Single Global Lock Semantics in a Weakly Atomic 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 need for well-defined semantics. In contrast to database transactions, transaction memory (TM) semantics are complicated by the fact that programs may access the same memory locations both inside and outside transactions. *Strongly atomic* semantics, where non-transactional accesses are treated as implicit single-operation transactions, remain difficult to provide without specialized hardware support and/or significant performance overhead. As an alternative, many in the community have informally proposed that a *single global lock semantics* [16, 9], where transaction semantics are mapped to those of regions protected by a single global lock, provide an intuitive and efficiently implementable model for programmers.

In this paper, we explore the implementation and performance implications of single global lock semantics in a weakly atomic STM from the perspective of Java, and we discuss why even recent STM implementations fall short of these semantics. We describe a new weakly atomic Java STM implementation that provides single global lock semantics while permitting concurrent execution, but we show that this comes at a significant performance cost. We also propose and implement various alternative semantics that loosen single lock requirements while still providing strong guarantees. We compare our new implementations to previous ones, including a strongly atomic STM. [22]

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.

In spite of this, in one respect TM has complicated multithreaded 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 this area. Formal memory models have been developed that precisely define the set of allowable behaviors in the presence of threads. In Java, where threads are intrinsically part of the core language, the memory model is a fundamental part of the language specification and critical to its type 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. In particular, unlike database systems [8] where all data accesses are transactional, most TM implementations are obligated to handle situations when the same data is accessed both transactionally and non-transactionally. Because of this, nearly all published STM systems provide weaker guarantees for transactions than locks. Consider the publication example in Figure 1. Thread 2 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, this program will run correctly: `val` would either be `0` or `1` depending upon which critical section executed first. Nevertheless, most STMs would produce the value `42` in Thread 2 given the interleaving in Figure 1. As Thread 1 writes `data` outside a transaction, a weakly atomic STM would not detect any conflict and, thus, would not invalidate Thread 2's read operation.

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 1. Publication example with a seemingly benign race

It should be noted that, even under locks, this program has a data race: Thread 1 and Thread 2 may access `data` simultaneously. However, from the programmer's perspective, the value is never used in this case, and the race should be benign. Java's memory model [17] (assuming lock semantics) specifically disallows any execution that produces `42` in Thread 2. In fact, this example also has implications for correctly synchronized programs. Standard compiler reordering can inadvertently introduce a data race. Consider a correctly synchronized variant of this program in Figure 2 where Thread 2 only accesses `data` inside the conditional. Compiler optimizations such as speculative redundancy elimination or instruction scheduling may still introduce a data race by hoisting the access (e.g., to line 2 in Figure 1) if profitable. Such optimizations assume that introduced races are benign.

An appealing solution is to provide strong atomicity [5, 22] [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 restric-

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

```
Initially data = 42, ready = false, val = 0
Thread 1            | Thread 2
--------------------+------------------------
data = 1;           | atomic {
atomic {            |  if(ready)
 ready = true;      |   val = data;
}                   | }
            Can val == 42?
```

Figure 2. A correctly synchronized publication example

tive than locks and, thus, provide programmers with sufficiently strong guarantees. However, strong atomicity typically requires either specialized hardware support [21, 4, 10, 19] not available on existing systems, a sophisticated type system [12, 20, 1] that may not be easily integrated with languages such as Java or C++, or runtime barriers on non-transactional reads or writes [22] 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 specifically, there is a notion that transactions should behave as lock-based regions based upon a single global lock [16].

