University of Washington Technical Report: UW-CSE-07-11-01

Towards a Lock-based Semantics for Java STM

Vijay Menon[1] Steven Balensiefer[2] Tatiana Shpeisman[1] Ali-Reza Adl-Tabatabai[1] Richard L. Hudson[1] Bratin Saha[1] Adam Welc[1]

[1]Intel Labs
Santa Clara, CA 95054

[2]Department of Computer Science and Engineering
University of Washington, Seattle, WA 98195

{vijay.s.menon,tatiana.shpeisman,ali-reza.adl-tabatabai,rick.hudson,bratin.saha,adam.welc}@intel.com
{alaska}@cs.washington.edu

Abstract

As memory transactions have been proposed as a language-level replacement for locks, there is growing consensus that software transactional memory (STM) implementations need to provide semantic guarantees at least as strong as locks. In this paper, we investigate the implications of lock-based semantics for transactions. We categorize safety properties imposed by these semantics into two sets: (1) those that maintain proper ordering and (2) those that prevent values from appearing out of thin air. As part of this, we define *publication safety*, the dual of privatization safety [16, 25, 23] and a property most existing STMs violate. For Java, we argue that an STM must respect all these properties to properly adhere to its memory model and maintain safety and security guarantees. Moreover, we also show that a weakly atomic, in-place update STM [3, 1] cannot provide these guarantees. For C++, we reason that only a subset of these properties need to be observed as the behavior of incorrectly synchronized programs can be left undefined. However, we show that violations of others can still lead to non-intuitive behavior in the presence of seemingly benign races and requires STM to impose surprising restrictions on what programmers and compilers are allow to do.

Many in the community have proposed that a *single global lock semantics* [16, 7], where transaction semantics are mapped to those of regions protected by a single global lock, provide the most intuitive model for programmers. In this paper, we present a weakly atomic Java STM implementation that provides these semantics, obeys all the above safety properties, and permits concurrent execution. We also propose and implement two alternative semantics that loosen single lock requirements while still providing privatization safety, publication safety, and other properties. We compare our new implementations to previous ones, including a strongly atomic STM. [23]

1. Introduction

Transactional memory (TM) offers a promising alternative to lock-based synchronization as a mechanism for managing concurrent access to shared data. Over the past decade, TM research has demonstrated how implementers can automatically extract concurrency from declarative critical sections and provide performance and scalability that are competitive with fine-grain locks.

Initially `data = 42, ready = false, val = 0`

Thread 1	Thread 2

`data = 1;`
`atomic {`
` ready = true;`
`}`
	`atomic {`
	` if(ready)`
	`  val = data;`
	`}`

Can `val == 42`?

Figure 1. A correctly synchronized publication example

The primary argument for TM, however, is that it eases concurrent programming. In order to scale, locks require programmers to reason about details such as complex fine-grain synchronization and deadlock avoidance. Moreover, locks provide only indirect mechanisms for enforcing atomicity and isolation, placing a greater burden on the programmer to ensure correctness. TM promises to automate this process. Recent work has shown how TM can be tightly integrated into mainstream languages such as C++, Java, or C# and used as a replacement for lock-based synchronization.

In spite of this, in one respect TM has complicated multi-threaded programming in these languages. In order to write correct multithreaded code, programmers must be able to reason about what observable behaviors are allowable. Outside of TM, there has been significant progress in the past two decades at simplifying multithreaded shared memory programming. A consensus has developed around the notion of *correctly synchronized programs*. If the programmer obeys certain conventions, the program is considered to be correctly synchronized, and the underlying system must provide strong semantic guarantees. Formal memory models provide a contract between programmers on the one hand and compiler and system implementers on the other. For languages such as Java or C#, memory models also limit the behaviours of incorrectly synchronized code to enable stronger safety and security guarantees.

When transactional memory is added to this mix, there is no consensus on how it should impact the language memory model. Many software implementations weaken the underlying memory model in two ways. First, they might have a stricter notion of what a correctly synchronized program means. As a result, a program that is correctly synchronized with locks may no longer be if the lock regions are replaced with transactions. Second, they often provide fewer guarantees on incorrectly synchronized code. Programs with innocuous data races in lock-based code can have surprising and unexpected results in the corresponding transactional code.

```
Initially data = 42, ready = false, val = 0
   | Thread 1         | Thread 2
---+------------------+------------------
1: |                  | atomic {
2: |                  |  tmp = data;
3: | data = 1;        |
4: | atomic {         |
5: |  ready = true;   |
6: | }                |
7: |                  |  if(ready)
8: |                  |   val = tmp;
9: |                  | }
              Can val == 42?
```

Figure 2. Publication with a seemingly benign race

Consider a simple *publication* idiom in Figure 1. The variable data is initially private to Thread 1 and contains a value (42) that should not be accessible by other threads. Thread 1 overwrites this variable (with 1) and only then *publishes* that variable by setting the shared variable ready inside a transaction. Thread 2, inside a transaction, tests the ready variable before accessing the now public data. If atomic is replaced with synchronized, this program would be considered correctly synchronized in Java and in emerging memory models for C and C++. Thread 2 should never observe the private value 42.

Consider the slightly modified program in Figure 2. Thread 2 now reads data early into the private location tmp. However, that value is only used if ready is set. Otherwise, tmp is dead and never accessed again. If the transactions are replaced with locks, it is reasonable to expect the program to run correctly. Nevertheless, most STMs would produce the value 42 in Thread 2 given the interleaving in Figure 2. As Thread 1 writes data outside a transaction, a weakly atomic STM would not invalidate Thread 2's read operation.

From a legalistic perspective, a C or C++ implementation may consider this program incorrectly synchronized as, when Thread 2's transaction is serialized first, there is a clear race on data. However, from the programmer's perspective, the value is never used in this case, and the race should be benign. Java's memory model (assuming lock-based semantics) specifically disallows any execution that produces 42 in Thread 2.

In fact, the situation is worse once we consider reordering introduced by standard compiler optimizations or STM algorithms. A reordering on a correctly synchronized program such as that in Figure 1 can inadvertently introducing a race. A compiler may transform Figure 1 into Figure 2 as part of standard code motion (e.g., partial redundancy elimination [14] in a production compiler would do this if it is deemed safe and profitable). In fact, a lazy versioning STM may essentially introduce an earlier read depending upon its copy granularity. An STM that creates a shadow object [28] upon a transactional write may direct later reads to the copy instead of accessing shared memory.

A clear solution is to provide strong atomicity [3] [1], where non-transactional memory accesses are analogous to single instruction transactions and prevented from violating the isolation of transactions. In this model, transactions are strictly more restrictive than locks and, therefore, provide programmers with sufficiently strong guarantees. However, strong atomicity typically requires either specialized hardware support not available on existing systems, a sophisticated type system that may not be easily integrated with languages such as Java or C++, or runtime barriers on non-transactional reads or writes that can incur substantial cost on programs that do not use transactions.

An alternative approach in the literature is to provide weak atomicity, where no general guarantee is made on non-transactional code, but to augment it to allow idioms such as privatization. In general, this approach is guided by the principle that, if a programming idiom is correct for locks, it should also be correct for transactions. More precisely, there is a notion that transactions should behave as lock-based regions based upon a single global lock [16].

