Inevitability Mechanisms for Software Transactional Memory

Michael F. Spear[†] Maged M. Michael[‡] Michael L. Scott[†]

[†]Department of Computer Science
University of Rochester [*]

[‡]IBM T.J. Watson Research Center

spear@cs.rochester.edu, magedm@us.ibm.com, scott@cs.rochester.edu

Abstract

Transactional Memory simplifies parallel programming by eliminating the use of locks to protect concurrent accesses to shared memory. However, when locks are used to provide mutual exclusion for irreversible operations (I/O, syscalls, calls to "black box" libraries), their replacement with transactions seems problematic: transactions can abort and restart at any arbitrary point in their execution, which is unacceptable when operations performed during a transaction have made the intermediate state of that transaction visible to an outside agent.

Permitting at most one transaction to operate in an "inevitable" mode, where it is guaranteed to commit, is widely accepted as a solution to the irreversibility problem for transactions, albeit one that is not expected to scale. In this paper we explore a variety of mechanisms to support inevitability in software transactional memory. We demonstrate that it is possible for an inevitable transaction to run in parallel with (non-conflicting) non-inevitable transactions, without introducing significant overhead in the non-inevitable case. Our mechanisms can also be used to improve the speed of certain common-case transactions.

1. Introduction

Transactional Memory (TM) [11] provides a simple and powerful mechanism for synchronizing concurrent access to data: programmers mark regions as `atomic` and rely on the underlying system to execute atomic sections concurrently whenever possible. Current TM designs may employ hardware, software, or a combination of the two, and may offer nonblocking guarantees or use locks to implement transactions.

Unfortunately, TM (and software TM (STM) [21] in particular) bears considerable runtime and semantic overhead. An STM must instrument program loads and stores, and may require up to $O(n^2)$ time and $O(n)$ space overhead to ensure the correctness of a transaction, where n is the number of locations accessed. Furthermore, any TM that permits optimistic concurrency must provide a rollback mechanism to resolve conflicts. Since rollback can happen at any time during the execution of a transaction, the programmer cannot safely perform operations that have irreversible side effects—e.g., I/O and system calls—from a transactional context. In a library-based STM, it is also unsafe for transactional code to call precompiled binaries that may access shared locations, unless binary rewriting is available [5, 18, 27].

A straightforward solution to these problems is to identify certain *inevitable* transactions, at most one of which can be active at a time, and to ensure that such a transaction never aborts. Once rollback is prevented, an inevitable transaction may safely perform I/O, syscalls, and other irreversible operations without concern that they will ever need to be "undone". It may also be able to skip certain bookkeeping operations associated with rollback, allowing it to run faster than an abortable transaction—an observation that raises the possibility of judiciously employing inevitability in the underlying system as a performance optimization for transactions that don't actually require its semantics.

[*] At the University of Rochester, this work was supported in part by NSF grants CNS-0411127, CNS-0615139, CCF-0702505, and CSR-0720796; by equipment support from IBM; and by financial support from Intel and Microsoft.

1.1 Inevitability in Existing TM Systems

The original TCC proposal [7], which serialized concurrent transactions at their commit points, provided a notion of "age" through which inevitability could be provided; in effect, the active transaction that had started earliest in time was guaranteed to commit. TCC's age-based inevitability served both to prevent starvation and to allow irreversible operations inside transactions: to perform such operations, the transaction waited until it was "oldest" and then could proceed safely. Subsequent discussions of inevitability tend to focus on this design point [1, 12, 22], and assume that an in-flight inevitable transaction must prevent all concurrent transactions from committing.

In software, this design is exemplified by JudoSTM [18], where all concurrent transactions block at their commit point when an inevitable transaction is in-flight. While this permits some concurrency (a transaction can execute from its begin point up to its commit point during the inevitable transaction's execution), it does not allow any non-inevitable transaction to commit until the inevitable transaction completes. Moreover several features of JudoSTM, including dynamic binary rewriting and value-based conflict detection, limit the applicability of their method to other TM designs.

