TRANSACT '09

Strong Isolation is a Weak Idea *

Luke Dalessandro Michael L. Scott
Department of Computer Science, University of Rochester
{luked, scott}@cs.rochester.edu

Abstract

In the transactional memory (TM) community, much debate has revolved around the choice of *strong* vs. *weak isolation* (SI vs. WI) between transactions and conflicting non-transactional accesses. In this position paper we argue that what programmers really want is the natural transactional extension of sequential consistency (SC), and that even SI is insufficient to achieve this.

It is widely agreed among architects and language designers that SC imposes unacceptable hardware costs and compiler restrictions. Programmer-centric, relaxed memory models were developed as a compromise, guaranteeing SC to programs that "follow the rules" while admitting many of the compiler optimizations that result in fast single-threaded execution. We argue for an analogous *transactional data-race-free* (TDRF) programming model. We observe that WI is strong enough to implement this model, and further that weakly isolated TM systems based on redo logging can provide the safety guarantees (no "out-of-thin-air reads") required by languages like Java. Seen in this light, strong isolation (SI) serves only to require more constrained behavior in racy (buggy) programs. We submit that the benefit is not worth the cost, at least for software TM.

1. Introduction

Databases systems guarantee that the execution of a sequence of transactions is serializable; that is, that any actual execution must produce results equivalent to an execution in which the transactions execute indivisibly in some sequential order [22]. This powerful idea provides the programmer an intuitive programming interface, while allowing the DBMS sufficient freedom to optimize the actual concurrency of the execution by overlapping transactions wherever it is possible without violating serializability.

Inspired by these transactional databases, Herlihy and Moss [15] observed that transactional memory (TM) would be a powerful tool to enable the development and composable use of concurrent data structures.

The apparent simplicity of the transactional interface to memory hides an underlying complication. Every access to a transactional database is required to be part of a transaction. In TM however it is sometimes natural and desirable to access the same data both inside and outside transactions. Extra-transactional access occurs in the well-documented *publication* [20] and *privatization* [17, 30, 32] programming idioms, where programmers use transactions to transition data back and forth between logically shared and private states, and access private data nontransactionally.

In a program with publication and privatization an obvious question arises: what happens if there is a race between conflicting transactional and nontransactional accesses—a case in which nontransactional access in thread A is not separated from transactional access in thread B by any transaction in thread A? Blundell et al. [8] observed that hardware transactional memory (HTM) designs seem to exhibit one of two possible behaviors when confronted with a such a transactional race. Some hardware systems provide *strong isolation*[1] (SI), where transactions are isolated from both other transactions and concurrent nontransactional accesses, while others provide *weak isolation* (WI), where transactions are isolated only from other transactions. In a WI system, nontransactional reads in a racy program may see the state of an incomplete (or even doomed) transaction; likewise, nontransactional writes may appear to occur in the middle of a transaction.

Larus and Rajwar, in an apparent attempt to provide an intuitive definition of Blundell's SI, present a third alternative [17, p. 27]: "Strong isolation automatically converts all operations outside an atomic block into individual transactional operations, thus replicating the database model in which all accesses to shared state execute in transactions." This definition is actually much stronger than Blundell appears to have intended: making nontransactional accesses miniature transactions requires that they serialize with respect to one another, resulting in a system in which non-

* This work was supported in part by NSF grants CNS-0411127, CNS-0615139, CCF-0702505, and CSR-0720796; by equipment support from Sun and IBM; and by financial support from Intel and Microsoft.

[1] Blundell actually uses the term strong *atomicity*, presumably based upon the hardware memory notion of *write atomicity* [12], to describe this property. Larus and Rajwar [17] note that the database terminology for this property is actually *isolation*. We prefer "isolation" because the database community already has an incompatible meaning for atomic, while isolation is unbound in the memory literature.

transactional races are guaranteed to have *sequentially consistent* [16] (SC) behavior. Our use of SI will always refer to Blundell's definition.