In this paper, we take this approach in a more systematic fashion. We explore the impact of TM on data-race free programs and define the issues that a weakly isolated TM implementation must address in order to preserve the correctness of such programs. We also study the impact of TM on incorrectly synchronized programs, and we discuss how a TM implementation can remain coherent with a single global lock semantics. In some cases, we discovered surprising consequences of forcing TM to adhere to standard memory models. In particular, we make the following high-level contributions in this paper:

- We show that a weak in-place update STM that supports rollback cannot adhere to the Java memory model. In particular, we show that speculative execution can introduce data races into otherwise race-free executions. We believe that a scalable STM for Java must either buffer writes or support strong atomicity to be consistent with its memory model.
- We introduce the notion of publication safety, the dual of privatization. We provide general definitions of both relating to ordering constraints between transactional and non-transactional memory operations. We argue that a Java STM must respect both publication and privatization safety to maintain correct ordering rules.
- We argue that, while a C/C++ STM can avoid handling publication or speculation problems by leaving the semantics of racy programs undefined, this approach requires cooperation and acceptance from programmers and compiler writers. In particular, we show that seemingly benign data races under locks lead to non-intuitive behavior under STM.
- We describe a weakly atomic Java STM implementation that provides *single global lock atomicity* semantics for transactions. Our implementation permits concurrency, but it only allows executions that are consistent (under the Java memory model) with a semantics where all transactions execute under a single global lock. We compare this implementation with previous unsafe weakly atomic and safe strongly atomic Java STMs.
- We describe and implement two weaker semantics: a *disjoint lock atomicity* semantics and an *asymmetric flow ordering* semantics. These semantics weaken the restrictions of single global lock atomicity to allow greater concurrency. While these semantics are not as intuitive, we believe that they still permit a robust subset of programming idioms allowable under single global lock atomicity including the publication safety example in Figure 2.

2. Notions of Correctness

Sequential consistency was originally proposed by Lamport [15] as an ideal model for the correct execution of concurrent programs. A sequential consistent execution can be defined as one whose behavior is compatible with a total ordering over all operations on all threads that also respects program order for individual threads. This ordering specifies the behavior of conflicting memory accesses. For example, a read from a memory location should return the most recent value written into that location in the total ordering. From a

[1] The term *strong isolation* is also used in the literature.

```
        Globally visible java.util.LinkedList list
        Initially list == [Item{val1==0,val2==0}]
    Thread 1                   | Thread 2
    ---------------------------+----------------------------
    Item item;                 | synchronized(list) {
    synchronized(list) {       |  if(!list.isEmpty()) {
     item = (Item)             |   Item item = (Item)
      list.removeFirst();      |    list.getFirst();
    }                          |   item.val1++;
    int r1 = item.val1;        |   item.val2++;
    int r2 = item.val2;        |  }
                               | }
    Can r1!=r2?                |
```

Figure 3. Thread 1 privatizes the previously shared object `item`. Can we safely replace `synchronized` with `atomic`?

programmer's point of view, sequential consistency is both simple and intuitive.

However, hardware and systems generally consider sequential consistency to be prohibitively expensive. It disallows even simple reordering of memory accesses and greatly limits the scope of compiler and hardware optimization. Instead, researchers and implementers have taken a more balanced approach of guaranteeing correct behavior for only a subset of programs that are deemed to be correctly synchronized. In the remainder of this section, we explore two notions of correct synchronization in the context of STM.

2.1 Segregation

One common notion of correct synchronization for STM is that of segregation. A program can be defined as segregated if all mutable shared memory locations are accessed either exclusively inside or exclusively outside a transaction. In this model, mutable shared memory is essentially divided into two: a transactional memory and a non-transactional memory.

For segregated programs, the problems of conflicting transactional and non-transactional code disappears. Moore and Grossman [19] show that, under segregation, strong and weak atomicity are in fact semantically equivalent. Languages such as STM Haskell [9] can enforce segregation in the type system, and, in such environments, a weakly atomic STM implementation essentially provides strong guarantees.

However, for languages such as Java, C++, or C#, such type systems are an open research question and don't yet exist. In STM implementations for such languages, it is typically the responsibility of the programmer to ensure that the program is segregated. From a practical standpoint, segregation is not a natural model of correctness for these languages. First, it is not complete. It provides no guidelines on correctness of conflicting non-synchronized code. Second, it is more restrictive than what is allowed under lock-based synchronization.

As an example, consider the well-known privatization idiom [16, 23, 25] in Figure 3. In this code, an item in a shared list is accessed both inside and outside a synchronized region. However, since the accesses cannot occur simultaneously, this idiom is considered correct in Java [18] and in emerging models for C++ [4, 26]. If transactions are to be a semantic improvement on locks, an STM should provide a guarantee of correctness for a transactional version of this idiom as well.

2.2 Data-Race Freedom

For imperative languages, memory models are being developed to precisely specify ordering relations between synchronizing actions such as lock acquires and releases or volatile reads and writes. Programmers can use these actions to constrain reordering of memory accesses between different threads and, thus, restrict the set of possible executions. For any given execution, memory models define a *happens-before* relation [2] between related synchronization actions on different threads that, in conjunction with program order, transitively establishes a partial ordering over all operations.

This order allows language designers to denote dangerous conflicting accesses in terms of a *data race*. Specifically, a sequential execution contains a data race on a memory location `loc` if and only if there exist two conflicting accesses on `loc` that are not ordered by the happens-before relation. If an execution has no data race on any location, it is a race-free execution. A program is considered *data-race free* if and only if every valid sequential consistent execution is data-race free. For example, the program in Figure 3 is data-race free. For any valid sequential execution of the program, there is a happens-before relation (via synchronization actions and single-thread ordering) between conflicting field accesses of the item in Threads 1 and 2. It is important to emphasize that data-race freedom is a dynamic property. Statically, the programs in Figures 1 and 3 may appear to be problematic, but, dynamically, no race occurs in any valid execution.

There is an emerging consensus around data-race freedom as a notion of correctness for imperative languages. More specifically, recent language memory model work aims to guarantee that any execution of a data-race free program will be sequentially consistent. For compiler and hardware implementers, this offers considerable flexiblity for program optimization while still preserving strong correctness guarantees. From an STM standpoint, this provides significant challenges and surprising restrictions, as we shall see in the next sections.

2.3 Correctness in the Presence of Data-Races

Language memory models differ significantly on the guarantees they provide for incorrectly synchronized (i.e., non data-race free) programs. In Java, concerns of type safety and security are paramount, and, while there is no guarantee of sequential consistency, there are still strong restrictions on behavior. In C++ and other non-managed language environments [4, 26], there appears to be a consensus towards a *catch-fire* semantics, where the presence of a single data race removes all guarantees and constraints on behavior.

From an STM perspective, the practical consequence of this is that implementations for "safe" (e.g., Java) and "unsafe" (e.g., C++) languages must offer fundamentally different guarantees. In Section 3, we will highlight properties that STM's must preserve for both safe and unsafe code.

3. Safety Properties for STM

For STM implementations to provide semantic guarantees that are at least as strong as locks, they must preserve sequentially consistent semantics for data-race free programs. Additionally, Java implementations must also make guarantees for racy programs.

In particular, Java makes two primary requirements. First, program executions must respect both program order (within a single thread) and synchronization order (across multiple threads). This ordering requirement ensures correctness for properly synchronized programs. Second, program executions must not create values *out of thin air*. Informally, this ensures that a memory read will return the value written by some memory write in that execution (where, to avoid cycles, that write itself was not control dependent on the read). In this section, we will discuss how memory transactions impact these requirements, and focus upon those issues that are often problematic for STM implementations.

[2] We use Java memory model terminology here. Other emerging memory models provide a similar ordering relationship.

	Thread 1	Thread 2
1:	`atomic {`	
2:	`S1;`	
3:	`}`	
4:		*[synchronizing action]*
5:		`S2;`

Figure 4. A privatization safety template where a transaction and synchronizing action are ordered and `S1` and `S2` conflict.

	Thread 1	Thread 2
1:	`S1;`	
2:	*[synchronizing action]*	
3:		`atomic {`
4:		`S2;`
5:		`}`

Figure 5. A publication safety template where a synchronizing action and transaction are ordered and `S1` and `S2` conflict.

3.1 Maintaining Ordering

Java STM implementations must respect a *happens-before* relation induced by the Java memory model between two operations. In general, STMs do not directly affect ordering between two non-transactional memory operations. Any ordering requirements between non-transactional accesses are left to existing Java mechanisms. STMs also are designed to properly handle interactions between two transactional operations. If two transactional operations conflict, the STM will detect this and ensure that the transactions are properly ordered. Unsurprisingly, difficulties typically arise when non-transactional accesses are ordered with respect to transactional accesses. We break down the corresponding ordering guarantees into two categories.