To the best of our knowledge, the only exception to what we might call the "commit-token model" of inevitability is the "unrestricted" hardware TM of Blundell *et al.*, which allows non-inevitable transactions to commit alongside an inevitable transaction. This system exposes the inevitable status of a transaction to the hardware, and leverages the fact that the cache coherence protocol can identify conflicts between inevitable reads and non-inevitable writes. Conflicts between an inevitable transaction and concurrent non-inevitable transactions are resolved by favoring the inevitable transaction; non-inevitable transactions are allowed to commit when no such conflicts are detected.

1.2 Contributions

This paper focuses on software TM. It presents five mechanisms that allow non-inevitable read-only transactions to commit while an inevitable transaction is active. Two of these mechanisms also allow non-inevitable writer transactions to commit. These five approaches to inevitability are largely orthogonal to the underlying TM algorithm. Any STM that provides versioning and ownership through metadata should be able to use our mechanisms with little modification, and we believe that hardware and hybrid TMs could support some of these mechanisms with little modification.

Through a series of microbenchmarks, we demonstrate that our mechanisms introduce very little latency and allow high levels of concurrency. We also assess the orthogonal question of whether inevitability is a reasonable mechanism for enhancing performance.

We consider a workload characterized by a large pool of tasks, each represented by a transaction, where tasks do not synchronize with each other. In such a setting, we find that inevitability can improve throughput so long as transactions are not trivial, but the effect decreases as parallelism increases.

2. STM Support for Inevitability

Most STM runtimes control concurrent access using metadata in object headers or in a hash table indexed by data address. Intel's McRT [19] and Microsoft's Bartok [9] use the metadata to lock locations on first access and make modifications in-place. TL2 [3] and some variants of TL [4] and RSTM [24] buffer updates, and only lock locations at commit time. Other STM algorithms, such as DSTM [10], OSTM [6], ASTM [14], RSTM [15], and NZSTM [25] use more complex metadata to achieve obstruction freedom.

While inevitability fundamentally introduces blocking into the STM, we see no other obstacles to the use of any of our mechanisms in all existing STMs. In our formulation, inevitability entails only two requirements, the first of which is trivial:

- At Most One Inevitable Transaction: The runtime provides a `become_inevitable()` call, which transitions a running transaction into inevitable mode.[1] The reference implementation of `become_inevitable` employs a test-and-set lock with exponential backoff to block other inevitable transactions.
- Inevitable Transactions Do Not Abort: After returning from `become_inevitable`, the calling transaction must not abort if it has performed irreversible operations.

This second requirement is the principal implementation challenge. We further classify the sources of aborts as self, explicit, and implicit, corresponding to aborts caused by program logic, read/write-after-write conflicts, and write-after-read conflicts.

Preventing Self Abort Many STM APIs include a `retry` mechanism [8] for efficient condition synchronization. Clearly this mechanism cannot be used after performing an inevitable operation. It may be possible, however, to perform condition synchronization *before* becoming inevitable or, in certain cases, in ways that are guaranteed never to force rollback of work already performed; we consider these options further in Section 4.3.

Preventing Explicit Aborts In all of the STM algorithms named above, an in-flight transaction must, at some point prior to committing, acquire exclusive ownership of each location it wishes to modify. Subsequent to this acquisition step, any concurrent reader or writer may use contention management [20] to receive permission to abort the exclusive owner. For inevitable transactions that use encounter-time locking, it is straightforward to augment contention management so that non-inevitable transactions always defer to inevitable transactions. This sacrifices nonblocking guarantees, but is otherwise a trivial extension. It also decreases overhead within inevitable transactions, since data written (or overwritten) by inevitable writes need not be logged. Furthermore, given the condition that there is at most one inevitable transaction, this does not introduce deadlock.

Preventing Implicit Aborts While some STM algorithms support a "visible reader" mode, in which transactions explicitly mark the metadata of locations they read, it is more common simply to record transactional reads in a private log, which is then validated at commit time to make sure that corresponding metadata has not changed. The runtime must ensure that inevitable transactions do not detect such changes after becoming inevitable, via some additional guard on reads. This requirement places the inevitable transaction somewhere between fully "visible" and fully "invisible," depending on the inevitability implementation.

Implicit aborts can trivially be prevented using a global read/write lock (GRL): a call to `become_inevitable` blocks until all in-flight transactions commit or abort and clean up metadata. The inevitable transaction then executes, while all other threads block at the start of their next transaction until the inevitable transaction commits. Unfortunately, this solution affords no concurrency, even in workloads with no conflicts. We now turn our consideration to mechanisms that prevent implicit aborts without sacrificing all concurrency.

