Enforcing Isolation and Ordering in STM

Tatiana Shpeisman[1] Vijay Menon[1] Ali-Reza Adl-Tabatabai[1] Steven Balensiefer[2]
Dan Grossman[2] Richard L. Hudson[1] Katherine F. Moore[2] Bratin Saha[1]

[1]Programming Systems Lab
Intel Corporation
Santa Clara, CA 95054

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

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

Abstract

Transactional memory provides a new concurrency control mechanism that avoids many of the pitfalls of lock-based synchronization. High-performance software transactional memory (STM) implementations thus far provide *weak atomicity*: Accessing shared data both inside and outside a transaction can result in unexpected, implementation-dependent behavior. To guarantee isolation and consistent ordering in such a system, programmers are expected to enclose all shared-memory accesses inside transactions.

A system that provides *strong atomicity* guarantees isolation even in the presence of threads that access shared data outside transactions. A strongly-atomic system also orders transactions with conflicting non-transactional memory operations in a consistent manner.

In this paper, we discuss some surprising pitfalls of weak atomicity, and we present an STM system that avoids these problems via strong atomicity. We demonstrate how to implement non-transactional data accesses via efficient read and write barriers, and we present compiler optimizations that further reduce the overheads of these barriers. We introduce a *dynamic escape analysis* that differentiates private and public data at runtime to make barriers cheaper and a *static not-accessed-in-transaction* analysis that removes many barriers completely. Our results on a set of Java programs show that strong atomicity can be implemented efficiently in a high-performance STM system.

Categories and Subject Descriptors D.1.3 [*Programming techniques*]: Concurrent Programming—Parallel Programming; D.3.3 [*Programming languages*]: Language Constructs and Features—Concurrent programming structures; D.3.4 [*Programming Languages*]: Processors—Code generation, Compilers, Optimization, Run-time environments

General Terms Algorithms, Measurement, Performance, Design, Experimentation, Languages

Keywords Transactional Memory, Strong Atomicity, Weak Atomicity, Isolation, Ordering, Escape Analysis, Compiler Optimizations, Code Generation, Virtual Machines

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLDI'07 June 11–13, 2007, San Diego, California, USA.
Copyright © 2007 ACM 978-1-59593-633-2/07/0006…$5.00

Globally visible `java.util.LinkedList list`
Initially `list == [Item{val1==0,val2==0}]`

Thread 1	Thread 2

`Item item;`
`synchronized(list) {`
` item = (Item)`
`  list.removeFirst();`
`}`
`int r1 = item.val1;`
`int r2 = item.val2;`
 	
Can `r1!=r2`?
	`synchronized(list) {`
	` if(!list.isEmpty()) {`
	`  Item item = (Item)`
	`   list.getFirst();`
	`  item.val1++;`
	`  item.val2++;`
	` }`
	`}`

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

1. Introduction

Transactional memory (TM) offers a simple concurrency control mechanism that avoids many of the pitfalls associated with locks. With TM, the programmer declares an atomic code block, and the underlying system guarantees atomicity and isolation during execution, giving the illusion that the block executes as an atomic step with respect to other concurrently executing operations.

Prior software TM (STM) systems mostly implement *weak atomicity* [9], in which non-transactional memory accesses go directly to memory and bypass the STM access protocols. Weak atomicity allows violation of a transaction's isolation if there is a data race between transactional and non-transactional code; furthermore, weak atomicity allows violation of established memory ordering rules if a thread outside of a transaction accesses a location modified by a committed transaction. To ensure isolation and consistent ordering, it is sufficient to manually *segregate* shared-memory into memory accessed only in transactions and memory accessed only outside transactions. As Section 2 explains in more detail, without this segregation, different weak-atomicity STM implementation techniques will exhibit different and surprising behaviors. Strict segregation is error-prone especially as software evolves; but more disturbingly, weak atomicity can require segregation *even in situations where lock-based critical sections do not*. In such situations, weak atomicity is a step backwards from the goal of reliable and efficient concurrent programming.

Consider the lock-based Java program in Figure 1, adapted from Larus and Rajwar [35] and from Hudson et. al. [32]. Thread 1 removes an item from `list` and dereferences it twice. Because the item becomes private to Thread 1 once it is removed, Thread

1 can dereference the item outside the critical section. Thread 2, which properly synchronizes on the list, clearly cannot touch the item once Thread 1 removes it. In this lock-based case, it is clear that `r1 == r2` as either both fields were incremented or neither were. The Java Memory Model [38] was designed to support such idioms; this program is correctly synchronized.

Now consider Figure 1 with the locks replaced by weak atomic blocks. There is no segregation because fields of `item` are accessed inside and outside transactions. Most of the existing multiprocessor STM implementations could violate isolation or ordering for reasons depending on the implementation approach. On systems implementing eager versioning and lazy conflict detection [27, 1], Thread 2 may speculatively update fields of `item` even as Thread 1 removes it and commits its transaction. Although Thread 2 will eventually abort, the unprotected dereference operations in Thread 1 may see speculative values before this. On systems implementing lazy versioning [25, 28, 39, 21], Thread 1 may commit its transaction after Thread 2 commits its transaction but before Thread 2 updates the fields of `item`. Although Thread 2 will eventually update these fields from its private buffers, Thread 1's unprotected dereference will see stale values before this.

We believe the simplest way to avoid the morass of unpredictable and implementation-defined behavior for such seemingly reasonable programming idioms is to provide *strong atomicity* [9], in which the system provides isolation and consistent ordering without requiring segregation. Prior implementations of strong atomicity, however, use hardware support [30, 42, 43, 24], assume a uniprocessor [47, 37], enforce strict segregation statically [26], or do not demonstrate scalable parallelism [31, 6].

This paper presents a high-performance strong-atomicity STM system. We make the following contributions:

1. We present the first scalable STM designed for multiprocessors that supports strong atomicity in an imperative language.
2. We characterize the kinds of problems that occur in weakly-atomic systems. Although past work has illustrated some of the problems that occur with weak atomicity, none have provided the same detail or as complete an analysis of the problems that different STM implementations exhibit. (Section 2)
3. We demonstrate how to implement strong atomicity via efficient read and write barrier sequences for memory accesses outside transactions. (Section 3)
4. We present new optimizations to reduce the cost of these read and write barriers (and often remove them entirely). First, *dynamic escape analysis* (Section 4) tracks thread-local objects at runtime to avoid unnecessary synchronization in the barriers. Second, a whole-program *static not-accessed-in-transaction analysis* (Section 5) eliminates barriers for code that cannot conflict with transactional memory accesses. Third, intraprocedural optimizations (Section 6) such as *barrier aggregation* help amortize the cost of barriers.
5. We measure the performance and scalability of strong atomicity on both non-transactional benchmarks and a set of multithreaded Java programs modified to use our atomic construct. We show that strong atomicity has no negative effect on scalability and that our optimizations reduce the overhead of strong atomicity to a fraction of the overhead imposed by an unoptimized implementation. (Section 7)

2. Characterizing weak atomicity behaviors

In a safe language such as Java, language constructs need semantics that precisely determine possible behaviors, but giving precise, formal semantics for transactions is beyond our present scope. Rather, we aim to describe as thoroughly as we can the various ways that existing STM systems for weak atomicity can violate isolation or ordering expectations, categorizing the issues with terminology from the database community [22] or prior memory-consistency work [2] where appropriate. Because strong atomicity has none of the pitfalls we present, we believe it is a better starting point for developing a comprehensive memory model that includes TM even though memory-model questions remain open [23].