3.1.1 Privatization Safety

The *privatization* problem [16, 24] is well-known among STM researchers. Figure 3 illustrates the classical privatization problem when `synchronized` is replaced by `atomic`. In this section, we generalize this as follows.

We define *privatization safety* as the requirement that an STM must respect a happens-before ordering relation from a transactional access `S1` to a conflicting non-transactional access `S2`.

Figure 4 illustrates an execution template for privatization safety where Thread 1's transaction is executed first, followed by Thread 2's synchronizing action (e.g., another transaction), and then the non-transactional operation `S2`. If the transaction and synchronizing action are ordered by our semantics, then there is clearly a happens-before relation from `S1` to `S2`. For example, if `S1` is a write to `x`, `S2` is a read from `x`, and there is no intervening write to `x`, then `S2` must read the value written by `S1`.

Intuitively, the term privatization reflects how this idiom may be used correctly in a data-race free program. The memory location `x` is shared when accessed by `S1` but private to Thread 2 when accessed by `S2`. An intervening privatizing action ensures that the location is not accessible by another thread. Figure 3 gives an example of this where an item is removed from a shared list.

When STM implementations allow conflicting transactions to overlap, they must take extra precautions to respect privatization safety. In a write buffering STM, Thread 1's transaction is ordered before Thread 2's transaction if they conflict and Thread 1 reaches its linearization point (typically the point of final validation) first. However, because stores are buffered, Thread 1's modified values are typically written after this linearization point. To avoid introducing a race, the STM must ensure that these writes are visible to S2. In an optimistic in-place update STM, a similar ordering violation may occur, but now due to an aborted transaction. In this case, Thread 1's aborted transaction (and undo write) happens-before Thread 2's successful one (as it read earlier state), and there must be an ordering between the undo write in Thread 1 and any conflicting access in `S2`.

As these violations can occur in correctly synchronized programs (as in Figure 3), there is general consensus that STM implementations must respect privatization safety for both unmanaged languages such as C++ and managed languages such as Java. A number of solutions have been recently proposed in the literature [24, 25, 27].

3.1.2 Publication Safety

We term the dual to privatization as publication. Just as an ordering from a transactional access to a conflicting non-transactional access must be respected, so too must the reverse.

We define *publication safety* as the requirement that an STM must respect a happens-before ordering relation from a non-transactional access `S1` to a conflicting transactional access `S2`.

Figure 5 shows an execution template for publication safety where `S1` is executed first, followed by Thread 1's synchronizing action (e.g., a transaction), and then Thread 2's transaction. If the synchronizing action and transaction are ordered, there is a transitive ordering from the non-transactional operation `S1` to the transactional operation `S3`.

As before, the term publication reflects how this idiom may be used in practice. In this case, a memory location `x` may initially be private when accessed by `S1`, published by the following transaction, and public when accessed by `S2`. Figure 2 shows a more concrete example of publication safety and its potential violation. Suppose we have an execution where Thread 1's transaction is ordered before Thread 2's. In this case, the execution is data-race free as there is an ordering between Thread 1's write to `data` and Thread 2's read. In this ordering, the final value of `val` must be 1. However, an STM that allows Thread 2 to read data before Thread 1 begins can inadvertently introduce a race on `data` and let Thread 2 illegally read a private value (`data == 42`) from Thread 1 and write it to `val`. This interleaving can affect both in-place update and write buffering STM implementations.

With privatization, violations may occur because transactional write operations can be delayed by an STM. With publication, violations may occur because transactional read operations may be speculated early. In the latter case, however, the program itself has a data race. In Figure 2, an execution where Thread 2's transaction preceeds Thread 1's has a race on `data` (albeit a seemingly benign one if `tmp` is dead), and thus the program itself is not correctly synchronized.

In contrast, the similar but correctly synchronized program in Figure 1 does not suffer a publication problem with the same interleaving. Because of this, an STM implementation can seemingly ignore publication-safety, but only under the following conditions:

- The programming language does not guarantee correct execution in the presence of benign races. For example, it does not allow the programmer to speculatively read `data` above the conditional as in Figure 2.
- The compiler does not speculatively hoist memory operations onto new program paths inside a transaction (e.g., during code motion or instruction scheduling). If it does, it could hoist the correctly synchronized read of `data` in Figure 1 and introduce a data race as in Figure 2.
- The STM itself does not introduce speculative reads of data inside a transaction. For example, STM implementations that cre-

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

	Thread 1	Thread 2	Thread 3
1:	`atomic {`		
2:		`atomic {`	
3:		// open read x, y	
4:	`x++;`		
5:		`if(x != y)`	
6:		`z = 1;`	
7:			`z = 2;`
8:	`y++;`	`*abort*`	
9:	`}`	`}`	

Can `z == 0`?

Figure 6. A observable consistency example

ate shadow copies of an object or a cache line on write essentially introduce early reads of non-written fields. In Figure 1, a write to a different field above the conditional in Thread 2 must not also introduce a speculative read of `data` that is later used inside the conditional. This type of speculative read has been previously referred to as a granular inconsistent read. [23]

The combination of these conditions prevent the early execution of a read that can in turn lead to an ordering violation. This may be acceptable in the context of emerging memory models for C or C++. In the context of Java, however, the first condition is itself too strict. The Java memory model clearly limits the effect of benign data races. Moreover, in race-free executions of racy-programs (as in Figure 2), sequential consistency is still expected. Thus, Java STMs must explicitly disallow publication safety violations.

3.2 Preventing Values Out of Thin Air

In addition to respecting ordering constraints, STM implementations must avoid allowing values to appear out of thin air. Here, we investigate three different safety properties that, when violated, can lead to values appearing out of thin air. The first two properties are essential to preserving the correctness of race-free programs and should be respected by all STM implementations. The last is an issue for incorrectly synchronized programs and leads to important restrictions on Java STM implementations.

3.2.1 Observable Consistency

An STM observes *observable consistency* if it only permits side effects to be observable by other threads if they are based upon a consistent view of memory.

This condition is a standard STM requirement in unmanaged platforms. In this context, an inconsistent memory access can lead to a catastrophic fault (e.g., a read or write to protected memory) that would never have occurred in a lock-based program. In this section, we also argue that it is an essential requirement for preserving the correctness of race-free programs, and, thus, a strict safety condition that managed STM implementations must also respect.

Consider the program in Figure 6. It is clear that, outside Thread 1's transaction, the values for `x` and `y` should always be equivalent. Consider, however, an execution by an optimistic in-place update STM with weak isolation. [1, 10]. In the illustrated interleaving, Thread 2 records version numbers for `x` and `y` without locking those locations. Thread 1 then locks and writes `x`. Thread 2 then continues with the updated `x` and original `y` and writes `z`. The condition succeeds and the write to `z` happens only becauses Thread 2 is viewing an inconsistent state of memory. Thread 2 will eventually abort when the transaction validates and undo its write to `z`.

For an unmanaged platform, this is already insufficient. The write to `z` could result in a program fault, and it never would have occurred in a non-speculative execution. Recent STMs for unmanaged platforms prevent this by enforcing a consistent view of memory before a transactional access. [27]

For a managed platform, faults are not problematic. An error due to an invalid memory access or execution of an illegal operation is converted into an exception that an managed STM can lazily validate and recover from. Nevertheless, inconsistent writes are problematic. The definition of a correctly synchronized program is dynamic, and, even though Thread 2 contains an access to `z`, it can never occur in a sequentially consistent execution. Thus, this program is correctly synchronized by the standard definition of data-race freedom. In this example, our execution introduces a data-race on `z` with Thread 3 and can lead to incorrect results.

Because of this, Java STM implementations must ensure that transactional writes are only observable by other threads if they reflect a consistent view of memory.

Initially `x.g==0`

	Thread 1	Thread 2
1:	`atomic {`	
2:	`x.f=1;`	
3:		`x.g=1;`
4:	`}`	

Can `x.g==0`?