Strong isolationists argue that SI is more intuitive than WI, and that it is natural to implement in hardware transactional memory. Weak isolationists have historically agreed about the semantic benefits of SI, but are both unwilling to pay the overhead of SI in a software system and unconvinced that SI is practical in virtualizable hardware.

We argue that, just as parallel programmers expect sequential consistency, transactional programmers expect transactional sequential consistency (TSC)—SC with the added restriction that memory accesses from a given transaction should be contiguous in the total execution order. Neither SI nor WI is strong enough to give this guarantee. Moreover, we doubt that even most strong isolationists will be willing to accept the cost of TSC, due to its requirement of SC for nontransactional accesses.

This dichotomy between *what programmers want* and *what implementers want* is evocative of the arguments of Adve and Hill [7] in their development of the programmer-centric *data-race-free* memory models. Such models are now widely accepted; among other places, they appear in Java [19] and in the emerging C++ standard [10, 19]. Grossman et al. [14], Moore and Grossman [21], Abadi et al. [2, 4], and Spear et al. [28, 30] all suggest that an analogous programmer-centric model may provide the best of both worlds for TM, by affording TSC semantics to transactional data-race-free (TDRF) programs.

Such a model captures the idea that races are likely to be bugs. It enables correct programs and transactional programming paradigms to be ported safely and easily across both languages and hardware platforms, present and future.

With a TDRF model as context, we show that a TM implementation with SI semantics provides no semantic advantage over weaker implementations for data-race-free programs, and little extra benefit for programs with races. Given the cost of SI in software TM, the need for HTM/STM interoperability, and the difficulty guaranteeing SI across context switches even in hardware, we conclude that SI is unnecessary overkill. WI with some implementation restrictions is sufficient to provide the guarantees required by languages like Java. Languages (e.g., C++) willing to further relax the requirements for incorrect programs admit even faster WI implementations.

In Section 2 we provide an overview of the background required for the discussion. In Section 3 we formalize the TDRF model, and show how it can be used to define the same Java-like happens-before consistency model developed by Grossman et al. [14] to constrain the behavior of racy programs. We show that some weakly isolated implementations are compatible with this resulting memory model. In Section 4 we look at the possible advantages of strongly isolated solutions, as well as their costs. Finally in Section 5 we conclude and present future work.

2. Background

We present some simple background that formalizes some of the notions from the introduction that are required for the rest of the paper. Readers familiar with memory models and strong and weak isolation may skip to Section 3.

2.1 Traditional Memory Models

The model of program execution for single threaded code is simple. Program statements appear to execute sequentially in order, with one statement completing before the subsequent statement starts. These intuitive semantics leave a large amount of room for compiler and hardware optimizations.

The natural extension of this model to a multithreaded setting is defined by Lamport as *sequential consistency* [16] (SC). A multithreaded system should appear to execute program statements from each thread sequentially in order, with operations from each thread interleaving in some global total order. Read operations should return the value of the most recent write to the same location in this total order.

SC maintains the semantic cleanliness of the single threaded model, but greatly restricts the potential for optimizations at both the hardware and compiler levels. With very few exceptions, modern systems provide a memory consistency model that is weaker than SC. These models are system-centric and system-specific, depending heavily on implementation details. Writing, understanding, maintaining, and porting parallel code in this setting is difficult and thus extremely costly.

Programmer-centric memory models were developed to make it easier to understand programs and to port them from one system to another. These models specify a contract between the programmer and the system. Acknowledging the semantic benefit of SC, modern memory models are written in Adve's sequentially consistent normal form [5]. If the programmer writes applications that conform to the memory model contract, then the system will appear to be sequentially consistent. The contract is designed in such a manner as to maximize programmability while minimizing optimization barriers.

The programming language community has adopted Adve and Hill's *data-race-free* contract [2] as the basic model for both Java and C++. Using a framework similar to Dubois et al.'s *weak ordering* [13], these models separate memory operations into normal accesses and various forms of synchronization accesses. They then exploit the fact that most of the important single-threaded optimizations only result in observably non-SC behavior in the presence of *data races*—conflicting normal accesses (accesses by different threads to the same location where at least one is a write) that are not

[2] Technically *data-race-free-1* [6].

```
Initially: x = y = 0
```

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

Figure 1. Can `r1 = r2 = 0`? TSC says no, but SI allows the unexpected result.

separated by synchronization accesses. The data-race-free memory model contract specifies that if the program has no data races then the system will appear to be sequentially consistent.