3. Inevitability Mechanisms

As suggested in Section 2, the primary requirement of a concurrency-permitting inevitability mechanism is a method to guard the inevitable transaction's read set, so that a conflicting (non-inevitable) writer does not invalidate an inevitable read. In practice, this requirement entails a tradeoff between precision and overhead: more precise mechanisms afford greater concurrency between the inevitable transaction and transactions that write to disjoint areas of memory, but at the cost of higher overhead within all in-flight transactions. We have already described the global read/write lock (GRL). In the remainder of this section we describe five additional mechanisms that afford progressively higher levels of concurrency.

3.1 Global Write Lock

Read-only transactions may progress (and commit) alongside an inevitable transaction so long as the inevitable transaction updates the metadata of locations that it writes (thereby enabling the concurrent readers to detect conflicts). Similarly, a writing transaction that uses commit-time locking can progress up to its commit point while an inevitable transaction is in-flight, but must wait to commit until after the inevitable transaction finishes. We refer to this inevitability mechanism as a global write lock (GWL).

When a transaction attempts to become inevitable, GWL makes no assumptions about the state of other threads. The inevitable transaction must thus check metadata before performing any transactional read (requiring a read-before-read (RBR) memory fence on processors with relaxed memory consistency models) and might need to block until a lock is released. To avoid races with non-inevitable writers that have committed or aborted but not yet cleaned up, the inevitable transaction must also use atomic instructions (`compare-and-swap` (CAS) or `load linked/store conditional` (LL/SC)) to update metadata for writes. At the same time, it can elide all read-set logging, and need not postvalidate metadata after a transactional read.

Since inevitable transactions do not validate their read set, a non-inevitable writer must check the GWL flag after acquiring any location, to detect the presence of a concurrent inevitable transaction; if none is detected, the transaction may proceed. Otherwise it must release the location and restart. Testing the GWL flag after acquiring requires an acquire fence. On architectures like the x86 and SPARC, an atomic CAS provides an acquire fence implicitly. On the PowerPC architecture, a lightweight instruction-sync instruction can be used after the acquires to impose acquire fence ordering (which is needed in any case in STM implementations with deferred validation).

GWL allows concurrent non-conflicting read-only non-inevitable transactions to complete during an inevitable transaction's execution, but we expect GWL to afford only limited parallelism with read-write transactions, since it prevents a non-inevitable writer

[1] In restricted cases, static analysis may determine that two inevitable transactions will never conflict with one another. It may then be possible to allow multiple concurrent inevitable transactions.

from committing even when it could logically serialize prior to a concurrent inevitable transaction. We also expect that supporting GWL introduces minimal overhead: with commit-time locking, we add only a single memory fence and comparison to the commit sequence of the non-inevitable transaction. Furthermore, we anticipate that GWL transactions should run significantly faster than their non-inevitable counterparts, even when the semantics of inevitability are not strictly required: there is no need for logging and validating the consistency of reads, which usually account for a significant portion of the overhead of inevitable transactions.

In GWL, while inevitable reads avoid logging and validation, still each inevitable read must check whether the location to be read is in the process of being written by a concurrent non-inevitable transaction, and if necessary wait until the location is released. In the next two designs, we explore the possibility of eliminating per-inevitable-read overheads completely.

3.2 GWL + Transactional Fence

Several STMs include quiescence mechanisms [3, 6, 13, 23, 26] that permit threads to wait for concurrent transactions to reach predictable points in their execution. These mechanisms support memory reclamation and privatization, or ensure correctness during validation. By using quiescence to eliminate the possibility of not-yet-cleaned-up transactions, we can avoid the need to instrument inevitable reads, at the cost of a possible delay after becoming inevitable. (Transactional writes still require instrumentation to ensure that locations are acquired before they are written. Without this instrumentation, concurrent readers would not be able to correctly detect conflicts.)

To evaluate the utility of quiescence, we use a Transactional Fence [23], which waits for all active transactions to commit or abort *and clean up*.[2] Once the transactional fence completes, the inevitable transaction is guaranteed that there will be no concurrent updates by non-inevitable transactions during its execution.