Figure 7. A granular safety violation.

3.2.2 Granular Safety

Granular safety is a well-understood issue in STM implementations [23]. Granular safety requires that transactional accesses to one location do not adversely affect non-transactional accesses to an adjacent location by essentially inventing writes to those locations out of thin air. If an STM implementation's granularity of buffering or logging subsumes multiple fields, it must avoid observable writes to fields not explicitly written to in the original program.

Figure 7 illustrates an example of a granular lost update taken from Shpeisman et al. [23]. In a buffering implementation, Thread 1 must not overwrite `x.g` on commit. In an in-place update STM, Thread 1 must not revert `x.g` on abort. In either case, failure may result in losing Thread 2's update.

Granular safety is clearly a requirement to handle correctly synchronized programs and should be respected by all STM implementations.

3.2.3 Speculation Safety

The last property that protects against out-of-thin-air values is *speculation safety*. Violations of this property lead to speculative dirty reads and speculative lost updates [23]. In some cases, this leads to data races in otherwise race-free executions.

Consider the program in Figure 8 executed by an in-place-update STM. The transaction in Thread 2 initially reads a value of `0` in `x`, and updates the value of `y`. This transaction, however, is later aborted and restarted. In the process, however, it may clobber the write of `y` in Thread 3 and set the value back to `0`. On the second execution, it sees that `x == 1`, and it writes `z = 1` instead and commits.

Note that there is no valid sequential execution where Thread 2 writes both `y` and `z`. The result `z = 1` is therefore only consistent with a race-free execution where no race exists on `z`. The speculation in Thread 2, however, created a new value (a write of `0`) out of thin air. To be consistent with the Java memory model, an STM implementation must guarantee speculation safety to avoid creating a data race in a race-free execution.

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

	Thread 1	Thread 2	Thread 3
1:	`atomic {`		
2:		`atomic {`	
3:		`if(x == 0)`	
4:		`y = 1;`	
5:		`else`	`y = 2;`
6:		`z = 1;`	
7:	`x = 1;`		
8:	`}`		
9:		`*abort*`	
10:		`}`	

Can `z == 1` and `y == 0`?

Figure 8. A speculation safety example

Note, however, that this program is not a race-free program even though the considered execution does not contain a race. For memory models that provide no guarantees on a race-free program, there is no need to respect speculation safety. Any bad behavior can instead be attributed to some racy execution. In contrast, Java's strong guarantees still disallow any execution where, for our example, `z == 1` and `y == 0`.

Speculation safety is not an issue for data-race free programs in an STM that enforces observable consistency. Informally, we make the following argument. Because of consistency, a racy access introduced by speculation must be part of some valid execution of that transaction. Suppose, after the speculative execution of that racy access, we suspend all other transactions. In our new execution, the speculative transaction should commit and complete. If the transactional access was racy in the original execution, however, it it is still racy in the modified execution. Thus, with observable consistency, if a speculative execution introduces a data race, the program must not be data-race free.

3.3 Discussion

We argue that all of the safety properties discussed in the section must be preserved by a Java STM. To date, the only scalable STM implementations that meet this criteria are strongly atomic [23]. In Section 5, we will present a weakly atomic write-buffering STM that also respects these properties. A important consequence of the discussion here is that a weakly-atomic in-place-update STM is not compatible with the requirements of the Java memory model.

If C and C++ communities indeed converge to memory models are allowed to *catch fire* on racy programs, then the corresponding restrictions on STM implementations for these languages is much weaker. Speculation safety is optional (for reasons discussed above), and publication safety is optional (with the caveats listed above). A STM implementation that respects observable consistency, granular safety, and privatization safety would be sufficient for these languages.

4. Transactional Semantics

While the safety properties in the previous section are generally applicable to STMs, we have left open the exact semantics of memory transactions, especially with respect to ordering, until here. Precise ordering rules are necessary, for example, to determine whether a publication or privatization safety violation occurred. In the context of Figures 4 and 5, these rules tell us when a transaction must be ordered with respect to another transaction (or synchronizing action). If the semantics requires no ordering, there is no corresponding privatization or publication safety constraint.

Initially `data = 42, ready = false, val = 0`

	Thread 1	Thread 2
1:		`atomic {`
2:		`tmp = data;`
3:	`data = 1;`	
4:	`atomic { }`	
5:	`ready = true;`	
6:		`if(ready)`
7:		`val = tmp;`
8:		`}`

Can `val == 42`?

Figure 9. Publication via empty transaction

Initially `data = 42, ready = false, val = 0`

	Thread 1	Thread 2
1:		`atomic {`
2:		`tmp = data;`
3:	`data = 1;`	
4:	`atomic {`	
5:	`test = ready;`	
6:	`}`	
7:		`ready = true;`
8:		`val = tmp;`
9:		`}`

Can `test == false` and `val == 42`?

Figure 10. Publication via anti-dependence

As examples, consider Figures 9 and 10. In either case, the transaction in Thread 1 is only part of a publishing action if that transaction must be ordered with the one in Thread 2.

Other researchers [16, 8, 7] have proposed various semantics for transactions and how they interact with existing language features. In this section, we consider a few different proposals and introduce a new semantics that is weaker than locks.

4.1 Single Global Lock Atomicity (SGLA)

The simplest and most intuitive semantics is to consider transactions as if executed under a single global lock. [16, 7] We refer to this semantics as *single global lock atomicity* (SGLA). With this model, we can define the semantics of a transactional program to be equivalent to a corresponding non-transactional program where every transactional region is converted as follows:

```
atomic { S; } → synchronized (global_lock) { S; }
```

SGLA is appealing for a number of reasons. First, it matches our natural understanding of transactions. It provides complete isolation and serializability over all transactions. Second, it provides sequentially consistent semantics for correctly synchronized code. It supports correct behavior for race-free idioms in Figure 1 and 3. Third, combined with the strong guarantees of the Java memory model, it provides an intuitive behavior for programs with races. For example, it tolerates the benign race in Figure 2 and prevent the private value from leaking to another thread. Finally, it leverages years of research into the semantics of lock-based synchronization.

On the other hand, SGLA arguably imposes ordering in situations where a programmer might not actually expect one. Revisiting our two examples in Figures 9 and 10, SGLA restrictively imposes an ordering between the transactions on the left and on the right in both cases.

4.2 Disjoint Lock Atomicity (DLA)

An alternative model is to only impose an ordering between transactions that conflict. We refer to this semantics as *disjoint lock atomicity* (DLA). With this semantics, we denote two transactions as conflicting if there exists some memory location `loc` where one transaction writes to `loc` and the other either reads or writes to it. If no such conflicting access exists, the transactions do not have to be ordered with respect to each other. If a conflict exists, the transactions are ordered, and a happens-before relation exists from the end of the first to the beginning of the second. Similar semantics are suggested by Harris and Fraser [8] and Manson, et al. [7]. [3]

Intuitively, under DLA, any transactional execution has a semantically equivalent lock-based one where each transaction is protected by some minimal set of locks such that two transactions share a common lock if and only if they conflict. From an ordering perspective, all locks are acquired in some canonical order (to avoid deadlock) at the beginning of a transaction and released at the end.

DLA is somewhat more difficult to reason about than SGLA. In contrast to SGLA, we cannot statically construct an equivalent lock-based program with DLA. First, our semantics are dynamic. The disjoint "locks" must be determined at runtime to reflect the data accessed in a given execution. Two different executions of the same transaction may touch completely different locations. Second, our semantics are prescient. The "locks" must be acquired in some canonical order at the beginning of the transaction to (a) avoid deadlock and (b) ensure that conflicting transactions do not overlap from the perspective of external observers (e.g., via non-transactional memory accesses). In practice, for non-trival transactional regions, it is undecidable to determine what data and locks would be required ahead of time.