2.2 Strong Isolation

Strong isolation (SI) has two competing definitions in the TM literature. It was originally proposed by Blundell et al. [8] to describe TM implementations in which transactions serialize both with other transactions and with nontransactional accesses. They later refined the definition [9] to describe TM implementations that provide the properties of *containment* and *non-interference* for transactions, without any programmer annotation. Informally, containment means that a transaction is invisible until it commits; non-interference means that nontransactional writes do not appear to occur mid-transaction.

Maessen and Arvind [18] capture the same properties in their ordering rules for transactional memory. We will say that strongly isolated TM implementations provide the guarantee that if a nontransactional access conflicts with any access in a transaction, then it is ordered the same way with respect to all accesses in the transaction. In this framework, containment says that if a transactional write is seen by a nontransactional read, then all accesses internal to the transaction must have happened before the read. Non-interference says that if a transactional read sees a nontransactional write, then the write must have happened before any access in the transaction.

Larus and Rajwar provide a more operational definition [17, p. 27]: a strongly isolated TM implementation behaves as if all nontransactional accesses were single-instruction transactions. This is not equivalent to Blundell's SI given the assumption that transactions are serializable: if every nontransactional access to shared data is a singleton transaction, then all shared data accesses must serialize, resulting in behavior that will always appear to be sequentially consistent. Blundell et al. make no guarantees with respect to nontransactional accesses; in fact, they explicitly state that all TM implementations must chose a traditional memory model in addition to an isolation model. Larus and Rajwar's definition is essentially SI implemented on top of an already sequentially consistent system, resulting in a model equivalent to *transactional sequential consistency* (TSC) as given in Section 3, Definition 1. Our use of SI always refers to Blundell's definition.

Thread 1	Thread 2
1: Node* n = new Node()	Node* n = NULL
2: n->data = job	atomic {
3: atomic {	n = worklist.dequeue()
4: worklist.enqueue(n)	}
5: }	Job* j = n->data

Figure 2. Thread 1 initializes and publishes a list node. Thread 2 privatizes the node and interacts with it nontransactionally.

It is easy to see the difference between SI and TSC. Consider the well known example of Figure 1. 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. On the other hand, SI relies on an underlying memory model to define the outcome, and may well permit an ordering loop.

To the best of our knowledge, all extant implementations of SI assume a relaxed underlying model. In particular, the implementations of Shpeisman et al. [27] and Schneider et al. [24] are realized as extensions to Java, and presumably inherit its relaxed memory model [19]. Similarly Abadi et al. [3] operate in the context of C# with its relaxed model. All of these relaxed models permit the unintuitive result of `r1 = r2 = 0`.

2.3 Weak Isolation

Weak isolation (WI) describes any TM system in which transactions serialize only with other transactions. In general, this means that the system may fail to guarantee containment, non-interference, or both in the presence of transactional/nontransactional races. Furthermore, weakly isolated systems need to be careful not to introduce races that did not exist in the original program. This is particularly important when supporting publication and privatization.

It is natural to use transactions to transition data between logically shared and private states. In Figure 2, Thread 1 privately allocates and initializes a worklist node with a job, publishing it in a transaction. Thread 2 dequeues a node, privatizing it, and privately looks at the associated job. The accesses to `n->data` are conflicting accesses, but not races, as the two threads should not be able to perform them at the same time.

Unfortunately many weakly isolated implementations do not correctly handle publication, privatization, or both, introducing races and unexpected behavior. We describe two recent transactional models that allow both patterns, requiring an implementation to take steps to make sure that it does not create artificial races in otherwise correctly synchronized code. The semantics described by Grossman et al. [14], Abadi et al. [2], and Moore and Grossman [21] all make similar requirements.