As expected, all unexpected behaviors involve transactional and non-transactional code accessing the same shared data with at least one write access. Section 2.1 presents well-known isolation violations that also occur with locks. However, there are additional problems specific to weakly-atomic STMs. Section 2.2 presents isolation anomalies that can result from the speculate-and-abort strategy of eager-versioning STM systems [1, 27], in which a transaction updates shared memory directly and rolls back its writes if it aborts. Section 2.3 presents ordering anomalies that can result from lazy-versioning STM systems [25, 28, 39, 21], in which a transaction computes with private versions of written-to data and updates shared memory after it commits. Section 2.4 presents anomalies due to the coarse granularity at which some lazy- and eager-versioning STM systems manage multiple versions of data.

Some of the examples in this section illustrate programs that are properly synchronized using critical sections in lieu of atomic blocks, while others illustrate programs that have data races. We believe not only that the properly synchronized programs should execute correctly using atomic blocks, but also that in the spirit of the Java memory model, we should provide some guarantees for programs with data races and prohibit "out-of-thin-air" values that may compromise safety and security [38].

Note that we use the term *weak atomicity* in a more general fashion than Blundell, et.al., [9] have defined it. In contrast to their work, we do not assume a particular behavior when transactional and non-transactional code access the same data. Instead, we refer to any STM system as weakly atomic if it allows non-transactional accesses to access memory directly, bypassing the STM system's mechanisms for accessing shared memory. As shown below, different types of weakly-atomic STM systems behave very differently. [1]

2.1 Problems shared with locks

Under weak atomicity, one typically expects that data races cause isolation violations just like with improperly synchronized lock-based code. There are three issues, all demonstrated in Figure 2.

Non-repeatable reads: Figure 2(a) illustrates a *non-repeatable read* (NR). Thread 1's two reads should observe the same value for `x` since they are inside a transaction. But if Thread 2 writes `x` between the two reads then Thread 1 will observe two different values for the variable, violating transaction isolation. A similar problem happens if Thread 1 first writes to `x` (say the value `10`) and then reads `x`; Thread 1 will not observe the value it wrote (`10`) if Thread 2 writes `x` between Thread 1's write and read.

Intermediate lost updates: Figure 2(b) illustrates an *intermediate lost update* (ILU) where the same memory location is updated both inside and outside a transaction. In a serialized execution of the two threads, both updates should compose. The shared variable `x` should have the final value of either `10` (Thread 2 executed last) or `11` (Thread 2 executed first); but `x` will have the final value `1` (as if Thread 2's write never happened) if Thread 2 updates `x` between Thread 1's read and write operations.

[1] Blundell, et.al. [9], appear to assume that weak atomicity implies an eager-versioning STM where transactional updates become immediately visible to non-transactional code. In a lazy-versioning STM, such updates are visible only after commit.

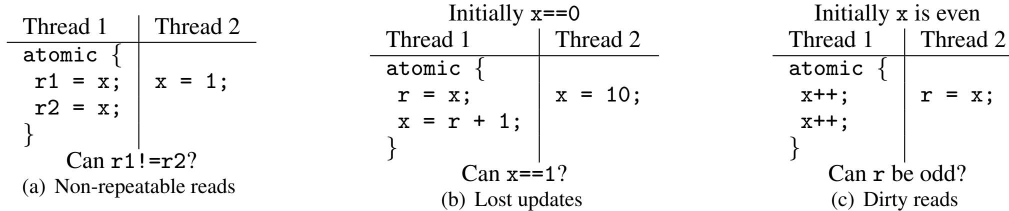

Figure 2. Isolation violations expected with data races.

Intermediate dirty reads: Figure 2(c) illustrates an *intermediate dirty read* (IDR) where a non-transactional access can observe the intermediate state of a transaction. Thread 1 maintains the invariant that `x` is even, but Thread 2 will observe an odd value if it reads `x` between Thread 1's two increments. Under lazy versioning, intermediate dirty reads cannot occur, but at the cost of ordering violations discussed in Section 2.3.

2.2 Eager-versioning anomalies

Eager-versioning STM can exhibit dirty read and lost update behaviors that are not otherwise possible in lock-based code. These behaviors are due to the speculate-and-undo strategy of eager versioning, in which a transaction speculatively updates shared memory in place and then on abort, rolls back these updates with a compensating write. A rolled-back transaction thus manufactures new shared memory writes that are not present in any sequentially-consistent execution, resulting in new lost update and dirty read scenarios.

Speculative lost updates: Figure 3(a) illustrates a *speculative lost update* (SLU) where a non-transactional update is lost due to a write during transaction rollback. Assume Thread 1 updates `x` first, and then Thread 2 updates `y` and `x`. If Thread 1 now rolls back, it will restore `x`'s value back to `0` and skip over the update to `x` on re-execution (because it now observes `y==1`), resulting in `x==0`.

Speculative dirty reads: Figure 3(b) illustrates a *speculative dirty read* (SDR) where a non-transactional read observes the speculative state of a transaction. Assume Thread 1 updates `x` first, and then Thread 2 updates `y` after observing `x==1`. If Thread 1 now rolls back, it will restore `x`'s value back to zero and skip over the update to `x` on re-execution, resulting in `x==0`.

2.3 Lazy-versioning anomalies

Lazy-versioning STM can exhibit memory ordering problems similar to memory consistency problems in shared-memory multiprocessors [2]. Lazy-versioning STM buffers transactional updates privately and then writes the buffered updates back to shared memory "lazily" when the transaction commits. The window of time between the transaction commit and the update to shared memory can cause memory ordering violations because non-transactional code does not see all committed values during that time.

Memory inconsistency: Figure 4 illustrates *memory inconsistency* (MI) due to violation of established ordering rules. In Figure 4(a), Thread 1 initializes a field in the object `el` and then publishes the object by writing it to a volatile shared variable `x`. Thread 2 may now see the published object in `x` but not see the initialized value of its field because a lazy-versioning STM copies buffered values to memory one at a time in no particular order. Since `x` is volatile, this ordering is inconsistent [38]. The same problem can occur when a final field is initialized inside a transaction but is reordered with a publishing write. This is similar to the multiple *overlapped writes* problem described in [2].

Figure 4(b) shows another memory inconsistency example distilled from Figure 1. Thread 1 takes a shared value in `x` and makes it thread local. Once `x` is set to `null`, the object in `r1` is not visible to other threads, and from the programmers point of view, it should be safe to access `x` outside an atomic region. In a lazy-versioning STM, Thread 2 may buffer an update to `x.val`, validate itself, and commit. But before it has flushed the new value to memory, Thread 1 may execute its transaction and start accessing `r1.val`. Logically, Thread 2's transaction executes before Thread 1's transaction, and Thread 1's accesses to `r1.val` execute after Thread 1's transaction. But because the STM updates shared memory lazily, Thread 1's accesses to `r1.val` end up racing with the STM's update. This is similar to the *buffered writes* problem described in [2].