Nevertheless, we believe that DLA is still relatively intuitive for programmers to reason about. As with SGLA, it provides familiar guarantees for programmer familiar with locks and leverages years of research into the semantics of locks. We believe that for race-free programs, SGLA and DLA allow for the same set of observable behaviors. On the other hand, DLA provides weaker guarantees in the presence of races. For example, it allows the results prohibited by SGLA in Figure 9. But, it provides equivalent guarantees for the racy programs in Figures 2 and 10.

4.3 Weaker Than Locks

Manson et al. [7] propose two different semantics for transactions that are weaker than DLA or SGLA. In their first proposal, which they label a $write \rightarrow^{hb} read$ semantics, two transactions are ordered by a happens-before relationship only if the first writes a value read by the second. Intuitively, transactions are only ordered if a data may *flow* from the first to the second. In this way, transactions behave similarly to volatile accesses, where a volatile write can only be source of a happens before relation, and a volatile read can only be the target. In their second proposal, they strengthen these semantics with an additional *prewrite* rule that prohibits certain executions. Unlike DLA or SGLA, neither semantics require an implementation to disallow the results shown in Figure 10. Under these semantics, a read-only transaction cannot act as publishing action, and, as far as we are aware, this is consistent with common usage.

On the other hand, both semantics appear to allow nonintuitive results for privatization. In Figure 3, Thread 2 executes before Thread 1, and both have conflicting accesses to `list`. However, Thread 2 only reads `list` to access its first `item`. In both of the weaker semantics proposed by Manson et al., there is no happens-before relation between the two transactions, and Thread 1 cannot assume private access to `item` once it is removed from the list. Because of this, we do not believe these semantics are strong enough for languages such as C or Java that allow privatization under locks.

[3] DLA is essentially the same as the weakest "conflicting regions" semantics defined by Manson, et al.

4.3.1 Asymmetric Flow Ordering (AFO)

The above semantics provide generally uniform ordering rules for memory access operations regardless of whether they are transactional or non-transactional. Transaction boundaries are used to establish happens-before and other relations between accesses, but once this is done, the ordering constraints are the same regardless of whether an access is transactional. Moreover, the corresponding restrictions on the values a read operation may observe are the same regardless of whether the read is transactional. Because of this, semantics that weaken publication ordering (and the values read by corresponding transactional reads) also inadvertantly weaken privatization ordering (and the values of non-transactional reads).

We can avoid the problem above by introducing different rules on observable values for transactional and non-transactional read operations. We refer to these semantics as *asymmetric flow ordering* (AFO). Informally, these semantics are similar to DLA, but weakens the constraints on transactional reads to permit them to see earlier values under certain conditions.

More formally, we can define AFO in terms of the happens-before ($\rightarrow^{hb}$) and prewrite ($\rightarrow^{pw}$) relations defined by Manson et al. [7] for their prewrite ordering semantics discussed above. In contrast to prewrite ordering, however, we enforce stricter rules on observable values for non-transactional reads. Under AFO, a non-transactional read r may observe a write w unless $r(\rightarrow^{hb} \cup \rightarrow^{pw})w$ or there exists a w' such that $w(\rightarrow^{hb} \cup \rightarrow^{pw})w'(\rightarrow^{hb} \cup \rightarrow^{pw})r$. On the other hand, a transactional read r may observe a write w unless $r(\rightarrow^{hb} \cup \rightarrow^{pw})w$ or there exists a w' such that $w(\rightarrow^{hb} \cup \rightarrow^{pw})w' \rightarrow^{hb} r$.

From a practical perspective, this allows a transactional read to observe the value of an earlier write w and ignore an intervening w' under certain conditions. More generally, this does not allow a read-only transaction to act as a publishing action as suggested by Figure 10, but does support the publication idioms in Figures 1 and 2 as well as all privatization idioms.

As far as we are aware, there are no other published semantics that are weaker than DLA yet strong enough to support common privatization idioms. While AFO is admittedly more complex than DLA or SGLA, it appears to correctly handle idioms that reflect common usage with fewer additional constraints. In the next two sections, we will discuss how this affects the scalability of STM implementations.

4.4 Interactions in the Language

4.4.1 Native methods, I/O, and Irrevocable Actions

Lock-based semantics provide a natural semantics for native methods, I/O, or other difficult-to-undo actions inside transactions, should we choose to allow them. Implementing semantics for such actions, however, may require falling back onto a single global lock or other specialized behavior. [2] These are largely orthogonal to the issues discussed in the paper, and we do not explore this further here.

4.4.2 Volatiles and visibility

Lock-based semantics do give us flexibility when considering interactions with other synchronization constructs. For example, the values of volatile writes are typically made immediately visible to other threads. When volatile writes are embedded inside transactions, it is not clear whether visibility of the volatile should

```
Initially x=y=0, x,y are volatiles
Thread 1                 | Thread 2
-------------------------+----------------------
atomic {                 |
 S1: x=1;                | S2: while (x == 0) {}
 S4: while (y == 0) {}   | S3: y = 1;
}                        |
```

Figure 11. Volatiles in transactions: is it legal for this program to deadlock?

take precedence over the isolation of the transaction from non-transactional code; see Figure 11. The Java memory model makes no fairness guarantees, however, and it is thus legal to publish volatile writes only when the transaction commits.

Alternatively, we can treat volatile accesses as I/O operations as discussed above. In this case, volatile writes may become immediately visible, but at the cost of limiting concurrent execution of other transactions.

4.4.3 Advanced Transactional Features

Unsurprisingly, lock-based semantics do not provide a natural semantics for transactional language constructs beyond `atomic`. For example, they define no intuitive semantics for explicit rollback (e.g., `retry` and `orelse` [9]) or for selective violation of transactional isolation (e.g., open nesting [20]). For now, we leave these features to future work.

5. A Weakly Atomic Implementation

In this section, we present a weakly atomic Java STM system that enforces lock-based semantics for both correctly and incorrectly synchronized programs. Our TM implementation builds upon an earlier software stack [1, 23] that provides both weakly and strongly isolated update-in-place transactional memory implementations. As with those implementations, we rely on optimistic read versioning, encounter time 2-phase locking for writes, read-set validation prior to commit, and conflict detection at either an object or block granularity. However, we fundamentally altered our new TM implementation to enforce the safety properties discussed in the previous section.

5.1 Write Buffering

Our weakly atomic implementation is a canonical write-buffering STM similar to those described by Harris and Fraser [8] and in TL2 [5]. Each transactional write is buffered into thread local data structures. A transactional read must check these data structures before accessing the shared heap to ensure that it obtains the correct value. Writes acquire a lock at encounter time and record that fact. Reads add a new entry to the transactional read set. Before a transaction commits, it must validate the read set. If the transaction is valid, it copies all buffered data into the shared heap and releases all write locks. Note, to provide granular safety, only modified locations are written to the heap. If the transaction is invalid, it discards its local buffer and restarts the transaction.

As in TL2 [5], we use a global linearization timestamp, and we generate a local timestamp for each transaction immediately prior to validation by incrementing the global one. TL2 assumes a segregated memory (as described in Section 2), and in this case, the local timestamp ordering directly reflects the serialization order of the transactions. The point at which a transaction generates its local timestamp is sometimes referred to as its *linearization point*.

Unlike TL2, we do not use this timestamp to enforce observable consistency prior to potentially unsafe operations (such as reads or writes to the heap). In Java, this is unnecessary as exception handling allows us to trap all faults (i.e., no faults are catastrophic), and we can rely on buffering of writes to delay those operations until the state is consistent. Instead, we use timestamps to enforce publication and privatization safety, as discussed below.

We use a commit log to buffer writes. To ensure that field reads within transactions see the correct values, we must also add code that checks the commit log for any writes to the same address. Because we use encounter-time locking, this only needs to be done when we read data that we've already locked.

Our basic write buffering implementation guarantees that values do not appear out of thin air. It maintains observable consistency, granular safety, and speculation safety. Below, we discuss how we augment our implementation to enforce ordering properties.