Menon et al [20] define transactional `atomic {}` blocks in terms of reductions to Java locks. This maps transactional memory directly into the existing Java Memory Model. If the

transactional program is data-race-free after the reduction, then publication and privatization work as expected.

The most intuitive version of Menon's reduction is single global lock atomicity (SGLA). Transactions must behave *as if* they acquire and release a single global lock on entry and exit. More complicated reductions act as if they acquire per-address locks at various times during their execution. These can admit higher-performance implementations, but sacrifice the guarantee of transactional serializability and require the programmer to understand the sometimes-complex reduction in order to reason about races.

As an alternative to reducing transactional memory to an existing, lock-based memory model, Spear et al. [28] develop memory models based directly on transactions as the fundamental building block. Strict serializability (SS) is presented as the analogue of SGLA. SS says that a thread's non-transactional accesses are ordered with respect to their preceding and following transactions in program order, and that all the transactions in the system have a total global order. As with SGLA, SS has relaxations that provide implementations with more optimization flexibility, however these relaxations maintain the serializability of transactions.

3. Transactional-Data-Race-Free

Writing a parallel program is a large undertaking, even with transactional memory. Blundell et al. [8] showed that strong and weak isolation are incompatible. There exist programs that are correct (for some programmer defined notion of correctness) when executed under strong isolation that are incorrect under weak isolation, and vice versa. In order for transactional memory to become successful, some uniform semantics must be adopted by the community.

The intuition behind Adve and Hill's data-race-free model serves as the basis for the memory models in both Java and C++. We use the same reasoning and structure to develop a simple model that can be used as a basis for a real-world transactional model that Java or C++ programmers can understand and use.

Grossman et al. [14] make the same extension to transactionalize the Java Memory Model's (JMM) definition of happens-before consistency. Unsurprisingly, our model, when extended to a happens-before consistency setting, is identical to theirs. The data-race-free foundation allows us to emphasize transactional sequential consistency as a basis for programming with TM in many different settings.

3.1 TDRF

Transactional programmers expect the system to be sequentially consistent, with the added constraint that accesses within a transaction happen atomically. This suggests the following natural transactional extension of SC.

DEFINITION 1. *A system is* transactionally sequentially consistent *(TSC) if and only if the result of any realizable execution of a program is equivalent to some sequentially consistent execution of the program in which a transaction's accesses occur contiguously in the global total order of accesses.*

TSC is what Larus and Rajwar referred to as strong isolation. It is also equivalent to the *strong semantics* described by Abadi et al. [1] and the *StrongBasic* semantics developed by Moore and Grossman [21].

This notion is also similar to a number of different ideas in the memory model literature. In Shasha and Snir's [26] model a compound operation is very similar to a transaction. Maessen and Arvind [18] provide essentially the same idea in their discussion of store atomicity for transactional memory, but allow local reorderings that are not sequentially consistent. Adve and Hill [7] muse about transactional serialization (essentially TSC) as a possible foundation for their memory models, but dismiss it as too costly.

The following are a simple set of rules extending the *data-race-free-0* [7] model to a setting that relies on transactional consistency as its basis for execution, and uses transactions rather than locks for synchronization.

DEFINITION 2. *Two accesses* conflict *if they access the same location, at least one is a write, and they are executed by different threads.*

DEFINITION 3. *Accesses from an individual thread are ordered by* program order.

DEFINITION 4. *All transactions in the execution are totally ordered by a* transactional order ($<_t$). *If transaction A is transactionally ordered before B then access a in A is transactionally ordered before access b in B.*

DEFINITION 5. *The irreflexive transitive closure of program order and the transactional order defines a* transactional-happens-before ($<_{thb}$) *partial ordering on all accesses in the execution.*

DEFINITION 6. *A* transactional data-race *exists between two accesses in an execution if and only if they conflict and are not ordered by $<_{thb}$.*

DEFINITION 7. *A program is* transactional-data-race-free *(TDRF) if and only if no transactional data races exist in any TSC execution of the program.*

DEFINITION 8. *A TM implementation is TDRF if and only if any realizable execution of a TDRF program produces the same results as some TSC execution of the program.*