This same mechanism could be used in conjunction with GRL: upon becoming inevitable, a transaction could wait for all concurrent transactions to complete. Since all new transactions (even those that do not perform writes) would block at their begin point, once the transactional fence completed, the inevitable transaction would be guaranteed exclusive access to shared locations, and could safely perform uninstrumented reads *and writes*.

3.3 Writer Drain

While waiting for all active transactions to finish is a straightforward way to ensure that an inevitable transaction can always safely read, it is more conservative than necessary. All that is really required is to wait until all concurrent *writers* have completed. We can do this by merging a writer count into the GWL flag. We call the resulting mechanism a *Writer Drain*. With the Drain in place, an inevitable transaction requires neither instrumentation of reads nor atomic instructions in metadata updates for writes.

Abstractly, the Drain is an implementation of GWL in which an inevitable transaction cannot start until all non-inevitable writers have released their metadata. When commit-time locking is used, this behavior can be ensured using a single-word status variable updated according to the protocol shown in Figure 1. The status variable consists of three fields: a bit i indicating an active inevitable transaction, a reservation bit r indicating a transaction that will become inevitable as soon as extant non-inevitable writer transactions finish, and a count n of the number of such writers. This protocol resembles a traditional fair reader-writer lock in which inevitable transactions take the role of "writer" and non-inevitable writer transactions take the role of "reader".

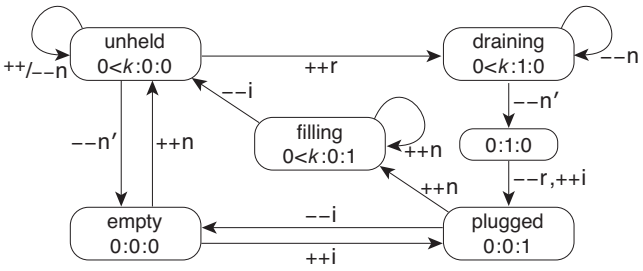

Figure 1. State Transitions for the Writer Drain. State labels indicate the values of the drain's three fields: $n:r:i$. Transitions that change r or i are made by an inevitable transaction; transitions that change n are made by non-inevitable writer transactions. If the current state lacks a desired outgoing transition, the thread that wishes to make that transition must wait. "--n'" indicates completion of the only remaining non-inevitable writer.

In the draining state, thread I has reserved permission to execute inevitably, but is waiting for all non-inevitable transactions *that are about to acquire a location* to complete. When the last of these transactions (indicated by an n' transition) completes, the drain transitions through a transient state to the plugged state. In the plugged or filling state, thread I may execute without restriction, but non-inevitable transactions must block at the point where they wish to acquire a location. If there are pending transactions in the drain when I completes, the drain moves to an unheld state, in which non-inevitable transactions must inform the drain of their status but may otherwise execute without restriction. When there are no inevitable transactions, and no non-inevitable transactions have acquired locations, the drain is empty.

When the drain is plugged or filling, the inevitable transaction is guaranteed that all metadata is unacquired; it will never encounter a locked location. During its execution, it is also guaranteed that no other transaction will modify metadata. Given these guarantees, the inevitable transaction requires no instrumentation of reads. In addition, though it must still modify metadata before writing the associated data, it can use ordinary writes to do so. In most architectures, such write-before-write ordering has no overhead; in the PowerPC, it can be achieved with a lightweight-sync instruction.

Like GWL, Drain should only afford limited concurrency with read-write transactions since non-inevitable writers cannot commit during the execution of any inevitable transaction. Furthermore, in the absence of inevitable transactions we expect higher overhead, since writing transactions must increment and decrement the non-inevitable active count in the drain at the beginning and end of their commit sequence (or, in the case of encounter-time locking, before first acquire and after commit, respectively). In the face of any concurrency, these updates will result in adverse cache behavior. However, the throughput of the inevitable transaction should be substantially higher. Consequently, an intelligent just-in-time compiler could reduce the code of an inevitable transaction to almost the size and complexity of a nontransactional equivalent.

3.4 Inevitable Read Locks

Both GWL and Drain use a single guard to protect all locations read by the inevitable transaction. Though simple, these mechanisms limit concurrency, since concurrent writers are prohibited from committing even when their write sets do not overlap with the inevitable transaction's read set. We now consider a precise mechanism to guard only the locations read by the inevitable transaction,

[2] Less expensive quiescence mechanisms may also suffice, though they may be specific to particular TM implementations.

thereby allowing a maximum amount of concurrency between the inevitable transaction and disjoint writers.