2.4 Anomalies due to coarse-grained versioning

When the granularity at which the STM system manages data versions is greater than the granularity at which non-transactional code writes data (e.g., if the STM logs or buffers writes in 8-byte blocks while a non-transactional access writes a 4-byte value within that block), then additional problems can occur in both lazy- and eager-versioning STM systems.

Granular lost updates: Figure 5(a) illustrates a *granular lost update* (GLU) where the non-transactional update to `x.g` is lost *even though the transaction never accesses this field and there is no data race*. Eager-versioning STM systems [27, 1] maintain undo log entries that may be larger than individual object fields (or array elements). If Thread 1's transaction creates an undo log entry that spans fields `f` and `g` of `x`, Thread 2's update to `x.g` could be lost if Thread 1 aborts and rolls back `x.f`. A similar problem can happen in lazy-versioning STM's that buffer values at a similar granularity; for example, if Thread 2 updates `x.g` after Thread 1 has created a private copy that spans fields `f` and `g`, then the update will vanish after Thread 1 commits and writes back its copy to shared memory.

Granular lost updates arise because the STM manufactures new writes to variables that lie adjacent to a variable updated inside a transaction. These writes do not exist in any sequentially-consistent execution of the program. Granular lost updates are similar to the problem of rewriting adjacent data described by Boehm [10].

Granular inconsistent reads: Figure 5(b) illustrates a *granular inconsistent read* (GIR) where a transaction may see inconsistent updates from a non-transactional thread. Granular inconsistent reads are similar to granular lost updates but may only occur in lazy versioning STMs. Here, the shared variable `y` is volatile and imposes certain ordering constraints between Thread 1 and Thread 2. In particular, if Thread 1 observes Thread 2's update to `y`, it must also observe Thread 2's update to `x.g`. In a lazy-versioning STM, however, Thread 1's transaction (as in the earlier GLU example) may have created a private copy on the write to field `x.f` that also spans `x.g`. In this case, the transaction will later read its own stale copy of `x.g` and not observe Thread 2's update as required by the Java memory model. Note that a granular inconsistent read is a memory inconsistency anomaly akin to those described in Section 2.3.

```
Initially x==0 and y==0
Thread 1          | Thread 2
atomic {          |
  if (y==0)       | x = 2;
    x = 1;        | y = 1;
  /*abort*/       |
}                 |
Can x==0?
```

(a) Speculative lost updates

```
Initially x==0 and y==0
Thread 1          | Thread 2
atomic {          |
  if (y==0)       | if (x==1)
    x = 1;        |   y = 1;
  /*abort*/       |
}                 |
Can x==0?
```

(b) Speculative dirty reads

Figure 3. More isolation violations for eager versioning STM.

```
Suppose x is volatile
Initially x==null
and el.val==0
Thread 1          | Thread 2
atomic {          | r=-1;
  el.val=1;       | if(x!=null)
  x=el;           |   r=x.val;
}                 |
Can r==0?
```

(a) Overlapped writes

```
Initially x!=null
and x.val==1
Thread 1          | Thread 2
atomic {          | atomic {
  r1=x;           |   if(x!=null)
  x=null;         |     x.val++;
}                 | }
r2=r1.val;        |
r3=r1.val;        |
r1.val=0;         |
Can r2!=r3 or r1.val!=0?
```

(b) Buffered writes

Figure 4. Lazy-versioning ordering violations.

```
Initially x.g==0
Thread 1          | Thread 2
atomic {          |
  x.f=1;          | x.g=1;
}                 |
Can x.g==0?
```

(a) Granular lost updates

```
Suppose y is volatile
Initially x.g==0 and y==0
Thread 1          | Thread 2
r=-1;             |
atomic {          |
  x.f=...;        | x.g=1;
  if (y==1)       | y=1;
    r=x.g;        |
}                 |
Can r==0?
```

(b) Granular inconsistent reads

Figure 5. Anomalies due to coarse-grained versioning.

Non-Txn	Txn	Anomaly	Versioning Eager	Versioning Lazy	Locks	Strong
write	read	NR	yes	yes	yes	no
		GIR	no	yes	no	no
write	write	ILU	yes	yes	yes	no
		SLU	yes	no	no	no
		GLU	yes	yes	no	no
		MI	no	yes	no	no
read	write	IDR	yes	no	yes	no
		SDR	yes	no	no	no
		MI	no	yes	no	no

Figure 6. Summary of weak atomicity behaviors.

Because of granular lost updates and inconsistent reads, the programmer must consider versioning granularity when segregating data in a weakly-atomic system, and the weak-atomicity programming interface must explicitly define this granularity; otherwise, the STM must manage versions at the granularity of the individual fields updated inside a transaction. A strongly-atomic system hides this granularity, but optimizations such as our not-accessed-in-transaction analysis must take the granularity into account when analyzing which variables are updated inside transactions.

2.5 Discussion

Figure 6 summarizes the behavior of weak atomicity, comparing it with locks. This table shows the behaviors for read-write, write-write, and write-read accesses between transactional (Txn) and non-transactional (Non-Txn) code. The eager and lazy versioning columns represent weak atomicity for these version management policies, the Locks columns represents lock-based critical sections. The Strong column represents strong atomicity, emphasizing that the techniques that we present later avoid these behaviors.

Revisiting the privatization example in Figure 1, the problem with eager versioning is an SDR (Thread 2 may increment and then decrement `item.val++` due to an abort) and the problem with lazy versioning is an MI since the non-transactional accesses "see" that the increments happen after the transaction commits.

3. Enforcing isolation and ordering

Enforcing memory ordering and isolation between transactional and non-transactional threads requires *read and write isolation barriers* in code that executes outside of atomic blocks. Avoiding dirty reads requires read barriers that detect simultaneous writes by a transaction, avoiding non-repeatable reads and lost updates requires write barriers that prevent a simultaneous access by a transaction, and avoiding memory inconsistencies requires barriers that detect pending buffered updates by a transaction.

We have implemented our techniques in a high-performance STM system that extends Java with an `atomic{ B }` construct for declaring an atomic code block `B` [1]. Our system supports a full range of transactional features including closed and open nesting [45] and user-initiated retry operations. The JIT compiler automatically inserts and optimizes STM operations for code that executes inside a transaction and isolation barriers for non-transactional code. At the core of our system lies McRT-STM [49], which implements optimistic concurrency control using versioning [34] for reads and strict two-phase locking [22] and eager versioning for writes. For our whole-program static analysis we used the Paddle [7] extension to Soot [56].

3.1 Transaction records

In the base STM system, a pointer-sized *transaction record* [1] tracks the state of each object accessed inside a transaction. The transaction record can be in either the *shared* state, which allows read-only access by any number of transactions, or the *exclusive* state, which allows read-write access by the single transaction that owns the record. In the shared state, the record contains a version number used for optimistic read concurrency. In the exclusive state, it contains a pointer to the owning transaction's *descriptor*. Each object has a *transaction field* holding its transaction record.