Transactional data-race-freedom does not directly constrain the behavior of TM implementations for racy programs. This may be appropriate in languages like C++, but Java has strict safety requirements for all programs. We can easily accommodate Java by using the same happens-before extension that the JMM uses. This extension results in the model of Grossman et al. [14].

```
Initially x = 0
```

Thread 1	Thread 2
1: atomic {	
2: x = 1	
	3: r1 = x
4: abort	

Figure 3. This code fragment depicts a situation where Thread 1 writes x transactionally and then aborts due to contention. Can r1 = 1? THBC disallows the result, but a weakly isolated undo log implementation would exhibit this behavior.

DEFINITION 9. *An execution of a program is* transactional-happens-before consistent *(THBC) if the value returned by every read access in the execution can be explained based on the transactional happens-before ordering in the following way. A read is allowed to return the value of the last write to the same location that is the most recent along some $<_{thb}$ path, or of any write to the same location that is unordered by $<_{thb}$.*

DEFINITION 10. *A TM implementation is THBC if and only if every realizable program execution is THBC.*

A THBC system guarantees TSC to TDRF programs. By guaranteeing THBC to racy program we can avoid behaviors like "thin-air-reads" that are considered dangerous in Java. (We omit Java's treatment of causality for brevity, but assert that THBC fits into the JMM in the same way Java's current happens-before consistency does. Additionally a THBC based JMM will suffer from the same limitations as the current JMM, as discussed by Ševčík and Aspinall [31].)

3.2 Permitted Implementations

While TM generally assumes a speculative implementation to achieve good scalability, transactional memory models do not address speculation directly. Any speculative implementation of transactional memory that guarantees TSC to TDRF programs can be used in a setting where the behavior of racy programs can be undefined. Requiring THBC from racy programs restricts the pool of acceptable implementations but still admits certain weakly isolated implementations that do not instrument nontransactional code.

Weakly isolated, in-place update TM implementations make transactional speculative updates to shared locations directly, and store the valid values in a local *undo-log* so that they can be restored if necessary. Menon et al. [20] and Spear et al. [28] both note that in-place update is fundamentally incompatible with their models, so it should be no surprise that they are incompatible with THBC as well.

Consider a nontransactional read that returns the value of an unordered speculative transactional write that was done in place, as in Figure 3. If that write is later rolled back because the transaction aborts, it does not appear in any thread's history, and thus the read appears to have obtained the value

```
Initially x = 0
```

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

Figure 4. This code fragment depicts a situation where Thread 1 writes x inside a transaction, logging the undo value 0. Thread 2 writes 2 to x. Then Thread 1 aborts due to contention, reverting x to 0. Can r1 = 0? THBC disallows the result, but a weakly isolated undo log would admit this behavior.

```
Initially x = 0
```

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

Figure 5. An example of containment. Can r1 = 1? Strong isolation says no, THBC says yes.

out of thin air, an obvious violation of the TDRF happens-before values-read requirement.

An undo log implementation may also undo the effect of a nontransactional write, if the transaction wrote to the same location and is aborting. This is depicted in Figure 4. In this case the only valid value that the load to r1 can see is 2, as it is the most recent value written to x along a $<_{thb}$ path. In this example, the speculation mechanism introduces a write that never occurs in any program history (the undo write), generating a result incompatible with THBC.[3]

Weakly-isolated, buffered update TM implementations make speculative writes into a private buffer, the redo log, and then propagate those writes to their correct locations once a transaction is guaranteed to commit. Because these writes are never made visible to other threads while they are still speculative, neither of the two problems described above for in-place update systems can occur.

4. Strong Isolation

The argument from strong isolationists has been that SI is more intuitive than WI. SI does constrain the behavior of racy programs more than weaker options. Recall that SI guarantees containment and non-interference; this is true even in the presence of transactional data races. Weaker models at best maintain happens-before consistency, which prevents out-of-thin-air values from being read.

