sees the value of any transactional write, then it must see the results of all of the transactional writes. In this case, it must return 2. This result appears to require that a strongly isolated STM implementation be prepared to block on the nontransactional read of x until the transactional writer has cleaned up completely. An analogous situation occurs when a nontransactional write races with a transactional read. The write must appear to occur either totally before or after the transaction.

Preservation of non-interference and containment reduces the set of unexpected results a racy program might generate, but it does not remove all unexpected results. Consider again the example of Figure 5. If we enclose the code of Thread 1 in a transaction, SI-SS still fails to prohibit the unexpected result of r1 == r2 == 0. SI-SS says only that neither of the instructions in Thread 2 may appear to occur during the execution of Thread 1's transaction. The compiler, however, may chose to reorder the independent instructions in Thread 2 based on its underlying memory model. Likewise, the hardware may reorder writes on many machines. While TSC disallows the unexpected result, SI-SS clearly does not.

It is important to note that these results are not restricted to our SI-SS model. In particular, at the implementation level, where SI has traditionally been considered, the systems of Shpeisman et al. [35] and Schneider et al. [31] are realized as extensions to Java, and presumably inherit its relaxed memory model [23]. Similarly Abadi et al. [2] operate in the context of C# with its relaxed model. All of these relaxed models permit the unintuitive result of r1 == r2 == 0.