To support efficient strong atomicity, we extend the transaction record to four states encoded in its three least-significant bits (Figure 7). The shared and exclusive states are as before. The *exclusive anonymous* state indicates that some thread owns the object exclusively for read-write access, but the record does not indicate who owns it. This state prevents a transaction from accessing data that a non-transactional thread is concurrently updating. The upper bits in this state contain the version number from the record's prior shared state. An object whose transaction field is in the *private* state is visible only to a single thread. Threads never contend for private objects so the runtime can avoid most of the barrier overheads on accesses to private objects.

Encoding	State	Value in upper bits
x..x011	Shared	Version number
x..xx00	Exclusive	Owner address
x..x010	Exclusive anonymous	Version number
1..1111	Private	All ones

Figure 7. Transaction record encoding.

This encoding enables efficient read and write barriers outside transactions. A non-transactional read can check whether it conflicts with a transaction — that is, detect dirty reads in an eager-versioning STM or pending updates by a committed transaction in a lazy-versioning STM — by inspecting only the second lowest bit.[2] A non-transactional write can acquire exclusive anonymous ownership of a record by atomically flipping the lowest bit from one to zero with a single IA32 bit-test-and-reset (BTR) instruction — thus avoiding non-repeatable reads and lost updates — and can release ownership and at the same time increment the version number by incrementing the record by 9. Figure 8 shows the state transition diagram for the transaction record. The write barrier detects publication of a private object and calls the `publishObject` function (described later) to transition its record to the shared state. A transaction acquires ownership of a record using an atomic compare-and-swap operation (CAS) in its open-for-write barrier [1, 27] and releases ownership and increments the version number when it ends (Txn end).

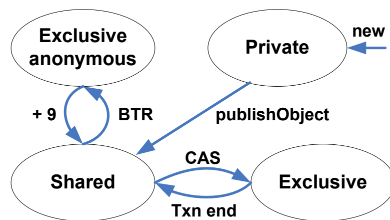

Figure 8. Transaction record state transitions.

3.2 Read and write barriers for enforcing isolation

Figure 9 shows the read and write isolation barrier instruction sequences for non-transactional code. The read barrier first reads the transaction record followed by the accessed address. If the object is not in the exclusive state, the barrier validates that the record did not change, ensuring that no other thread acquired ownership of the record after the first read of the record. If the record is in an exclusive state or if validation fails, then the read barrier invokes the conflict handler (`handleConflict`). By design, this barrier may not detect some conflicts between two non-transactional threads as such conflicts do not violate any transaction's isolation; it can detect such conflicts by simply checking the lowest-order bit.

The write barrier tries to acquire ownership of the record by flipping the lowest bit of the record with an atomic BTR instruction.[3] If the record is already in either of the exclusive states, then the write barrier invokes the conflict handler. After performing the write operation, the write barrier increments the version number and sets the record to the shared state by incrementing it by 9.

The barriers invoke the conflict manager whenever multiple threads access a shared location simultaneously with at least one of the accesses updating the location. The conflict manager backs off

```
readBarrier:
 mov ecx, [TxRec]
 mov eax, [addr]
 test ecx, 2
 jz readConflict
 cmp ecx, [TxRec]
 jne readConflict
readDone:
 ...
readConflict:
 push TxRec
 call handleConflict
 jmp readBarrier
```
(a) Read isolation barrier

```
writeBarrier:
 lock btr [TxRec],0
 jnc writeConflict
 mov [addr],val
 add [TxRec],9
writeDone:
 ...

writeConflict:
 push TxRec
 call handleConflict
 jmp writeBarrier
```
(b) Write isolation barrier

Figure 9. Read and write barriers for accessing shared data in a non-transactional thread.

and returns so that the barriers retry. Alternatively, conflicts could signal a race by throwing an exception or breaking to the debugger. Isolation barriers can thus aid in debugging concurrent programs.

3.3 Barriers for enforcing ordering

Lazy-versioning STM systems acquire transaction records on commit and release them after writing back updates to shared memory (or after aborting if commit fails). These systems do not need a read barrier to enforce isolation as dirty reads are never written to shared memory, but they do need one to enforce consistent ordering (as shown in Figure 1). The read barrier for enforcing ordering in a lazy-versioning STM thus simply checks for a pending update by a committed transaction:

```
test [TxRec], 2
jz readConflict
mov eax, [addr]
```

Note that this read barrier does not need to recheck the transaction record after the read because it needs to make sure only that the pending updates from the most recent transaction since the last synchronization action are done.

3.4 Quiescence for respecting privatization

A *quiescence* mechanism can provide partial isolation and ordering guarantees and can handle the privatization problem illustrated in Figures 1 and 4(b) without requiring non-transactional read or write barriers. Recent work in the context of unmanaged languages [32, 18] uses quiescence to ensure that doomed transactions (i.e., invalid transactions that have not yet aborted) do not cause run-time faults due to an inconsistent view of memory. They guarantee that a thread does not free memory while a doomed transaction could still access it.[4]

Other work [58, 52] demonstrates how to extend quiescence to handle the privatization problem by requiring that a transaction can complete only when all other transactions reach a consistent state. In the transactional variant of Figure 1, this ensures that, for an eager-versioning STM, the transaction in Thread 1 waits until Thread 2 can no longer access `item` before committing. Quiescence can also prevent the privatization problem in a lazy-versioning STM. Here, a transaction must wait until previously serialized transactions finish applying their updates to memory before completing itself. In Figure 1, this ensures that, when the transaction in Thread 1 completes, all updates from Thread 2 are already visible.

[2] We can also detect conflicts with either concurrent transactional or concurrent non-transactional writes by inspecting only the lowest bit.

[3] We can also use a compare-and-swap instruction.

[4] With managed languages, an STM can rely on garbage collection and type safety to avoid these problems.

```
readBarrier:
 mov ecx, [TxRec]
 mov eax, [addr]
 cmp ecx, -1
 jeq readDone
 test ecx, 2
 jz readConflict
 cmp ecx, [TxRec]
 jne readConflict
readDone:
 ...

readConflict:
 push TxRec
 call handleConflict
 jmp readBarrier
```

(a) Read isolation barrier

```
writeBarrier:
 cmp [TxRec], -1
 jeq privateWrite
 lock btr [TxRec],0
 jnc writeConflict
 * cmp val, 0
 * jz publicWrite
 * cmp [val+txFld],-1
 * jne publicWrite
 * push val
 * call publishObject
publicWrite:
 mov [addr],val
 add [TxRec],9
 jmp writeDone
privateWrite:
 mov [addr], val
writeDone:
 . . .
```

(b) Write isolation barrier

Figure 10. Read and write isolation barriers with dynamic escape analysis. The italicized code shows instructions due to dynamic escape analysis. In the read barrier, this code is optional. In the write barrier, the asterisked code is for reference types only.

Quiescence and other solutions that focus on privatization [52] rather than strong atomicity, do not solve general isolation and ordering problems such as speculative dirty reads and memory inconsistency. We also note that aggressive read-set validation [53, 18, 58] solves neither the general problems nor the privatization problem.

4. Dynamic escape analysis

Dynamic escape analysis detects if an object is *private* (visible to one thread) or *public* (visible to multiple threads). A freshly minted object is private and becomes public (is *published*) only when a reference leading to the object is written into either another public object or a static field. The read and write barriers for private objects are shorter and never perform a synchronized operation.