In this paper, we explore weakly atomic semantics from the perspective of Java. In particular, we investigate semantics that adhere to two basic principles that Java's memory model already provides for lock-based programs. First, for correctly synchronized programs, a transactional semantics may only admit sequentially consistent executions. Second, for incorrectly synchronized programs, a transactional semantics must still result in "reasonable" behavior to provide sufficient safety and security: values may not appear magically out of thin air and ordering rules must be respected. We make the following high-level contributions in this paper:

- We discuss the implications of single global lock semantics on STM behavior. In particular, we will show that, under such semantics, the Java memory model requires an STM to be privatization and publication safe and to prevent speculative effects from becoming visible to other threads.
- We describe a weakly atomic Java STM implementation that provides single global lock semantics for transactions. Our implementation permits optimistic concurrency, but it only allows executions that are consistent (under the Java memory model) with a semantics where all transactions execute as if under a single global lock.
- We describe and implement three weaker semantics: *disjoint lock atomicity*, *asymmetric lock atomicity*, and *encounter-time lock atomicity*. These semantics progressively weaken the restrictions of single global lock atomicity to allow greater concurrency. The first two obey the principles described above for correctly and incorrectly synchronized programs. The last does as well but restricts compiler and hardware reordering within transactions.
- We investigate the performance implications of these implementations by comparing them with previous unsafe weakly atomic and safe strongly atomic Java STMs. We discuss the various performance challenges that these semantics raise for future research.

```
      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`?

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 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. [3] 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. Researchers [20, 1] have shown that, under segregation, strong and weak atomicity are in fact semantically equivalent. Languages such as STM Haskell [12] 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, 22, 24] 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 [17] and in emerging models for C++ [6, 25]. 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 2 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 [6, 25], 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.

[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.

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.

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, 23] 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

	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.

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 [23, 24, 26].

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 1 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 1, 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 2 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:

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

- 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 1.
- 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 2 and introduce a data race as in Figure 1.
- The STM itself does not introduce speculative reads of data inside a transaction. For example, STM implementations that create shadow copies of an object or a cache line on write essentially introduce early reads of non-written fields. In Figure 2, 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. [22]

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 1), 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.

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.

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. [2, 13]. 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. [26]

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.

3.2.2 Granular Safety

Granular safety is a well-understood issue in STM implementations [22]. 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. [22]. 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 [22]. 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

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

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.

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 this section must be preserved by a Java STM. To date, the only scalable STM implementations that meet this criteria are strongly atomic [22]. Nevertheless, given the overhead of strong atomicity on non-transactional code, there remains a compelling argument for weakly atomic STM designs.

However, all published weakly atomic STM implementations suffer from one or more of the problems discussed here. Eager versioning STM systems can exhibit both out-of-thin-air (i.e., granularity and/or speculation) violations as well as ordering (i.e., privatization and/or publication) violations. Lazy versioning STM systems are somewhat safer: they avoid speculation problems by only allowing writes to become visible on commit, but they too allow ordering violations.

If C and C++ communities [6, 25] indeed converge to memory models that permit *catch fire* behavior on racy programs, then the corresponding restrictions on STM implementations for these languages may be relaxed. 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.

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. In the remainder of this paper, we will explore different weakly atomic STM semantics and implementations that provide safety guarantees even in the presence of data races.

4. Single Global Lock Atomicity

Short of strong atomicity, the simplest and most intuitive semantics is to consider transactions as if executed under a single global lock. [16, 9] 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. Third, combined with the strong guarantees of the Java memory model, it provides a reasonable behavior for programs with races. For example, it tolerates the benign race in Figure 1 and prevent the private value from leaking to another thread. Finally, it leverages years of research into the semantics of lock-based synchronization.

4.1 Implementing SGLA

To implement this model, we build upon an earlier software stack [2, 22] 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.

4.1.1 Write Buffering

Our weakly atomic implementation is a canonical write-buffering STM similar to those described by Harris and Fraser [11] and in TL2 [7]. 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 [7], 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, 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*.

```
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 9. Commit linearization

Unlike TL2, we do not use this timestamp to enforce consistent execution 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 is already locked.

Our basic write buffering implementation guarantees that values do not appear out of thin air. It maintains granular safety and speculation safety. As in Harris and Fraser [11], we enforce ordering on non-transactional volatile and lock operations by essentially treating them as single operation transactions. Below, we discuss how we augment our implementation to enforce ordering properties.