[3] Alternately, we could say that in the realized history the read at line 5 sees the write that created the initial value of 0, but this is not allowed under $<_{thb}$ either, given the intervening "lost" write at line 3.

```
Initially x = y = 0
```

Thread 1	Thread 2
1: atomic {	
2: x = 1	y = 1
3: r1 = y	r2 = x
4: }	

Figure 6. SI is not TSC, as it makes no guarantees about program order.

Consider the code fragment in Figure 5. The two transactional writes of `x` in Thread 1 race with the nontransactional read of `x` in Thread 2. Under THBC, the read in thread 2 may return 0 (the initial value of `x`), 1, or 2, as they are all valid based on the values-read specification (Section 3, Definition 9).

It is easy to see how a weakly isolated buffered-update implementation might actually realize the result `r1 = 1`. The implementation may 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 the read occurs during that period, it will see the partially committed results. This is the same result one would expect using an SGLA reduction.

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.

Figure 6 shows a case where SI fails to prohibit the unexpected result of `r1 = r2 = 0`. SI says only that neither of the instructions in Thread 2 may appear to occur during the execution of Thread 1's transaction. The compiler may chose to reorder the independent instructions in Thread 2 based on its underlying memory model. Likewise, the hardware may reorder the writes on many machines. While TSC disallows the unexpected result, SI clearly does not.

A strongly isolated implementation ensures TSC results for certain racy programs. One could imagine a memory model in which these programs are considered to be properly synchronized. Such a model continues to provide TSC for TDRF programs, but it authorizes programmers to write some forms of racy programs. This is a much more complicated model to reason about: one must decide which races are bugs and which ones aren't.

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

Figure 7. Is this a correct program under an SI-based memory model?

4.1 Access Granularity

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 7. 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.

4.2 TDRF Implementation With SI

While TDRF does not require a strongly isolated TM implementation, it does not exclude one. Grossman et al. [14] point to *sequential reasoning* as an advantage of strong isolation. Given a strongly isolated TM implementation, all traditional single-threaded optimizations are valid within a transactional context, even for a language with safety guarantees like Java. With this in mind we would not discourage development of strongly isolated hardware transactional memories.

This notwithstanding, we note that a standard redo-log based TM implementation with a hashtable write-set meets two of the three properties Grossman et al. attribute to SI, permitting many of the same traditionally unsafe compiler optimizations that SI does, and weakening the potential argument for software SI. Furthermore, recent work of Spear et al. [29] shows that this style redo-log implementations of STM can perform competitively with undo-log implementations.

4.3 Costs

We estimate the overhead of strong isolation using the Delaunay triangulation benchmark distributed with the RSTM framework [25]. This particular implementation of mesh creation makes heavy use of privatization, first geometrically partitioning the space of points and assigning threads independent regions to triangulate, and then using transactions to "stitch up the seams". The application is correct under TDRF as written, as various private/public phases of computation are separated by synchronization barriers.

All of the private uses of data are annotated. By instrumenting just these uses with operations that inspect and modify metadata, we can approximate the cost of a strongly

isolated TM implementation. There are many factors that we cannot measure in our framework which prevent us from making more than a general statement about the relative cost of SI to WI. Shpeisman et al. [23] present a number of aggressive compiler-based optimizations that can be done to reduce the cost of SI, and that we are unable to do in our library implementation. At the same time, our implementation provides precise identification of privatized data, avoiding the need to instrument anything that is never shared.

Aggressive alias analysis can prove that certain locations are not accessed inside of transactions and thus do not need instrumentation. We believe that this analysis would not distinguish our annotated private uses, as they conflict with transactional accesses. It is possible that stronger, barrier aware race detection could eliminate all instrumentation in this circumstance; however we find no evidence of this power in currently tractable alias analysis. It is possible as well that only instrumenting the annotated private use is an overly optimistic decision. Overall we feel that it plausibly reflects a general "best-cast" scenario for alias analysis optimization. Other compiler analysis might be used to amortize instrumentation over multiple nontransactional accesses. The effectiveness of this optimization is likely to be highly application dependent as well. Escape analysis is not applicable here as the objects are all truly shared.