Once an object is public, our analysis leaves it public. Thread objects become public prior to the thread being spawned since both the spawning thread and the spawned thread have access to the thread object.

Figure 10 shows the instruction sequences for read and write isolation barriers with dynamic escape analysis. The italicized code shows new instructions compared to Figure 9. The read barrier reads the transaction record and the accessed adress and then skips over the rest of the barrier if the object is private. This privacy check is optional because, like the exclusive anonymous and shared states, the second-lowest bit of the transaction record is also set in the private state. The write barrier starts from doing the privacy check and skips the rest of the barrier if the object is private.

For writes of reference types the write barrier also contains the instructions to publish a private object that became public because of the write. (These instructions are marked with asterisk in Figure 10 and are not present for write barriers of non-reference types.) If the new value that is being written references a non-null private object then these insturctions call the function `publishObject` (Figure 11) to publish the written object before it is visible to other threads. Since the object is still private, `publishObject` does not concern itself with race conditions. Each object has associated with it a vtable containing a map of the object's fields holding references (*slots*). The slots are iterated over and the graph rooted by the object is traversed marking any private object encountered as public. A mark stack similar to those used by garbage collectors encoded the naturally recursive nature of the traversal. Once all reachable private objects are marked as public the object can be published.

```
void publishObject(object) {
    mark object public
    markStackPush(object);
    while (obj = markStackPop()) {
        forall (slots in obj) {
            if (*slot is private) {
                mark *slot public
                markStackPush(*slot);
}   }   }   }
```

Figure 11. Object publication algorithm.

transactional access	remove barrier outside atomic: read	write
none	yes	yes
only read	yes	no
only written	no	no
read and written	no	no

Figure 12. The barrier removal allowed by our not-accessed-in-transaction (NAIT) analysis.

The termination argument for the `publishObject` routine is similar to the one that guarantees a garbage collector's stop-the-world heap traversal terminates. The graph of private objects reachable from the root object is finite and fixed. Since the graph is private, objects cannot be added during the traversal. No private objects are reachable through public objects. Private objects are immediately annotated as public when first encountered. Later encounters will not continue the traversal beyond the public object, eliminating cycles of private objects. Every hop in the traversal that discovers a private object reduces the number of reachable private objects. Therefore the traversal will have to visit a finite number of nodes and needs to visit them a finite number of times.

In an eager-versioning, optimistic-read-concurrency STM system such as ours, compiler and runtime optimizations must consider that one transaction may read the dirty data of another transaction. Such a doomed transaction will abort eventually as it has read data speculatively written by another concurrently-executing transaction; but before it aborts, it can access objects published by the other thread. The compiler and runtime cannot assume, therefore, that a private object becomes visible to other threads only on commit — inside a transaction, a write of a reference into a public object immediately publishes any referenced private objects. Static escape analysis algorithms for detecting transaction-private objects must also take this into account.

5. Static not-accessed-in-transaction analysis

This section presents an effective whole-program static analysis that operates on Java bytecodes to optimize away read and write isolation barriers. The analysis is based on the following observation: *A memory write does not need a barrier* if the memory it writes is *never accessed in a transaction*. *A memory read does not need a barrier* if the memory it reads is *never written in a transaction*. Figure 12 summarizes this barrier-removal opportunity. Note that in a program not using transactions the analysis would remove all barriers.

There exists considerable prior work on identifying thread-local objects [3, 15, 8]. The not-accessed-in-transaction analysis (here-

after NAIT) complements thread-local analysis (hereafter TL) in two ways. First, truly thread-shared data may never be accessed in a transaction. A common example is "data handoff," such as objects that are transferred among threads via shared queues. Often the queues are accessed in critical sections, but not the objects passed through them. NAIT optimizes this situation naturally whereas TL requires complicated additions of limited effectiveness [12]. Another example is fields in subtypes of `Thread`, which are never thread-local. Second, NAIT and TL have complementary static approximations. For example, TL typically treats a static field as thread-shared even if only one thread ever uses it.

5.1 Pointer Analysis

For each field or array access outside a transaction, we need to know if it might access an object that is also accessed within a transaction, which is clearly an aliasing question. We use the Paddle [7] extension to Soot [56] to compute points-to sets for each bytecode that accesses memory. The analysis is a sound whole-program, field-sensitive, flow-insensitive analysis. (See Section 5.3 for discussion regarding whole-program analysis.)

Conceptually, the analysis is context-insensitive (OCFA) *after* code duplication, where for each method there is one version called during transactions and one called otherwise. After this duplication (a common implementation technique for transactions), a program point is "in a transaction" if and only if it is in the transactional version of a method or it is lexically in an atomic block. However, we do *not* perform this duplication on bytecodes; it is simpler and more efficient to do it lazily in the JIT.

Therefore, we *simulate* the effect of duplication by defining a new form of context-sensitivity during pointer analysis: The context is just "in transaction" or "not in transaction", so each method is analyzed in at most two contexts. (Hence efficiency is within a factor of two of OCFA and in practice, nowhere near the worst case.) All calls inherit the current context except calls lexically in `atomic` always analyze the callee under "in transaction." For the full effect of code duplication, we use *heap specialization*, meaning abstract objects are pairs of allocation site and context. Paddle's support for defining new kinds of contexts was crucial and elegant.

Hence after pointer analysis, each bytecode instruction that accesses memory has two points-to sets (one for each context of the enclosing method). Except for our novel definition of contexts, we are simply clients of pointer analysis.

5.2 Annotating Memory Operations

Given the points-to sets, annotating bytecodes with barrier-removal information requires only two more passes over the code. First, for each abstract object we compute how it may be accessed within transactions (the left column in Figure 12) by using (1) the "in transaction" points-to set for each load and store, as well as (2) the "not in transaction" points-to set for loads and stores lexically in `atomic`.[5] Second, for each load and store not lexically in `atomic`, we use its "not in transaction" points-to set and the result of the first pass to determine if the non-transactional version of the instruction needs an isolation barrier. No barrier is needed if the instruction is a load and no object in the points-to set is written in a transaction, or if the instruction is a store and no object in the points-to set is read or written in a transaction.

Though our focus has been on removing strong-atomicity barriers, the analysis information could also be used to remove STM operations within transactions. In particular, given weak atomicity, we could remove transactional open-for-read barriers [1] for the "in transaction" version if that points-to set contained no objects potentially written in a transaction. This is unsound under strong atomicity because the instruction may have a conflict with a non-transactional write.

[5] For the latter, the loads and stores *are* in transactions, but the *context* for the enclosing method is "not in transaction."

program	type	total	barrier removed by		
			NAIT-TL	TL-NAIT	TL+NAIT
JVM98	read	12671	8796	0	12671
	write	9885	7961	0	9885
tsp	read	106	89	0	93
	write	36	16	0	17
OO7	read	300	279	0	292
	write	136	114	2	117
JBB	read	804	364	24	798
	write	621	131	344	575

Figure 13. Static counts of barriers removed in reachable non-transactional code by NAIT but not TL (NAIT-TL), TL but not NAIT (TL-NAIT), and both analyses applied together (TL +NAIT).