Our Inevitable Read Locks (IRL) mechanism adds read locks to the underlying STM's metadata. Typically, lock-based STMs overload a single word with lock and ownership information; the least significant bit is used to indicate locking, with all other bits compose a version number or pointer to the lock holder. Assuming that all pointers must be aligned to at least 4 bytes, we can take another low-order bit and use it to indicate that the location is being read (but not written) by the inevitable transaction.

Using IRL, inevitable transactions must read and (potentially) atomically update metadata on every transactional read, and must release read locks (using normal stores) upon commit. During a read or write, the inevitable transaction may detect a conflict with an in-flight transaction writing the same location. To resolve the conflict, the inevitable transaction must issue a remote abort, perhaps after briefly waiting briefly for the conflicting transaction to complete. In STMs that use encounter-time locking with in-place update, the inevitable transaction may need to wait after issuing a remote abort, so that the aborted transaction can undo any modifications it made.

Non-inevitable transactions also require minor modification to support IRL. During reads, transactions may ignore benign conflicts with an inevitable reader (detected when the transaction reads an ownership record and finds the read bit set), but during transactional writes, the runtime must prevent acquisition of locations that are actively being read by an inevitable transaction. To resolve such conflicts, the writer can either block or abort itself.

While IRL should provide a high degree of concurrency between inevitable transactions and nonconflicting writing transactions, we expect scalability to be limited for a number of reasons. First, inevitable transactions incur significant overhead due to a high number of expensive atomic operations. Secondly, inevitable transactions must still bookkeep their reads, so that they can release their read locks upon commit. Worse yet, the additional bus messages generated by the increased number of atomic operations may cause a slowdown for truly disjoint non-inevitable transactions. Lastly, as with visible reads in STM, we expect decreased scalability due to cache effects: updates to metadata by the inevitable transaction cause additional bus messages and cause cache misses when concurrent non-inevitable readers perform validation.

3.5 Inevitable Read Filter

By approximating the set of read locks as a Bloom filter [2], our Inevitable Read Filter (Filter) mechanism decreases impact on concurrent reads, albeit at the expense of a more complex protocol for handling non-inevitable writes and inevitable reads. An inevitable transaction records the locations it reads (or the locations of the associated metadata) in a single, global Bloom filter, and writing transactions refrain from acquiring locations recorded in the filter. The size of the filter and the set of hash functions are orthogonal to the correctness of the mechanism, and serve only to decrease the frequency of false conflicts.

Since inevitable readers and non-inevitable writers both must interact with the filter and with ownership metadata, we must take care to avoid data races. The non-inevitable writer always acquires a location before checking the filter, and upon a positive lookup in the filter the writer must abort. Similarly, the inevitable reader always records locations in the filter before checking metadata, and if that subsequent metadata check fails, the inevitable transaction blocks until the ownership record is released.

The complexity of the above protocol for interacting with the filter and ownership metadata is compounded by the processor memory model; the inevitable transaction must write to the filter before reading a location, and non-inevitable transactions must acquire locations (via atomic writes) before testing whether those locations are present in the filter. In the former case, an explicit write-before-read (WBR) memory fence is required on the x86, SPARC, and PowerPC architectures. The latter case introduces the need for a full sync instruction (SYNC) on the PowerPC, but no additional ordering on x86 or SPARC. With commit-time locking, a single SYNC suffices for the non-inevitable transaction, as it can acquire all needed metadata, issue the SYNC, and then test all acquired locations in the filter. Until hardware designers decrease the best-case overhead of WBR fences, their presence on the critical path of committing transactions introduces significant overhead.

Due to the unavoidable requirement for WBR ordering, we expect high overhead for the inevitable transaction using Filter. However, we expect better scaling than IRL, since read-read concurrency does not introduce any overhead, and does not sacrifice the read-write concurrency of IRL. The ability to tune the filter by changing its size or hash functions may prove useful as developers gain experience with transactional workloads.