4.1.2 Commit Linearization for Privatization

In a write buffering STM, writes are performed after the linearization point. Unless we prevent it, these transactional writes can inadvertently race with non-transactional accesses on another thread leading to privatization safety violations. Privatization solutions for write buffering have been proposed in the literature [24]. Solutions that avoid non-transactional barriers essentially implement *commit linearization*, a simple form of quiescence [7, 26] that ensures that transactions complete in linear order.

In our implementation, shown in Figure 9, 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 transitively, from a transactional write (performed before

```
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 10. Start linearization

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 11. Publication via empty transaction

transaction end) and a non-transaction read (performed after the privatizing action). Because we treat volatiles and lock operations transactionally, our approach safely covers any ordering action.

4.1.3 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 1 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 also matches linearization and commit 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 1, 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 10 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).

We argue that the above mechanisms provide single global 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. The linearization number for each transaction defines a total ordering. If the linearization number of one transaction is smaller (greater) than the corresponding linearization number of another, then the first is ordered before (after) the second.

Note, an SGLA execution may be mapped to a semantically equivalent single lock execution via a series of transpositions of adjacent operations. [18] In contrast to an implementation that explicitly uses single global locks, our SGLA implementation allows concurrent execution of transactions in a staggered, pipelined fashion. This staggered execution provides an explicit total ordering over all transactions.

5. Disjoint Lock Atomicity

SGLA arguably imposes ordering in situations where a programmer might not actually expect one. For the incorrectly synchronized example in Figure 11, SGLA imposes an ordering between the empty transaction on the left and the transaction on the right even though they appear unrelated. It disallows the final result of `val == 42`.

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 [11] and Grossman, et al. [9]. [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 11. But, it provides equivalent guarantees for the racy programs in Figures 1 and 14.

5.1 Implementing DLA

To implement DLA, we make a slight modification to our SGLA implementation. In particular, we decouple start linearization from

[3] DLA is essentially the same as the weakest "conflicting regions" semantics defined by Grossman, et al. [9]

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

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

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

Figure 12. Decoupled start linearization for DLA

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 13. Disjoint lock atomicity != Single global lock atomicity

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 14. Publication via anti-dependence

commit linearization as shown in Figure 12. This decoupling allows independent transactions to start and commit in different orders.

For example, consider the interleaving in Figure 13. Thread 2's transaction begins first. When Thread 1's transaction can commit, it must wait for Thread 2 to reach its linearization point. However, at this point, Thread 1 may commit first (i.e., if it acquires an earlier linearization number in Figure 12). For Figure 13, this behavior is not permitted under SGLA but is allowable under DLA. Since the two transactions do not conflict, they are unordered by DLA.

We argue that this modified implemenation provides disjoint lock atomicity semantics for transactional Java programs. Note that 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 simultaneously) 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.

The arguments for race-free programs (sequential consistency) and racy programs (consistency with Java's memory model) are essentially the same as before. In this case, the implementation ensures that start, linearization, and commit points are already in the proper order for conflicting transactions. Thus any interleaved execution can be transformed to an equivalent non-interleaved execution that respects the partial order above.

6. Asymmetric Lock Atomicity

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 can serve as publishing actions and invalidate transactional accesses on another thread. Figure 14 illustrates this as Thread 1's read on `ready` invalidates Thread 2's earlier read on `data`. Start linearization prevents ordering violations by forcing threads to wait. The example in Figure 14, however, is rather contrived. Thread 1's transaction is read-only and, intuitively, it is odd to view it as a publishing action. Moreover, Thread 2 cannot glean any information from Thread 1 a priori. It is only after the fact that one can look at the global execution and determine the presence of an ordering.

We introduce an alternative semantics, *asymmetric lock atomicity* (ALA), that relaxes DLA further to not support such examples. Intuitively, the idea behind ALA is that information may only flow into a transaction via read operations. As in DLA, any transactional execution has a semantically equivalent lock-based one. In ALA, however, only locks protecting read accesses need to be acquired at the beginning of a transaction. Locks protecting write accesses may be acquired lazily at any point before the write operation. (From the compiler's perspective, lazy lock acquisition also has relaxed fence semantics: it does not artificially block compiler reordering of memory accesses within transactions.) In Figure 1, ALA still provides a strong ordering guarantee: Thread 2 reads `ready` and "acquires the lock" (in the equivalent lock execution) at the beginning on the transaction. As a result, the entire transaction must see the effect of Thread 1. On the other hand, in Figure 14, ALA provides a weaker ordering guarantee. Thread 1 is only ordered before the write in Thread 2's transaction. The early read of `data` may access the old value.