The mesh application spends less than 5% of its total execution time inside transactions. This highlights the cost of SI, and downplays costs related to ensuring that the WI implementation is publication and privatization safe [20, 28]. We argue that phase-based privatization is likely to be an attractive programming idiom in many data-parallel applications. For these the cost of SI instrumentation is likely to be prohibitive.

Our base implementation is a word-based, lazy-acquire, redo-log, timestamp-validating implementation similar in spirit to TL2 [11]. It is augmented with transactional fences at the start and end of transactions in order to ensure publication and privatization safety [28]. Our SI implementation takes the same baseline system, and rather than adding fences, instruments private read and write accesses similarly to [23] (see Figures 8 and 9 for pseudo-code). Note that the instrumentation is somewhat generous: we do not need to check for contention as our code is known to be race free. Hidden in the pseudo-code is the fact that accessing an orec is a volatile operation.

We consider three test platforms: a 2.33 GHz Intel Core2 Duo running Mac OS 10.5.5, a Sun Niagara T1 running Solaris 10, and a 1.2GHz SunFire 6800 also running Solaris 10. We compiled with gcc 4.2.1 on the Core2 machine and 4.2.4 on both of the Suns always using optimization level O3. We triangulate one million points using a single thread, and report private work times that are the average of five runs.

```
write(address,value)
  orec = hash(address)
  old = compare_and_swap(orec, orec, 0)
  address = value
  orec = old
```

Figure 8. The write instrumentation locks the ownership record by writing to it atomically. It then writes the desired datum and restores the original orec. In a real implementation we would need to check to see if our lock was successful, and handle contention if it failed. We would also need to compute a new version.

```
read(address)
  while (true)
    orec = hash(address)
    cache = orec.version
    if (cache.locked)
      continue
    temp = address.value
    if (cache == orec.version)
      return temp
```

Figure 9. The read instrumentation. We need a consistent snapshot of the value, while the orec is not locked. In a real implementation we would need to handle contention if the orec was locked, rather than continuing.

The WI implementation completes in 4.6, 13.7, and 33.9 seconds for the Core2, SunFire, and Niagara machines respectively. Unsurprisingly, instrumenting for SI adds significant overhead, producing execution times of 6.7, 21.6, and 44.6 seconds, respectively—increases of 46%, 57%, and 30%. It appears that the volatile accesses and increased cache pressure of the instrumentation interfere with the more aggressive architectures, limiting their opportunity for optimization. On all three machines, the overhead seems a heavy price to pay for the limited benefits of SI under TDRF.

5. Conclusion

Viewing transactional memory through the lens of a traditional data-race-free memory model allows us to provide the behavior that the programmer expects—transactional sequential consistency—while at the same time allowing high-performance weakly isolated TM implementations. The model also addresses portability, as transactional-data-race-free programs can be run on any conforming TM implementation, including stronger systems should they become available in hardware. Indeed this flexibility is one of the major advantages of TDRF.

Java is of particular interest because it requires more than just TSC execution for TDRF programs: racy programs must appear at least transactional-happens-before consistent. Even this stronger requirement does not necessitate strong isolation, as we have seen that weakly isolated implementations that refrain from making speculative writes visible are compatible.

Strong isolation is overkill for a Java-like model, and the cases where it isn't (i.e., models that guarantee TSC for certain racy programs) are unattractive in both overhead and complexity. If we adopt the position that data races are bugs, then TDRF would appear to be the "right" model for transactional memory, and there is no need for strong isolation.

Future Work The transactional-data-race-free model as given here may be more limiting for implementations than it needs to be. Programmers using publication and privatization correctly know which transactions act to publish or privatize data. Spear et al.'s [28] relaxation from SS to SSS (selective strict serializability) takes advantage of this knowledge, allowing the programmer to indicate which transactions do which activities. Implementations may then use this information to avoid certain kinds of overhead on transactions that are not marked. Abadi et al. [2] propose a similar relaxation, in which the programmer explicitly marks the data that are published or privatized. We plan explore SSS-style relaxation of TDRF ordering, and to evaluate both its potential for performance improvement and its (subjective) cost in notational complexity.