3.6 Summary

Table 1 summarizes the impact of each inevitability mechanism on the behavior of concurrent non-inevitable transactions, and also identifies which mechanisms may introduce a delay in the inevitable transaction at its begin point. In Section 4 we discuss limitations that an inevitability mechanism imposes on the sorts of irreversible operations that can be performed; those limitations may, in turn, require a fallback mechanism to use GRL when the properties of another inevitability mechanism prove inadequate. Table 2 summarizes the sources of latency introduced by each mechanism.

4. Programming with Inevitability

We now turn our attention to the impact of inevitability on transactional programming models. For simplicity and generality, we assume that the STM is implemented as a library, and we ignore the concurrent execution of transactional and nontransactional code.

4.1 Irreversible Operations

In this section we contrast inevitability with other ways of accommodating irreversible operations, centering the discussion on the desirable properties that are intrinsic to inevitability.

Enforcing Mutual Exclusion of Multi-Instruction I/O Alternatives to inevitability for I/O include buffering, open nested transactions [17], and punctuating transactions [22]. Buffering is not a general solution for interactive I/O. We therefore limit our discussion to the other two alternatives. For each of these, we consider their use in a simple, output-only operation.

Listing 1 presents transactional operations for an abstract set initialized via the `init()` function. We consider an array of such sets, accessed via `WorkLoad1` from Listing 2. For $N>1$ threads, each thread $n \in N$ executes `WorkLoad1(n)`. We assume that `SetPrint()` is executed inevitably, via an API call that correctly ensures at most one inevitable transaction in flight at any time.

Since inevitability enforces mutual exclusion, there will be no interleaving of print operations between threads. Thus while the order in which N sets are printed cannot be predicted, inevitability guarantees that each of the N sets will be printed without interruption by concurrent printing threads. This property does not hold for open nesting or punctuating transactions.

Under open nesting, calls to `SetPrint` can abort; consequently, the programmer must specify compensating actions to undo or mitigate the effects of `print` instructions issued from transactions that call `SetPrint` but do not commit. Since such actions are properties of the specific irreversible operation performed, it appears unlikely

Mechanism	Delay upon becoming inevitable	concurrent read-only	concurrent writes
GRL	Yes	No	No
GWL	No	Yes	No
GWL + Fence	Yes	Yes	No
Drain	Sometimes	Yes	No
IRL	No	Yes	Yes
Bloom	No	Yes	Yes

Table 1. Summary of benefits and drawbacks of different inevitability options.

Mechanism	Inev Read Instr	Inev Write Instr	Inev Read Logging	Inev Commit Overhead	Begin Overhead	NonInev Commit Overhead
GRL	None	None	No	None	WBR	N/A
GWL	Wait	Acquire	No	None	None	Test
GWL + Fence	None	Store	No	None	WBR	Test
Drain	None	Store	No	CAS	None	2 CAS
IRL	Acquire	Acquire	Locks	None	None	None
Bloom	Write WBR Wait	Acquire	Filter	None	None	WBR Intersect

Table 2. Overheads imposed by inevitability mechanisms.

Listing 1 Add, Remove, and Print transactions for a simple set.

```
Set sets[numthreads];

sub init()
 foreach(set in sets)
   set.fill_with_random_vals();

sub SetPrint(Set S)
 atomic
   print("Set " & S.id);
   foreach (s in S)
     print("  " & s.toString());
   print("End of set " & S.id);

sub SetAdd(Set S, value v)
 atomic
   if (!S.contains(v))
     S.add(v);

sub SetRemove(Set S, value v)
 atomic
   if (S.contains(v))
     S.remove(v);
```

Listing 2 Parallel workloads for an array of sets.

```
sub WorkLoad1(int tid)
 SetPrint(sets[tid]);

sub WorkLoad2(int tid)
 do
   if (tid == 0)
     SetPrint(sets[0]);
   else if (tid % 2 == 0)
     SetAdd(sets[0], random());
   else
     SetRemove(sets[0], random());
 until (TimerInterrupt)
```

that a single design pattern or library call can be provided to make this task easy for programmers.

With punctuated transactions, individual I/O operations need not be rolled back. However, at each I/O point, the transaction effectively commits and starts a new transaction; to ensure correctness, the programmer attaches a set of predicates to the punctuating action (in this case, the call to `print`), specifying what conditions must be restored before the thread resumes a new transactional context that appears to be a continuation of its previous context. Unfortunately, at the point where the transaction is punctuated, the scheduler may permit any concurrent thread to execute the first of its `print` statements. We believe that this problem can be prevented through the explicit creation of a monitor-like mechanism that uses additional punctuating transactions to synchronize calls to `SetPrint`. However, even if the runtime provides such a mechanism, it would introduce a risk of deadlock and would reintroduce the need for global reasoning about locks.