5.3 Details

If the first use of a class `C` might be in a transaction, then its static initializer (method `clinit`) could run in a transaction. This method includes at least a write (bytecode `putstatic`) to each static field in `C`, which would naively prevent NAIT from removing any barriers on accesses to these fields. However, Java's class-initialization semantics prevents another thread from accessing these fields while `C` is being initialized. Therefore, an access of a static field of `C` within `C`'s `clinit` need not "count" for NAIT, and our empirical results include this improvement.

Soot/Paddle's whole-program pointer analysis is sound as is our specialized use of it. In general, Soot's soundness guarantee does require analysis users to provide classes that may be dynamically loaded or fields/methods that may be accessed from C code using JNI or via reflection. The benchmarks we consider do not require doing so.[6]

Whole-program analysis is not uncommon for concurrent programs [44, 48]. We believe it is practical even in a Java setting because one could modify a virtual machine to recompute analysis information incrementally when classes are dynamically loaded or C code uses JNI to access Java objects. In any case, our point is to show that NAIT is especially effective and should be exploited whenever possible.

5.4 Static Results

To give a sense of NAIT's effectiveness, we counted how many barriers were removed. (For benchmark descriptions and the effect on run-time, see Section 7.) We also implemented a straightforward TL analysis using the same points-to information for comparison purposes. Our results (Figure 13) show that for our benchmarks NAIT removes significantly more barriers and it removes almost all the barriers that TL removes.

The numbers we report include the non-transactional barriers for all object field, static field, and array accesses for nonlibrary classes, with two exceptions. First, we do not count instructions in methods the pointer analysis determines are unreachable. Second, in class initializers we do not count accesses to static fields of the class being initialized. For the former, barrier removal is always allowed but cannot affect performance. For the latter, barrier removal is sound without any analysis. In both cases, including these instructions in our counts would make our results appear better.

[6] `jbb` uses reflection, but only with constant strings. The analysis is sound without user intervention in this case.

Code generated for the source `a.x=0; a.y+=1;`

```
checknull a
t1 = ldfldaddr a.x
[t1] = stind.wb 0
t2 = ldfldaddr a.y
t3 = ldind.rb [t2]
t4 = add t3,1
[t2] = stind.wb t4
```

(a) Intermediate representation

```
cmp a, 0
jz nullPtrException
lock btr [a.txnfld],0
jnc conflict
mov [a.x],0
add [a.y],1
add [a.txnfld],9
```

(b) Generated code

Figure 14. Barrier aggregation example

Qualitatively, TL is ill-suited to common idioms that we see in `tsp` and `007`; for example, `tsp` uses fields of a subtype of `Thread` for data that is actually thread-local and accessed only outside transactions, but these fields are reachable from two threads (the one running and the one that created the object). For `jbb`, TL does better but NAIT still provides significant and complementary benefit.

6. JIT optimizations

Our JIT represents the non-transactional read and write barriers by annotations on the memory accesses. Such accesses map directly to the IA32 code sequences shown earlier (which vary depending on whether dynamic escape analysis is enabled) unless the JIT's own optimizations can either eliminate the barriers or combine the barriers for multiple memory accesses (*via barrier aggregation*).

The JIT does not insert barriers for accesses to immutable fields (e.g., final fields and internal fields such as the virtual method table or array length field) or immutable objects (e.g., objects of certain built-in classes such as `java.lang.Integer` or `java.lang.String`).

The JIT also detects and eliminates barriers to thread-local objects via a path-sensitive intraprocedural *escape analysis*, a traditional static escape analysis in contrast to the dynamic escape analysis of Section 4. Allocated objects begin thread-local and an iterative, forward dataflow analysis finds that objects escape when they are assigned to escaped locations (static variables or fields of escaped objects) or are reachable from method-call arguments. Aggressive inlining lowers the imprecision of the latter, and the use of types improves precision by ruling out incompatible assignments.

The code generator lowers the non-transactional read and write operations to the complete barrier sequences, exposing the operations within the barriers to further optimization at the basic-block level. *Barrier aggregation* then detects multiple barriers to the same object in the same basic block and combines them into a single aggregated barrier. Figure 14 shows an example that accesses the same object several times and thus is amenable to barrier aggregation. The IR generated by the JIT (Figure 14(a)) has two write-barrier-annotated store operations and one read-barrier-annotated load operation. Figure 14(b) shows the code generated after barrier aggregation. [7] The instruction sequence first performs a null pointer check and then attempts to acquire the transaction record. If it succeeds, it writes field `x` of object `a`, updates field `y`, and finally releases ownership by incrementing the record's version number.

The code sequence in Figure 14(b) is almost identical to that for a single write. We acquire the transaction record, perform operations on the object, and finally release ownership by setting the record to the incremented version number. Whereas unoptimized code acquires the record for every modification of an object, aggregated barriers acquire the record just once per multiple reads and writes to an object. In the presence of dynamic escape analysis aggregated barriers skip acquire and release of the transaction record if the object is private; the barriers also publish the objects that become public due to writes of reference types.

As with the standard barrier, aggregated barriers access a single object and perform a finite number of operations. To guarantee these properties and avoid deadlock, the JIT does not aggregate across basic blocks and does not allow function calls or access to multiple objects within an aggregated barrier.

[7] For simplicity, we show a code sequence without dynamic escape analysis.

7. Performance

We investigate the cost of strong atomicity and the effectiveness of our optimizations using both transactional and non-transactional workloads. For non-transactional benchmarks, we measure the overhead of strong atomicity by running each benchmark with and without our read and write isolation barriers. For transactional benchmarks, we investigate the performance of (1) a weakly atomic execution (with no isolation barriers), (2) a strongly atomic execution (with isolation barriers), and (3) a lock-based synchronized execution (with synchronized regions in the source instead of atomic ones). We show that enforcing strong atomicity has little effect on the scalability of multi-threaded transactional workloads. We also show that our optimizations are extremely effective in mitigating the overhead of non-transactional and single-threaded workloads.

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.

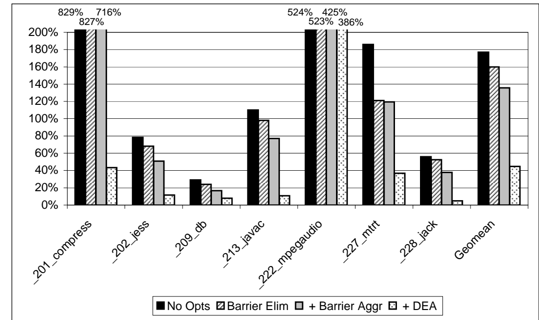

Figure 15. Overhead of strong atomicity on SPEC JVM98 (without whole-program optimizations).

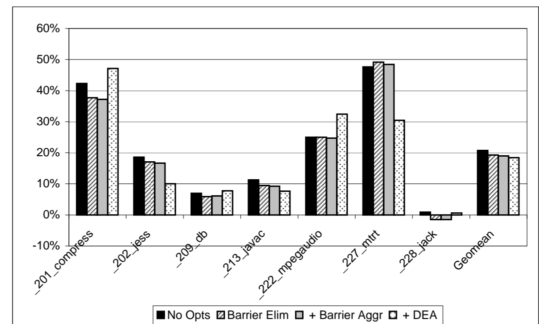

Figure 16. Read barrier overhead on SPEC JVM98.