To formalize the intuition behind ALA, we need to relax the program order between memory accesses in a transaction to avoid artificial fence constraints induced by lazy lock acquisition. As in the standard Java model, there is a program order relationship between any non-transactional accesses in the same thread. But, transactional accesses are ordered only if execution of the second access is dependent on the information that could flow into the transaction through the first access. We define a *strict program order* relationship between memory accesses A and B executed by the same thread if either 1) A or B are non-transactional accesses, or 2) reordering A and B would violate single-thread semantics. Essentially, strict program order preserves the compiler's (or STM's)

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

TxnOpenForRead(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 15. Lazy start linearization for ALA

freedom to reorder operations inside a transaction. For conflicting transactions, there is a synchronization relationship between the end of the first transaction and either the beginning of the second (if the conflicting access in the second transaction is a read) or the access itself (if it is a write). The happens-before relationship is a transitive closure of the synchronization relationship and strict program order.

6.1 Implementing ALA

Under ALA a read operation cannot serve as a publishing action. As a result, an ALA implementation can rely upon a lighterweight mechanism to enforce publication safety. Figure 15 modifies the DLA implementation to provide ALA semantics instead by performing *lazy start linearization*. First, to support lazy start linearization, a transaction must directly record global timestamps on transactions records (as in TL2 [7]) 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 operation, a transaction must check that the corresponding transaction record does not contain a timestamp for a transaction that started after it did. Under ALA, a publication violation can only occur if a transaction record contains a newer timestamp. Due to the check on read operations, no further quiescence is required on commit for start linearization. As privatization constraints are unchanged, quiescence for commit linearization is still performed.

7. Encounter-time Lock Atomicity

Finally, we can weaken ALA even further to disallow all racy publication patterns. We define *encounter-time lock atomicity* (ELA) such that, in the equivalent lock-based execution, *all* locks may be acquired lazily at any point before the corresponding data is accessed regardless of whether the access is a read or write. ELA supports the privatization pattern, but it does not support racy publication in Figure 1. It would, however, support the non-racy variation in Figure 2 where the transaction in Thread 2 accesses `data` only in the conditional *provided that speculative code motion is disallowed.*

We argue that ELA preserves sequentially consistency for race-free programs under compiler and hardware reordering restrictions. It provides full ordering semantics for any privatization pattern. A program with a publication pattern can be race-free only if transactional memory accesses to non-transactional data are dependent on the conflicting access. Otherwise, reordering transactions would not change the set of accessed memory location and the program would exhibit a race.

7.1 Implementing ELA

ELA requires publication safety only for race-free programs. This is already provided by the basic write buffering implementation described in Section 4.1.1. We enforce privatization safety using commit linearization as described in Section 4.1.2. Note that an STM that used pessimistic readers implements ELA.

As alluded to above, ELA forces us to consider the interaction of the TM implementation with compiler and hardware reordering. As our ELA implementation supports ordering for non-racy publication but not for racy publication, speculative load hoisting is illegal. Thus, a system that allows speculative load reordering in the compiler or hardware [4] would still require a stronger implementation such as the ALA implementation described in Section 6.

8. Other Weak Models

Grossman et al. [9] propose two different semantics for transactions that are weaker than SGLA or DLA. In their first proposal, which they denote 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* ordering between two transactions that only conflict via an output or anti-dependence. That is, two transactions are ordered by a prewrite relationship if the first reads or writes a memory location written by the second. With prewrite semantics, visibility rules are modified such that the value a read may observe is constrained by both happens-before and prewrite ordering. As in ALA, neither semantics require an implementation to disallow the results shown in Figure 14. 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 and do not support the classical privatization examples in the literature [16, 22, 24]. In particular, the synchronizing action (as shown in Figure 4) is typically a transactional write operation. In both of the weaker semantics proposed by Grossman et al., it is not ordered with earlier transactions that only read the corresponding memory location [18]. Because of this, these semantics are not strong enough to preserve correctness for idioms that are race-free under locks.

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

[4] IA32 does not allow such reordering.[14]