5.2 STM Barriers for Volatiles and Locks

As in Harris and Fraser [8], we enforce ordering on non-transactional volatile and lock operations by essentially treating them as single operation transactions. For example, to execute a non-transactional synchronized region, we acquire the lock as a transaction, execute the body non-transactionally, and release the lock as a transaction.

```
Initially x=y=0, y is volatile
Thread 1   | Thread 2
-----------+-------------
atomic {   |
 x=1;      | if (y == 1)
 y=1;      |  t=x;
}          |
      Is t==0 legal?
```

Figure 12. Privatization via a volatile

Treating all synchronizing actions as transactions simplifies our implementation and, below, our enforcement of publication and privatization safety. Figure 12 illustrates a privatization example where ordering is imposed by Thread 2's volatile read of `y`. Because we treat this read as, essentially, a small transaction, it matches our template in Figure 4 and is handled by our solution below.

5.3 Commit Linearization for Privatization

In a write buffering STM, writes are performed after the linearization point. Unless we prevent it, these transactional writes can inadvertantly race with non-transactional accesses on another thread leading to privatization safety violations. For example, in Figure 3, Thread 2's writes to `item.val1` and `item.val2` are made visible after the linearization point and, thus, may race with non-transactional reads in Thread 1.

Privatization solutions for write buffering have been proposed in the literature [25]. Solutions that avoid non-transactional barriers essentially implement *commit linearization*, a simple form of quiescence that ensures that transactions complete in linear order.

In our implementation, shown in Figure 13, we adopt this approach by recording when a transaction has reached its linearization point in a shared data structure (`commitStampTable`). After a transaction has published all its buffered values to shared memory, it signals that it is done by setting its entry to `MAXVAL`, it releases its locks, and it iterates over other threads in the system waiting for their completion. On abort, it does not have to wait as no transactional writes are actually published, and there is no ordering to enforce.

Commit linearization enforces privatization safety by creating an explicit ordering from the end of a transaction to any following synchronizing action on another thread (i.e., the privatizing action), and transistively, from a transactional write (performed before transaction end) and a non-transaction read (performed after

```
TxnCommit(desc){
  mynum = getTxnNextLinearizationNumber();
  commitStampTable[desc->threadid] = mynum;

  if(validate(desc)) {
    // commit: publish values to shared memory
    ...
    commitStampTable[desc->threadid] = MAXVAL;
    // release locks
    ...
    Quiesce(commitStampTable, mynum);
  } else {
    commitStampTable[desc->threadid] = MAXVAL;
    // abort : discard & release
    ...
  }
}

Quiesce(stampTable, mynum) {
  for(id = 0; id < NumThreads; ++id)
    while(stampTable[id] < mynum)
      yield();
}
```

Figure 13. Commit linearization

```
TxnStart(Descriptor* desc) {
  mynum = getTxnNextStartNumber();
  startStampTable[desc->threadid] = mynum;
  ...
}

TxnCommit(desc){
  startStampTable[desc->threadid] = MAXVAL;
  Quiesce(startStampTable, mynum);
  ...
}
```

Figure 14. Start linearization

the privatizing action). Because we treat volatiles and lock operations transactionally, our approach safely covers any ordering action.

5.4 Start Linearization for Publication Safety

In a write buffering STM, read operations are performed before the linearization point. Although they are validated afterward, a weakly atomic STM's validation process will not detect conflicts due to non-transactional accesses. For example, Thread 1's transactional read of `data` in Figure 2 is performed before Thread 1's transaction (which linearizes first), but validation will not detect the conflict with Thread 1's non-transactional write of the same field.

To enforce publication safety, we implement *start linearization*. As above, start linearization is another form of quiescence. In this case, however, we ensure that start order matches linearization order. We set the linearization timestamp when a transaction begins. When a transaction reaches its linearization point, it must wait its turn to proceed. This ensures that it will not indirectly publish the result of a non-transactional write. In Figure 2, Thread 1's transaction will wait on Thread 2's transaction before it linearizes, which, in turn, prevents Thread 2 from reading an updated value from `ready` (which would still be in Thread 1's buffer).

Figure 14 provides high-level pseudocode for start linearization. Start linearization enforces publication safety by creating an explicit ordering from a synchronizing action (i.e., the publishing action) on one thread and the start of a transaction on another. This, in turn, enforces a runtime ordering between a non-transactional access (before the publishing action) and a following conflicting transactional access (after transaction start).

5.5 Enforcing Disjoint Lock Atomicity

We argue that the above mechanisms provide disjoint lock atomicity semantics for transactional Java programs. Write buffering ensures that values are never created out of thin air. Commit and start linearization enforce privatization and publication safety and avoid ordering violations.

More subtly, transactions that conflict are properly ordered. For these transactions, the start order, linearization order, and commit order are the same. To understand this, consider the point in `TxnCommit` immediately after a transaction quiesces on the `startStampTable` but before it acquires a linearization number. If one transaction starts before the second but commits after it, due to start and commit linearization, both transactions must have simultaneously been at this point. In other words, the first transaction quiesced first on `startStampTable`, but the second overtook it acquire an earlier linearization number. However, if two transactions conflict then a record in the write set of one must be in either the write set of the other (which cannot happen simulataneously) or the read set of the other (which will trigger a validation failure and abort).

The start and linearization numbers for transactions combine to define a partial ordering. If the start number and linearization number of one transaction are less (greater) than the corresponding start number and linearization number of another, then the first is ordered before (after) the second. If the start and linearization orders do not match, the transactions are not ordered with respect to each other and, as argued above, cannot conflict.

In a race-free program, our implementation ensures sequential consistency. Intuitively, our base STM ensures that no data conflicts exist between overlapping transactions. Additionally, it ensures that start, linearization, and commit points are already in the properly order for conflicting transactions. Thus any interleaved execution can be transformed to an equivalent non-interleaved execution that respects the partial order above. This transformation can be accomplished by a series of transpositions of adjacent operations. The adjacent operations that must be swapped are guaranteed to be independent (by the STM if transactional and by data-race freedom if not) and, thus, the transformation does not alter the semantics of the original execution.

In a racy program, our implementation still observes the Java memory model. By similar reasoning, we can again convert an interleaved execution to an equivalent non-interleaved one. In this case, however, the Java memory model [18] gives us the flexibility to transpose conflicting memory accesses without altering the values that they read or write when the accesses race. In other words, a read may return the value of any write with which it has no happens-before relationship; it is not restricted to the value of the last write in the execution.

5.6 Enforcing Single Global Lock Atomicity

The mechanisms described above are not sufficient to enforce single global lock atomicity. Consider the example in Figure 15. Since the transactions are independent, then, as described above, Thread 1's transaction may start after Thread 2's, but complete before. As a transaction's reads occur before the linearization point and writes occur after, this can effectively lead to the interleaving shown in

Initially `x = y = 0`

	Thread 1	Thread 2
1:		`atomic {`
2:		`t1 = x;`
3:	`x = 1;`	
4:	`atomic { z = 1;}`	
5:	`t2 = y;`	
7:		`y = 1;`
8:		`}`

Can `t1 == 0` and `t2 == 0`?

Figure 15. Disjoint lock atomicity != Single global lock atomicity

```
TxnStart(Descriptor* desc) {
  mynum = getTxnNextLinearizationNumber();
  startStampTable[desc->threadid] = mynum;
  ...
}

TxnCommit(desc){
  mynum = startStampTable[desc->threadid];
  startStampTable[desc->threadid] = MAXVAL;
  Quiesce(startStampTable, mynum);
  commitStampTable[desc->threadid] = mynum;

  if(validate(desc))
    ...
}
```

Figure 16. Total linearization: start and commit linearization are interlocked

Figure 15. This interleaving is allowed by DLA but disallowed by SGLA.

To enforce SGLA, we make a simple modification to tightly couple start and commit linearization, as shown in Figure 16. We refer to this as *total linearization*. In this case, we use a single linearization number and require that all transactions start, linearize, and complete in the same order regardless of whether they conflict. The arguments for race-free programs (sequential consistency) and racy programs (consistency with Java's memory model) are essentially the same as before, but we now have a total order over all transactions.