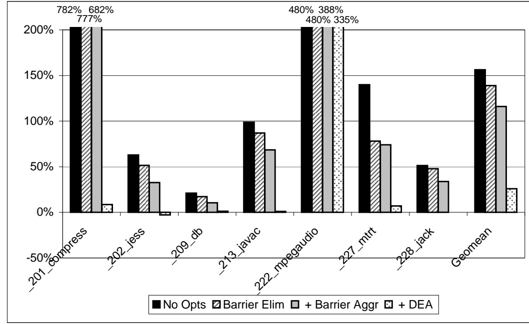

Figure 17. Write barrier overhead on SPEC JVM98.

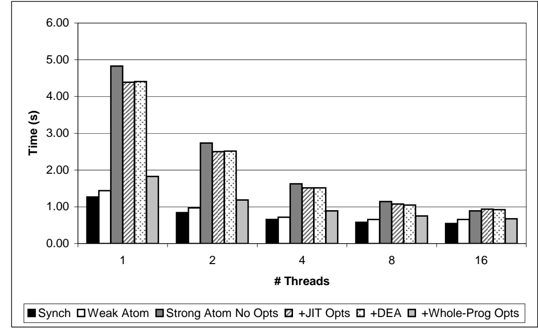

Figure 18. Tsp execution time over multiple threads.

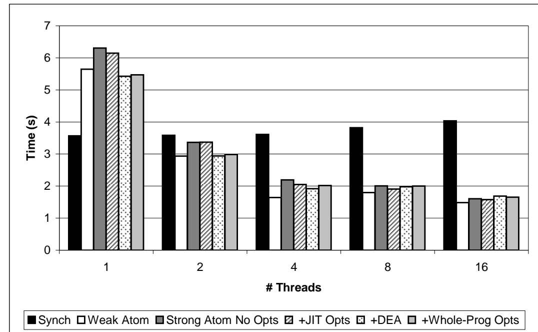

Figure 19. OO7 execution time over multiple threads.

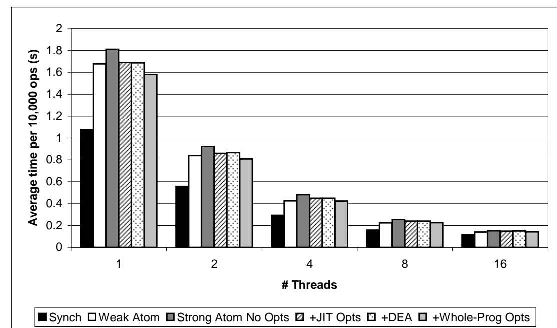

Figure 20. SpecJBB execution time over multiple threads.

We measure the cost of strong atomicity for non-transactional programs using the SPEC JVM98 [54] suite of benchmarks. We use steady-state execution time measured as the execution time of the third run of a benchmark during a single invocation. Figure 15 shows the overhead due to read and write isolation barriers with various levels of optimization. The `No Opts` bars show the overhead with no optimizations. The remaining bars show the cumulative influence of JIT and run-time optimizations: `Barrier Elim` shows the effect of barrier elimination for immutable object and fields and data detected to be thread-local by intra-procedural escape analysis, `+ Barrier Aggr` adds barrier aggregation, and `+ DEA` adds dynamic escape analysis (DEA). We also measured the overhead of inserting only read or only write barriers (Figures 16 and 17). These data both help to understand the nature of the strong atomicity overhead and provide an insight into the cost of enforcing different levels of isolation in an STM.

With no optimizations, the overhead of strong atomicity is significant - up to 8 times normal program execution time; the majority of the overhead comes from the cost of write barrier, which contains an expensive atomic instruction. Barrier elimination reduces the overhead by 30% on _227_mtrt, but has little effect on other benchmarks. Barrier aggregation significantly reduces the overhead for many benchmarks, especially for _201_compress and _222_mpegaudio where it succeeds in aggregating multiple accesses to an array. DEA dramatically reduces the remaining overhead for all benchmarks except _222_mpegaudio by practically eliminating the cost of write barriers. (For _201_compress its effect is especially impressive - the overhead goes down by an order of magnitude - from 700% to 40%.) DEA fails to remove the barrier overhead for _222_mpegaudio because that benchmark operates mostly on static arrays and static data is visible to multiple threads. The behavior of _222_mpegaudio shows that programming style can have significant effect on the performance of a strongly atomic STM system - unnecessarily exposing thread-private data to multiple threads may have detrimental effect on performance.

Note that Figures 15, 16 and 17 have no bars for the overhead in the presence of the whole-program optimizations. This is because for non-transactional programs not-accessed-in-transaction analysis (NAIT) removes all the barriers, and, thus, completely eliminates the overhead of strong atomicity.

We investigate the effect of strong atomicity on scalability using three multi-threaded transactional benchmarks - Tsp [57], OO7 [59] and SpecJBB [55]. For all the benchmarks, we created transactional versions by replacing the original Java synchronization with transactions.[8] Tsp solves a traveling salesman problem. In this benchmark, threads perform their searches independently, but share partially completed work and the best-answer-so-far via shared memory. OO7 performs a number of traversals over a synthetic database organized as a tree. Traversals either lookup (read-only) or update the database. OO7 allows transactional access (or locking, in the original version of the benchmark) at different levels of the tree. In our experiments we used root locking and a mixture of 80% lookups and 20% updates. SpecJBB, a well known Java server benchmark, emulates a 3-tier system for a wholesale company with multiple warehouses.

Figures 18, 19 and 20 show the performance of Tsp, OO7 and SpecJBB from 1 to 16 threads, where each thread is mapped to a different processor. The `Synch` bars show the performance of the lock-based versions, the `Weak Atom` bars show the performance of transactional versions under weak atomicity, and the rest of the bars show the performance of the transactional versions under strong atomicity with various levels of optimizations. The `Strong`

[8] In SpecJBB we did not convert to transactions the critical sections containing wait/notify and warehouse initialization

`Atom NoOpts` bars show the performance with no optimizations, `+JitOpts` add barrier elimination and barrier aggregation, `+DEA` adds DEA, and `+Whole-Prog Opts` add static whole-program optimizations (NAIT and TL).

The cost of unoptimized strong atomicity vs. weak atomicity for a single-threaded execution varies significantly depending on the benchmark (Tsp, which performs a lot of non-transactional accesses, is about 3 times slower; but there is less than 11% overhead for OO7 and SpecJBB, which spend the majority of their time in transactional code). Optimizations (especially NAIT), reduce the overhead of strong atomicity to less than 27%. (OO7 with DEA and SpecJBB with DEA and whole-program optimizations are even slightly faster than their weakly atomic versions. This happens because for OO7 dynamic escape analysis also speeds up some of the open-for-write operations [1] inside transactions.)

Also note that strong atomicity has no detrimental effect on scalability. For all three benchmarks strongly atomic executions scale as well as weakly atomic executions and as well or better than lock-based versions of the benchmarks. (Lock-based OO7 does not scale due to coarse-grained synchronization.) Moreover, as the number of threads increases, the cost of strong atomicity compared to weak atomicity drops further. With 16 threads the strongly atomic versions of Tsp, OO7 and SpecJBB are only 2%, 12% and 1% slower than their weakly atomic counterparts.