```
Initially data = 42, ready = false
  Thread 1           | Thread 2
---------------------+--------------------
                     | atomic {
                     |  data = 1;
  val = data;        |
  atomic {           |
   test = ready;     |
  }                  |
                     |  ready = true;
                     | }
Can test == false and val == 1?
```

Figure 16. Variant of publication via anti-dependence

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). Formally, we can define AFO in terms of the happens-before ($\rightarrow^{hb}$) and prewrite ($\rightarrow^{pw}$) relations defined by Grossman et al. [9] for their prewrite ordering semantics discussed above with one additional change: we strengthen the happens-before relation such that $a \rightarrow^{pw} b \rightarrow^{hb} c \implies a \rightarrow^{hb} c$ if b and c are not part of the same outer-level transaction. This ensures that memory accesses subsequent to the conflicting transaction are handled consistently (avoiding the privatization problem). The prewrite visibility rule remains as is: a read r may observe a write w unless $r(\rightarrow^{hb} \cup \rightarrow^{pw})^{+}w$ or there exists a w' such that $w(\rightarrow^{hb} \cup \rightarrow^{pw})^{+}w' \rightarrow^{hb} r$.

Informally, these semantics are similar but slightly stronger than ALA as described in Section 6. As in ALA, they do not allow a read-only transaction to act as a publishing action as suggested by Figure 14, but do support the publication idiom in Figure 1 as well as all privatization idioms. However, they are slightly more restrictive as illustrated by another anti-dependence example in Figure 16. In this case, the final result (`test == false` and `val == 1`) is allowable under ALA but forbidden under AFO.

While AFO is more esoteric than ALA, it provides somewhat stronger semantics at arguably little implementation cost: the ALA implementation described in Section 6.1 is strong enough to enforce AFO semantics.

9. Comparing Models

All the models described in Sections 4- 7 preserve sequential consistency for race-free programs and prohibit values out-of-thin air. The models differ in the ordering guarantees that they provide for transactional and non-transactional memory accesses in racy programs. Table 1 summarizes these guarantees using a generic transactional program template in Figure 17. For completeness, Table 1 also includes the information for strong atomicity, weak atomicity, and two models discussed in Section 8 (AFO and *write* $\rightarrow^{hb}$ *read with pre-write ordering* (Pre-Write) [9]).

In Figure 17, transaction T1 serializes before transaction T2. Strong atomicity and SGLA enforce a total ordering between T1, T2 and surrounding non-transactional accesses. DLA enforces this ordering only if T1 and T2 contain conflicting memory accesses. ALA and ELA relax the ordering for publication pattern. ALA enforces publication ordering only if the information can flow from T1 to T2 (that is, if conflicting access in T2 is a read). ELA further