In contrast to an implementation that explicitly uses single global locks, our SGLA implementation allows concurrent execution of transactions in a staggered, pipelined fashion.

5.7 Enforcing Asymmetric Flow Ordering

For DLA and SGLA semantics, start linearization provides a conservative mechanism to enforce publication safety. From an implementation standpoint, read-only publishing actions force a conservative approach. In most scalable STM systems, read operations are *invisible*. When a transaction reads a value, it does not acquire a shared lock or otherwise communicate its read operation to other threads. Under DLA or SGLA, however, read operations (as in Figure 10) can serve as publishing actions and invalidate transactional accesses on another thread. Start linearization prevents ordering violations by forcing threads to wait.

Under asymmetric flow ordering semantics, however, a read operation cannot serve as a publishing action. As a result, an AFO implementation can rely upon a lighterweight mechanism to enforce publication safety. Figure 17 illustrates how we modified our DLA implementation to provide AFO semantics instead by performing

```
TxnStart(Descriptor* desc) {
  desc->startTimestamp = getTxnNextStartNumber();
  ...
}

TxnOpenForRead(Descriptor* desc, TxnRec * txnRec) {
  if (txnRec->isTimeStamp() &&
      txnRec > desc->startTimestamp) {
     txnAbort(desc);
  else {
     ...
  }
}

TxnOpenForWrite(Descriptor* desc, TxnRec * txnRec) {
  if (txnRec->isTimeStamp() &&
      txnRec > desc->startTimestamp) {
     txnAbort(desc);
  else {
     ...
  }
}

TxnCommit(Descriptor* desc){
  mynum = getTxnNextLinearizationNumber();
  commitStampTable[desc->threadid] = mynum;

  if(validate(desc)) {
    // commit: publish values to shared memory
    ...
    commitStampTable[desc->threadid] = MAXVAL;
    // release locks
    for (all txnRec in write set)
      txnRec = desc->startTimestamp;
    ...
    Quiesce(commitStampTable, mynum);
  } else {
    commitStampTable[desc->threadid] = MAXVAL;
    // abort : discard & release
    ...
  }
}
```

Figure 17. Lazy start linearization for AFO

lazy start linearization. First, to support lazy start linearization, we must directly record global timestamps on transactions records (as in TL2 [5]) instead of independent data-specific version numbers. On commit, a transaction records its unique start timestamp on all data it has written. Second, on a read or write operation, a transaction must check that the corresponding transaction record does not contain a timestamp for a transaction that started after it did. Under AFO, a publication violation can only occur if a transaction record contains a newer timestamp. Due to the check on read and write operations, no further quiescence is required on commit for start linearization. As privatization constraints are unchanged, quiescence for commit linearization is still performed.

6. Experiments

In this section, we evaluate our new weakly atomic implementation and compare it to our earlier implementations of weak and strong atomicity [1, 23] in a Java system. We performed our experiments on an IBM xSeries 445 machine running Windows 2003 Server

Enterprise Edition. This machine has 16 2.2GHz Intel® Xeon® processors and 16GB of shared memory arranged across 4 boards. Each processor has 8KB of L1 cache, 512KB of L2 cache, and 2MB of L3 cache, and each board has a 64MB L4 cache shared by its 4 processors. In all experiments, we use an object-level conflict detection granularity in our STM. We also do not use any offline whole program optimizations [23].

We evaluate eight different variants altogether:

- **Synch** represents the original lock-based version using Java synchronized with no transactions or STM barriers.
- **WeakEager** is our weakly atomic, in-place update STM.[1] It provides granular safety, but no other safety properties.
- **WeakLazy** is our baseline write buffering STM described in Section 5.1. In addition to granular safety, it also provides observable consistency and speculation safety.
- **WeakLazyCL** adds privatization safety to the above via commit linearization, as describe in Section 5.3.
- **WeakLazySGL** adds publication safety to the above via start linearization, as described in Section 5.4, and interlocks the two to provide single global lock semantics as discussed in Section 5.6.
- **WeakLazyDL** weakens the above to provide disjoint lock semantics, as described in Section 5.5.
- **WeakLazyAFO** weakens the above further to provide asymmetric flow ordering semantics by combining commit linearization with lazy start linearization, as described in Section 5.5.
- **StrongEager** is our strongly atomic, in-place update STM.[23] It provides all of the above safety properties as well as strong isolation between transactional and non-transactional code via non-transactional barriers.

Of these variants, only WeakLazySGL and StrongEager provide semantics at least as strong as a single global lock while enforce safety properties. Additionally, WeakLazyDL and WeakLazyAFO show the potential performance effects of weaker semantics that still provide some measure of safety. We compare these variants on the following workloads.

Figures 18,19 and 20 show the results of our experiments.

6.1 TSP

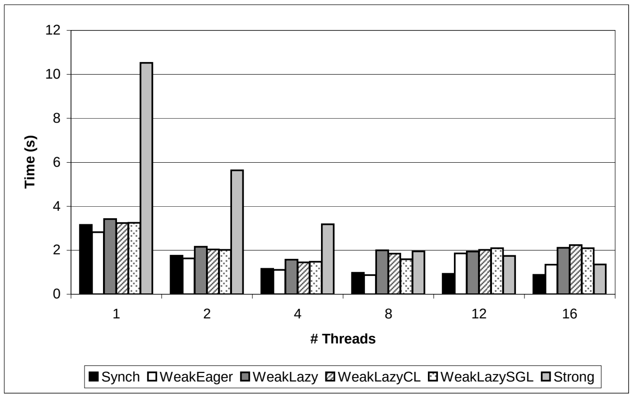

Figure 18. Tsp execution time over multiple threads

TSP is a travelling salesman problem solver. Threads perform their searches independently, but share partially completed work. The workload is fine-grained and already scales with locks. Interestingly, the workload contains a benign race where a shared variable representing the current minimum is monotonically decremented in a transaction, but read outside. As far as we can tell, our weak implementations execute correctly, even though our weakest implementation may cause threads to inadvertantly see this variable increase (due to speculation and rollback).

The overhead of weakly atomic implementations is low as relatively little time is spent inside of transactions. However, our WeakLazy implementation does not scale quite as well. The additional cost of commit linearization and start linearization is neglible. Our StrongEager implementation suffers from significant overhead on a single thread, but scales well and actually provides slightly better performance than WeakLazySGL at 16 processors.

6.2 java.util.HashMap

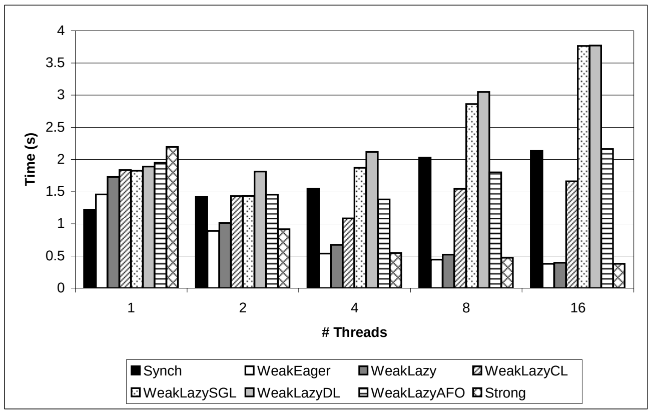

Figure 19. HashMap execution time over multiple threads

HashMap is a hashtable data structure from the Java class library. We test it using a driver execution 10,000,000 operations over 20,000 elements with a mix of 80% gets and 20% puts. The work is spread over the available processors, and little time is spent outside a transaction. The workload is coarse-grained; the synchronized version uses a single lock on the entire data structure and does not scale at all.

As most of the time is spent inside a transaction, the overhead of strong atomicity is minimal. The differences between our WeakEager, WeakLazy, and StrongEager are fairly small. All scale well to 16 processors. However, the cost of commit and start linearization is significant as the number of processors increases. At 16 processors, our WeakLazySGL implementation is significantly worse than StrongEager and even Synch. Although WeakLazyDL weakens the constraints of WeakLazySGL, it provides no performance benefit. In contrast, WeakLazyAFO is at least competitive with Synch and demonstrates the benefit of lazy start linearization. Nevertheless, it too does not scale well due to commit linearization.