Preserving Local Reasoning About Correctness While inevitability is a global property (at most one active transaction can be inevitable), it does not violate abstraction boundaries or require global reasoning about correctness. We now consider `WorkLoad2` from Listing 2, and demonstrate how alternatives to inevitability such as privatization and punctuating transactions can necessitate deep changes to the structure of an algorithm.

Under `WorkLoad2`, a single thread attempts to print a set that is being actively modified. The runtime must ensure that the output matches some dynamic instance of the set. Let us suppose that the set initially contains all odd values within the range $n \ldots m$; after $thread0$ prints values $n \ldots m/2$, the elements $n \ldots m$ are all removed from the set, and then all even elements in the range $m/2 \ldots m$ are added. With punctuating transactions, all set modifications can complete during a break in the execution of $thread0$. If $thread0$ continues printing, the output will not match any dynamic instance of the set. However, encoding an invariant to capture the requirement that all previously printed values remain in the set may be unacceptably complex: it must at least assert that all printed values are still in the set, and may also need to understand the set

iterator's implementation, so it may assert that there have not been updates that fall within the range already printed.

Remarkably, even privatization fails to allow `WorkLoad2` to execute correctly without changes to the set implementation. A naive use of privatization would excise the entire set, print it, and then de-privatize the set, but this is not enough: we must also also somehow lock the set, so that no set-changing operations are executed and all lookup operations block. Without locking, it may be impossible to correctly de-privatize the set after I/O completes, since failed removals may be lost, or duplicate inserts occur.

4.2 Non-Transactional Code and System Calls

A limitation of library-based STM is its inability to call precompiled code. Recompilation of libraries can solve this problem, as can dynamic binary rewriting [5,18,27]. Without such mechanisms, precompiled libraries and system calls can still be called safely from an inevitable transaction, so long as the inevitability mechanism and the library code obey the following rules.

First, the underlying STM must not use indirection. In an indirection-based STM [6, 10, 14, 15], read instrumentation is necessary in inevitable transactions, because data does not reside in place. Indirection-free systems have no such constraint, because we use encounter time locking and in-place update for the inevitable transaction.

Second, if the library code reads and writes shared locations, it must be instrumented (though not necessarily by the library itself) according to the inevitability mechanism. We provide *inevitable prefetch* instructions for this purpose: a write prefetch locks its corresponding location without modifying the value, and a read prefetch performs the appropriate read instrumentation (such as adding locations to a filter, acquiring read locks, or ensuring that write locks are not held). Without prefetching, only GRL can call precompiled code that writes locations that have not been previously written by the calling transaction, and GWL, IRL, and Filter cannot call precompiled code that reads new locations. With prefetching, library code can be called only if its writes can be conservatively predicted. Furthermore, if the underlying inevitability mechanism is not Drain or GWL+Filter, the code's reads must also be predictable. The bottom line: inevitability and prefetching together can be used to call some, but not all, precompiled library code.

Using inevitability to call precompiled code decreases scalability relative to dynamic binary rewriting and recompilation. Inevitability serializes transactions even when the library code they execute does not perform irreversible operations. Furthermore, some rewriting mechanisms achieve the effect of speculative lock elision when calling lock-based code [18]. In contrast, inevitability ensures that the calling transaction will not abort while holding a lock, (avoiding the chance of deadlock if the lock is not released correctly), but cannot extract parallelism that may be available.

4.3 Condition Synchronization

Whereas inevitability improves transactional programming by adding support for irreversible operations, system calls, and precompiled libraries, it complicates the use of condition synchronization via explicit self-abort. As mentioned in Section 2, we explicitly forbid inevitable transactions from self-aborting after performing irreversible operations.

This condition does not, however, prevent condition synchronization within inevitable transactions. An inevitable transaction can certainly synchronize at its begin point. More generally, it can synchronize before executing `become_inevitable`, and at any point in its inevitable execution at which it has only performed reads (writes, library calls, and system calls must be conservatively considered irreversible operations).