```
  Thread 1           | Thread 2
---------------------+--------------------
  S1;                |
  atomic { // T1     |
   ...               |
   S2;               |
   ...               |
  }                  |
                     | atomic { // T2
                     |  ...
                     |  S3;
                     |  ...
                     | }
                     | S4;
```

Figure 17. A generic template for ordering in a transactional program. Transactions `T1` and `T2` are ordered. The ordering guarantees between `S1` and `S3`, and `S2` and `S4` depend on the memory model as shown in Table 1.

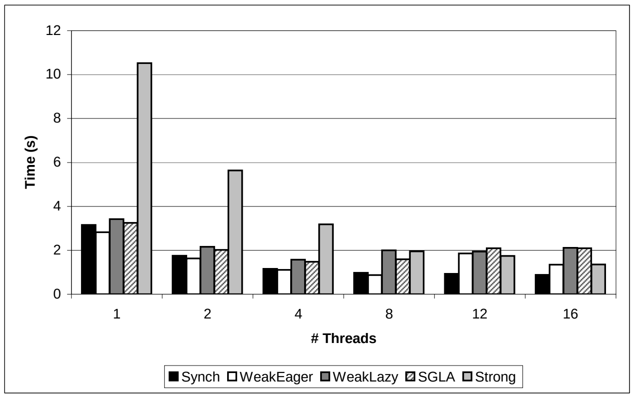

Figure 18. Tsp execution time over multiple threads

weakens the rules by providing ordering only for accesses that are dependent on the conflicting read.

AFO and Pre-Write relax DLA ordering in a different way; they provide no ordering for transactional reads in a publication pattern when the conflicting access is a write. In addition, Pre-Write does not guarantee privatization safety, and, thus, does not provide sequential consistency for race-free programs. Unlike the models presented in this paper, AFO and Pre-Write do not allow for a lock-based formulation, as they specify different ordering rules for transactional reads and writes. Instead, they are formulated using a somewhat unintuitive modification of Java Memory Model visibility rules.

Finally, weak atomicity provides no guarantees on ordering between transactional and non-transactional memory accesses.

10. Experiments

In this section, we evaluate our new weakly atomic implementations and compare it to our earlier implementations of weak and strong atomicity [2, 22] 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 [22].

	Publication: S1 happens before S3?			Privatization: S2 happens before S4?	
Memory	T1 and T2	T1 and T2 conflict on C in T2		T1 and T2	T1 and T2
Model	do not conflict	C is read	C is write	do not conflict	conflict
Strong	Yes	Yes	Yes	Yes	Yes
SGLA	Yes	Yes	Yes	Yes	Yes
DLA	No	Yes	Yes	No	Yes
ALA	No	Yes	No	No	Yes
ELA	No	if C $\rightarrow^{spo}$ S3	No	No	Yes
AFO	No	Yes	if S3 is a write	No	Yes
Pre-Write	No	Yes	if S3 is a write	No	No
Weak	No	No	No	No	No

Table 1. Summary of memory models based upon Figure 17 template.

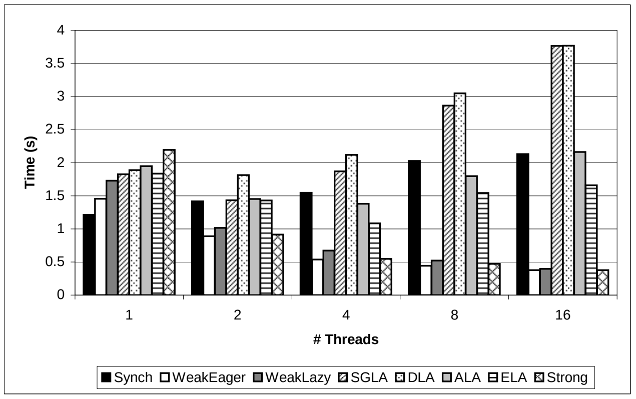

Figure 19. HashMap execution time over multiple threads

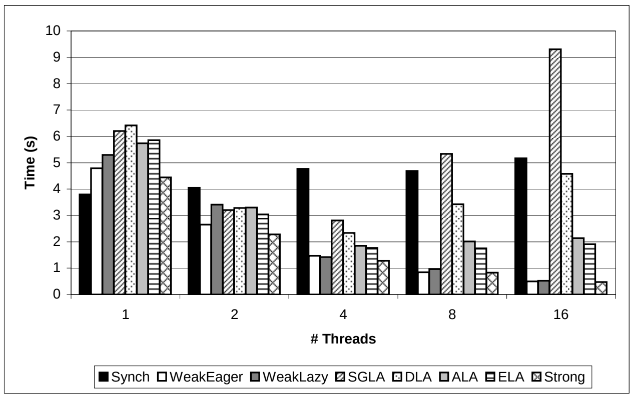

Figure 20. TreeMap execution time over multiple threads

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.[2] It provides granular safety, but no other safety properties.
- **WeakLazy** is our baseline write buffering STM described in Section 4.1.1. In addition to granular safety, it also provides consistent execution and speculation safety.
- **SGLA** adds publication and privatization safety to the above via start and commit linearization as discussed in Section 4.1.
- **DLA** weakens the above to provide disjoint lock semantics, as described in Section 5.1.
- **ALA** weakens the above further to provide asymmetric lock semantics by combining commit linearization with lazy start linearization, as described in Section 6.1.
- **ELA** weakens the above yet further to provide encounter-time lock semantics by removing any lazy start linearization, as described in Section 7.1.
- **Strong** is our strongly atomic, in-place update STM.[22] 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 SGLA and Strong provide semantics at least as strong as a single global lock for all programs. Additionally, DLA, ALA, and ELA show the potential performance effects of weaker semantics that still provide varying degrees of "reasonable" behavior. We compare these variants on the following workloads.

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

10.1 TSP