6.3 java.util.TreeMap

TreeMap is a red-black tree from the Java class library. We use the same driver and parameters as above. In comparison to HashMap, transactions are larger as individual puts and get are $O(log(n))$ rather than $O(1)$. Figure 20 shows the results for this bechmark. Qualitatively, the results are similar. However, in this case, WeakLazyCL and WeakLazySGL scale well to 4 processors and degrade quickly afterward. As before, our WeakLazySGL implementation is significantly worse than StrongEager and Synch at 16 processors. In this case, both WeakLazyDL and WeakLazyAFO decay noticeably more slowly than WeakLazySGL. Both are faster than Synch at 16 processors.

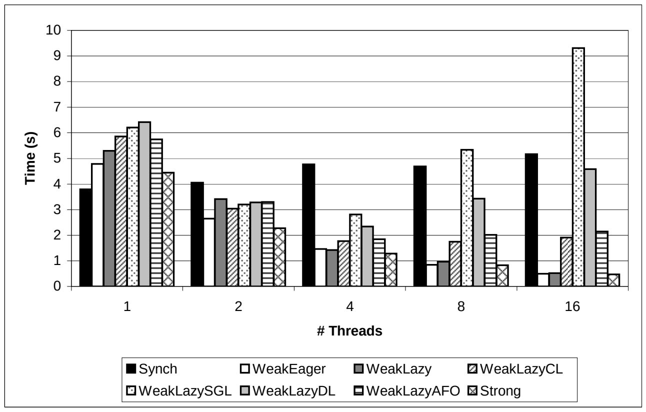

Figure 20. TreeMap execution time over multiple threads

6.4 Discussion

Our two safe implementations (WeakLazySGL and StrongEager) both impose significant costs to provide safety guarantees. However, the two variants show complementary strengths and weaknesses. StrongEager suffers from significant single thread overhead, but scales quite well with multiple processors. WeakLazySGL provides more reasonable single thread performance, but, when stressed with a constant stream of transactions, suffers from serious scalability issues due to commit and start linearization. While weakening the semantics (via WeakLazyDL or WeakLazyAFO) alleviates these concerns, it does not remove them.

7. Related Work

The concepts of transactional memory (TM) and software transactional memory (STM) were introduced by Herlihy and Moss [13] and Shavit and Touitou [22] respectively. As the research in STM matured, a new focus was placed on language integration. Harris and Fraser [8] present a thorough discussion of how transactions in their system should interact with existing Java language mechanisms, including native methods, existing synchronization constructs and the Java memory model. However, in their system they only provide consistency guarantees for programs that are correctly synchronized: all data accesses to shared locations must be either guarded by transactions or by mutexes, or locations must be marked as volatile. As far as we are aware, they did not address issues such as privatization or publication. Similarly, Welc et al. [28], in their attempt to reconcile differences between transactional semantics and the semantics of mutual-exclusion locks, restrict the set of programs their system would execute correctly to those that satisfy a set of safety criteria, including race-freedom. Other work [17] has investigated atomicity semantics from an architectural standpoint.

An alternate approach has been to integrate STM into a language whose type system can be leveraged to precisely control interactions between transactional and non-transactional code – an example of such integration in the context of Haskell has been presented by Harris et al. [9]. Moore and Grossman [19] demonstrate that such systems provides strong guarantees. Other alternative approaches include STM systems having segregated (into transactional and non-transactional) view of shared data [5] or implemented as language-level libraries [12] where issues such as treating transaction boundaries as memory fences do not appear at all.

The notions of weak and strong atomicity were introduced by Blundell et al. [3], who also demonstrated that standard software implementations of STM provided different semantics than locks. Shpeisman et al. [23] described the first scalable STM system supporting strong atomicity. In a similar vein, privatization [16] has been recognized as a serious problem in weakly atomic systems allowing interactions between transactional and non-transactional code – techniques close to our commit linearization have been proposed [25, 27, 23] as a solution to the privatization problem. As far as we know, there has been no similar work for publication.

Different STM implementation techniques have significant impact on what types of guarantees these systems provide with respect to interaction between transactional and non-transactional code. Buffered STMs [8, 5, 29, 28, 6, 11] provide speculation safety by limiting visibility of transactional writes until commit time. In-place update STMs [23, 10, 21] obtain lower performance overheads at the cost of weaker guarantees for non-transactional code.

8. Conclusions

In this paper, we have discussed the implications of lock-based semantics for software transactional memory. We have presented a systematic categorization of safety properties that STMs often violate. We have presented a weakly atomic Java STM that implements a single global lock semantics, and we have evaluated it against a strongly atomic Java STM.

We argue that, for unmanaged languages such as C++, an approach that enforces only observable consistency, privatization safety, and granular safety is sufficient. In particular, if an STM is observably consistent, we have shown that speculation safety is only an issue for incorrectly synchronized programs. However, programmers and compiler writers must be educated on the implications of STM and understand that even benign races are harmful. Moreover, ordering interactions between transactions and legacy synchronization (e.g., volatiles and locks) need to be better understood.

For Java and other managed languages that enforce strong guarantees for even incorrectly synchronized program, we demonstrate that a weakly atomic in-place update STM is insufficient. While we show that safety properties can be preserved in a weakly atomic write buffering STM, there are significant costs. Our results suggest that more research is needed to improve the performance of both strong and weak atomicity and on acceptable weaker semantics that provide the desired safety and security guarantees of managed languages while allowing aggressive STM implementations.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. S. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI 2006*.

[2] C. Blundell, E. C. Lewis, and M. Martin. Unrestricted transactional memory: Supporting i/o and system calls within transactions. Technical Report CIS-06-09, University of Pennsylvania, Department of Comp. and Info. Science, 2006.

[3] C. Blundell, E. C. Lewis, and M. M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2005.

[4] H. Boehm. A memory model for c++: Strawman proposal. In *C++ standards committee paper WG21/N1942*, February 2006. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1942.html.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC 2006*.

[6] R. Ennals. Software transactional memory should not be obstruction-free. http://www.cambridge.intel-research.net/ rennals/ notlockfree.pdf, 2005.

[7] D. Grossman, J. Manson, and W. Pugh. What do high-level memory models mean for transactions? In *MSPC 2006*.

[8] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA 2003*.

[9] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. In *PPoPP 2005*.

[10] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI 2006*.

[11] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC 2003*.

[12] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA 2006*.

[13] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA 1993*.

[14] R. Kennedy, S. Chan, S.-M. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. *ACM Transactions on Programming Languages and Systems*, 21(3):627–676, 1999.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 9(29):690–691, 1979.

[16] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool Publishers, 2006.

[17] J.-W. Maessen and Arvind. Store atomicity for transactional memory. *Electron. Notes Theor. Comput. Sci.*, 174(9):117–137, 2007.

[18] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL 2005*.

[19] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *The Second ACM SIGPLAN Workshop on Transactional Computing*, 2007.

[20] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open Nesting in Software Transactional Memory. In *PPoPP 2007*.

[21] B. Saha, A.-R. Adl-Tabatabai, R. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP 2006*.

[22] N. Shavit and D. Touitou. Software transactional memory. In *PODC 1995*.

[23] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and S. Bratin. Enforcing isolation and ordering in stm.

[24] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Brief announcement: Privatization techniques for software transactional memory. In *PODC 2007*.

[25] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization techniques for software transactional memory. Technical Report 915, University of Rochester, Computer Science Dept., 2007.

[26] H. Sutter. Prism - A Principle-Based Sequential Memory Model for Microsoft Native Code Platforms Draft Version 0.9.1. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2075.pdf, September 2006.

[27] 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 *CGO 2007*.

[28] A. Welc, A. L. Hosking, and S. Jagannathan. Transparently reconciling transactions with locking for java synchronization. In *ECOOP 2006*.

[29] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *ECOOP 2004*.