References

[1] M. Abadi, A. Birrell, T. Harris, J. Hsieh, and M. Isard. Dynamic Separation for Transactional Memory. Tr-2008-43, Microsoft Research, Mar. 2008.

[2] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of Transactional Memory and Automatic Mutual Exclusion. In *Conf. Record of the 35th ACM Symp. on Principles of Programming Languages*, pages 63–74, Jan. 2008.

[3] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, New York, NY, USA, 2009. ACM.

[4] M. Abadi, T. Harris, and K. Moore. A Model of Dynamic Separation for Transactional Memory. In *Proc. of the Intl. Conf. on Concurrency Theory*, Aug. 2008.

[5] S. V. Adve. Designing Memory Consistency Models for Shared-Memory Multiprocessors. Ph. d. dissertation, computer sciences technical report #1198, Univ. of Wisconsin–Madison, Jan. 1993.

[6] S. V. Adve and M. D. Hill. A Unified Formulation of Four Shared-Memory Models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[7] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *Proc. of the 17th Intl. Symp. on Computer Architecture*, pages 2–14, May 1990.

[8] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing Transactional Semantics: The Subtleties of Atomicity. In *Proc. of the 4th Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005. In conjunction with ISCA 2005.

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

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

[11] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, pages 194–208, Sept. 2006.

[12] M. Dubois and C. Scheurich. Memory Access Dependencies in Shared-Memory Multiprocessors. *IEEE Trans. on Computers*, 16(6):660–673, June 1990.

[13] M. Dubois, C. Scheurich, and F. A. Briggs. Memory Access Buffering in Multiprocessors. In *Proc. of the 13th Intl. Symp. on Computer Architecture*, pages 434–442, June 1986.

[14] D. Grossman, J. Manson, and W. Pugh. What Do High-Level Memory Models Mean for Transactions? In *Proc. of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, Oct. 2006. In conjunction with ASPLOS XII.

[15] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, pages 289–300, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, Dec. 1992.

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

[17] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2007.

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

[19] J. Manson, W. Pugh, and S. Adve. The Java Memory Model. In *Conf. Record of the 32nd ACM Symp. on Principles of Programming Languages*, Jan. 2005.

[20] 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 *Proc. of the 20th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, pages 314–325, June 2008.

[21] K. F. Moore and D. Grossman. High-Level Small-Step Operational Semantics for Transactions. In *Conf. Record of the 35th ACM Symp. on Principles of Programming Languages*, pages 51–62, Jan. 2008.

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

[23] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic Optimization for Efficient Strong Atomicity. In *Proc. of the SIGPLAN 2008 Conf. on Programming Language Design and Implementation*, pages 181–194, June 2008.

[24] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA 2008 Conf. Proc.*, pages 181–194, Oct. 2008.

[25] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay Triangulation with Transactions and Barriers. In *Proc. of the 2007 IEEE Intl. Symp. on Workload Characterization*, Sept. 2007. Benchmarks track.

[26] 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.

[27] 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 *Proc. of the SIGPLAN 2007 Conf. on Programming Language Design and Implementation*, June 2007.

[28] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Ordering-Based Semantics for Software Transactional Memory. In *Proc. of the 12th Intl. Conf. on Principles of Distributed Systems*, Dec. 2008.

[29] M. F. Spear, L. Dalessandro, V. J. Marathe, and M. L. Scott. Fair Contention Management for Software Transactional Memory. In *Proc. of the 14th ACM Symp. on Principles and Practice of Parallel Programming*, Feb. 2009.

[30] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. In *Proc. of the 26th ACM Symp. on Principles of Distributed Computing*, Aug. 2007. Brief announcement. Extended version available as TR 915, Dept. of Computer Science, Univ. of Rochester, Feb. 2007.

[31] J. Ševčík and D. Aspinall. On validity of program transformations in the java memory model. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 27–51, Berlin, Heidelberg, 2008. Springer-Verlag.

[32] 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 *Proc. of the Intl. Symp. on Code Generation and Optimization*, Mar. 2007.