TSP is a traveling 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 inadvertently 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 negligible. Our Strong implementation suffers from significant overhead on a single thread, but scales well and actually provides slightly better performance than SGLA at 16 processors.

10.2 java.util.HashMap

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 Strong 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 SGLA implementation is significantly worse than Strong and even Synch. Although DLA weakens the constraints of SGLA, it provides no performance benefit. In contrast, ALA is at least competitive with Synch and demonstrates the benefit of lazy start linearization. Nevertheless, it and ELA do not scale well due to commit linearization.

10.3 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, SGLA scales well to 4 processors and degrades quickly afterward. As before, our SGLA implementation is significantly worse than Strong and Synch at 16 processors. In this case, both DLA, ALA, and WeakLazyCL decay noticeably more slowly than SGLA. All three are faster than Synch at 16 processors.

10.4 Discussion

Our two safest implementations (SGLA and Strong) both impose significant costs to provide safety guarantees. However, the two variants show complementary strengths and weaknesses. Strong suffers from significant single thread overhead, but scales quite well with multiple processors. Our SGLA implementation 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 DLA, ALA, or ELA) can alleviate these concerns, it does not remove them.

11. Conclusions

In this paper, we have discussed the implications of weakly atomic semantics on STM from the perspective of the Java memory model, and we have shown that existing weakly atomic STM implementations fall short of standard Java principles. We have explored four different weakly atomic semantics that (a) provide sequential consistency for correctly synchronized programs and (b) preserve ordering and prevent out-of-thin air values for all programs. However, we have also shown that these semantics introduce challenges to STM design and scalability. Our results suggest that more research is needed to improve the performance of both strong and weak atomicity to provide acceptable STM semantics.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL 2008*.

[2] 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*.

[3] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[4] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA 2005*.

[5] 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.

[6] 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.

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

[8] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

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

[10] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA 2004*.

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

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

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

[14] Intel Corporation. *Intel 64 Architecture Memory Ordering White Paper*. http://www.intel.com/products/processor/manuals/318147.pdf.

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

[18] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Towards a lock-based semantics for Java STM. Technical Report UW-CSE-07-11-01, November 2007.

[19] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *HPCA 2006*.

[20] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL 2008*.

[21] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA 2005*.

[22] 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. In *PLDI 2007*.

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

[24] 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.

[25] 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.

[26] 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*.