8. Related work

Several researchers [51, 21, 28, 39, 29] have implemented STM APIs that allow a programmer to access memory transactionally. These systems guarantee transactional behavior only for accesses that go through the STM API – they do not guarantee isolation in the presence of conflicting accesses that do not go through the STM API. Other researchers [25, 27, 1] have introduced first-class transaction constructs into Java or C# and implemented them via weak-atomicity STMs.

Harris et.al., [26] add transactions to Concurrent Haskell, a functional language. Their approach uses Haskell's monadic type system to segregate transactional data from other mutable shared data, thereby ensuring that transactional data is accessed only within a transaction. Neither the data segregation nor the monadic type system map naturally to an imperative language such as Java.

On a uniprocessor, one can implement strong atomicity in software using scheduler-based techniques to ensure no transaction is active during a non-transactional access [47, 37]. While extremely efficient on uniprocessors, the approach does not extend naturally to multiprocessors.

The new HPCS language proposals – Fortress [4], X10 [14], and Chapel [16] – all define a transactional memory construct in lieu of locks. Fortress [4] uses the terms *shared* and *local* where we use *public* and *private*; it states that segregating the two will enable optimizations of transactional reads and writes. The current X10 reference implementation implements atomic blocks as a single mutual exclusion lock [14]. These languages are still in flux and currently do not appear to require strong atomicity.

Hardware transactional memory (HTM) systems [30, 24, 13, 41] provide strong atomicity naturally because they leverage the existing cache coherence logic to implement transactions. Recent work has provided ways to support transactions with memory footprints that do not fit in cache [46, 5, 42, 43]. Hybrid transactional memory [17, 33] provides architectural support for mixing HTMs and STMs, relying on the latter when transactions become too large. Hardware accelerated STM (HASTM) provides architectural support to accelerate transactions executed entirely in software [50]. Our work establishes that STMs can provide scalable strong atomicity, which is important when transactional hardware is unavailable (e.g., today), and to guide research on architectural support for transactional memory.

Domani et al. [19] dynamically segregated thread local objects from globally visible objects, in much the same way we do, to reduce the cost of garbage collection and synchronization. Lev and Maessen [36] sketch techniques similar to our dynamic escape analysis to implement strong atomicity. They do not describe an implementation and thus do not present implementation details and quantitative measurements.

Compilers have used static escape analysis for removing synchronization overheads and stack allocation of objects [11, 15, 8, 3, 48]. These escape analysis techniques could also be used to eliminate strong atomicity barriers, but we demonstrated that not-accessed-in-transaction is much more effective on our benchmarks.

Whole-program analysis is common in research on eliminating synchronization overhead [48] or detecting data races [44, 20]. Autolocker [40] uses whole-program analysis to implement transactions in terms of locks, using programmer annotations to guide what locks protect what data. Our whole-program not-accessed-in-transaction analysis is novel because it targets lowering the cost of strong atomicity. Hindman and Grossman [31] briefly sketched a similar idea, but they had neither points-to information (relying only on type-based alias information) nor an optimized implementation of transactions. They also performed the analysis after creating two versions of each method rather than treating "in a transaction" as a new form of context-sensitivity.

9. Conclusion

The success of transactional memory depends on simple, intuitive rules and semantics. Isolation and consistent ordering are among the most important of these. We have shown how strong atomicity meets the simplicity criteria better than weak atomicity by categorizing and characterizing non-intuitive weak atomicity anomalies.

We have shown how the cost of providing simpler and stronger semantics is ameliorated by our novel optimizations, including dynamic escape analysis that leverages a conservative runtime approximation of thread local data to avoid barriers, and new static compiler optimizations that elide read and write barriers for objects not used in transactions. We have implemented these semantics and optimizations in the context of a high performance system and shown that strong atomicity is competitive with weak atomicity in performance while retaining simple and intuitive semantics.

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

[3] J. Aldrich, E. G. Sirer, C. Chambers, and S. Eggers. Comprehensive synchronization elimination for Java. *Sci. Comput. Programming*, 47(2–3), May–June 2003.

[4] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, Jr., and S. Tobin-Hochstadt. The Fortress language specification version 1.0α. http://research.sun.com/projects/plrg/fortress.pdf, 2006.

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

[6] C. S. Ananian and M. Rinard. Efficient object-based software transactions. In *SCOOL 2005*.

[7] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *PLDI 2003*.

[8] B. Blanchet. Escape analysis for object-oriented languages: Application to Java. In *OOPSLA 1999*.

[9] C. Blundell, E. C. Lewis, and M. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2), Nov. 2006.

[10] H. Boehm. Threads cannot be implemented as a library. In *PLDI 2005*.

[11] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *OOPSLA 1999*.

[12] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA 2002*.

[13] B. D. Carlstrom, J. Chung, A. McDonald, H. Chafi, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *PLDI 2006*.

[14] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005*.

[15] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for Java. In *OOPSLA 1999*.

[16] Cray Inc. Chapel Specification 0.4. http://chapel.cs.washington.edu/specification.pdf, 2005.

[17] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS 2006*.

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

[19] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In *ISMM 2002*.

[20] C. Flanagan and S. N. Freund. Type inference against races. In *SAS 2004*.

[21] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Technical Report UCAM-CL-TR-579.

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

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

[24] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). In *ASPLOS 2004*.

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

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

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

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

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

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

[31] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *MSPC 2006*.

[32] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. Mcrt-malloc: A scalable transactional memory allocator. In *ISMM 2006*.

[33] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP 2006*.

[34] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), 1981.

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

[36] Y. Lev and J.-W. Maessen. Towards a safer interaction with transactional memory by tracking object visibility. In *SCOOL 2005*.

[37] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time Java. In *IEEE Real-Time Systems Symposium 2005*.

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

[39] V. J. Marathe, W. N. Scherer, and M. L. Scott. Adaptive software transactional memory. In *International Symposium on Distributed Computing 2005*.

[40] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. In *POPL 2006*.

[41] A. McDonald, J. Chung, B. D. Carlstrom, C. Cao Minh, H. Chafi, C. Kozyrakis, and K. Olukotun. Architectural semantics for practical transactional memory. In *ISCA 2006*.

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

[43] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS 2006*.

[44] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *PLDI 2006*.

[45] 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*.

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

[47] M. F. Ringenburg and D. Grossman. AtomCaml: first-class atomicity via rollback. In *ICFP 2005*.

[48] E. Ruf. Effective synchronization removal for Java. In *PLDI 2000*.

[49] 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*.

[50] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural support for software transactional memory. In *MICRO 2006*.

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

[52] 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.

[53] M. F. Spear, V. J. Marathe, W. N. Scherer, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *DISC 2006*.

[54] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. http://www.spec.org/osg/jvm98.

[55] Standard Performance Evaluation Corporation. SPEC JBB2000, 2000. See `http://www.spec.org/jbb2000`.

[56] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON 1999*.

[57] C. von Praun and T. R. Gross. Static conflict analysis for multi-threaded object-oriented programs. In *PLDI 2003*.

[58] 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*.

[59] A. Welc, S. Jagannathan, and A. L. Hosking. Revocation techniques for Java concurrency. In *Concurrency and Computation:Practice and Experience*. John Wiley and Sons, 2005.