Additionally, an inevitable transaction can synchronize using self-abort after performing irreversible operations in systems that provide closed-nested transactions, as long as (1) the retry is performed within a specially annotated closed-nested transaction; (2) the retry condition involves only reads to locations that were not accessed inevitably; (3) the inevitability mechanism permits concurrent writers to commit; and (4) the condition can be computed and written by any transaction without conflicting with the inevitable transaction's reads and writes. Since these properties do not cleanly compose and cannot in general be statically checked, we expect the usefulness of such condition synchronization to be limited.

5. Evaluation

In this section we analyze the performance of our different inevitability mechanisms across several microbenchmarks. We conducted all tests on an IBM pSeries 690 (Regatta) multiprocessor with 32 1.3 GHz Power4 processors running AIX 5.1. Our STM runtime library is written in C and compiled with gcc v4.0.0. All benchmarks are written in C++ and compiled using g++ v4.0.0. Each data point represents the average of five trials, each of which was run for five seconds.

STM Library Configuration Our STM library is patterned after the per-stripe variant of TL2 [3], and uses commit-time locking with buffered updates. We use an array of 1M ownership records, and resolve conflicts using a simple, blocking contention management policy (abort on conflict). Unlike the x86 TL2 implementation available with the STAMP suite [16], we do not mitigate conflicts on TL2's global timestamp via probabilistic mechanisms.

Inevitability Model Due to the lack of standard transactional benchmarks, much less those requiring inevitability, we use parameterized microbenchmarks to assess the impact on latency and scalability imposed by our algorithms. For each test, all threads are assigned tasks from a homogeneous workload. A single thread is instructed to perform each of its transactions inevitably, with all other threads executing normally. In this manner, we can observe the impact on scalability introduced by frequent, short-lived inevitable transactions, as well as the effectiveness of different inevitability mechanisms for speeding a workload of non-synchronizing transactions. Our choice to execute all inevitable transactions from the same thread increases the predictability and regularity of our results by avoiding cache contention for the inevitability token.

Our decision to analyze short inevitable transactions is deliberate. Baugh and Zilles [1] have argued that transactions that perform I/O are likely to run a long time, and to conflict with almost any concurrent activity. This suggests that quiescence overhead will be an unimportant fraction of run time, and that there is little motivation to let anything run in parallel with an inevitable transaction. While this may be true of operations that write through to stable storage, it is not true of more lightweight kernel calls, or of calls to pre-existing libraries (including buffered I/O) that are outside the control of the TM system. We are currently experimenting with a graphical game that uses transactions to synchronize with concurrent 3-D rendering in OpenGL. Without inevitability, the rendering thread may be forced to perform excessive copying (application-level buffering), or else to render a single visual object over the course of many separate transactions, at the risk of inconsistent output.

Inevitability Mechanisms For each workload, we evaluate 9 library variants. The default (Baseline) variant is a standard TL2 implementation with no inevitability support. For programs with inevitable transactions its behavior is incorrect, but useful for comparison purposes. We compare against the global read lock (GRL); the global write lock both with and without a Transactional fence

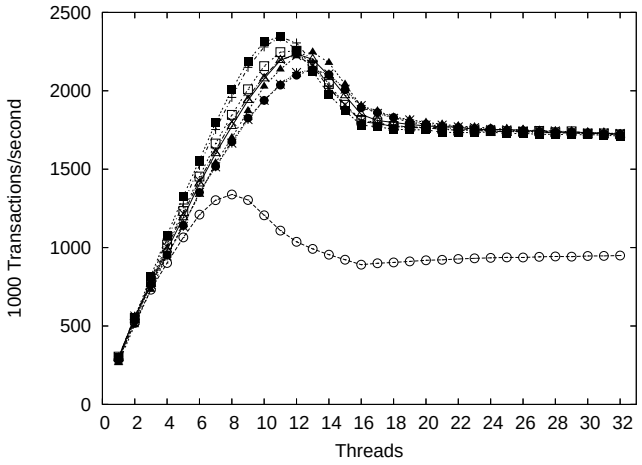

Figure 2. Disjoint Transactions, 20 accesses per transaction, 20% writes, using a TL2-like runtime. Inevitability is not used by any thread. Key appears in Figure 3.

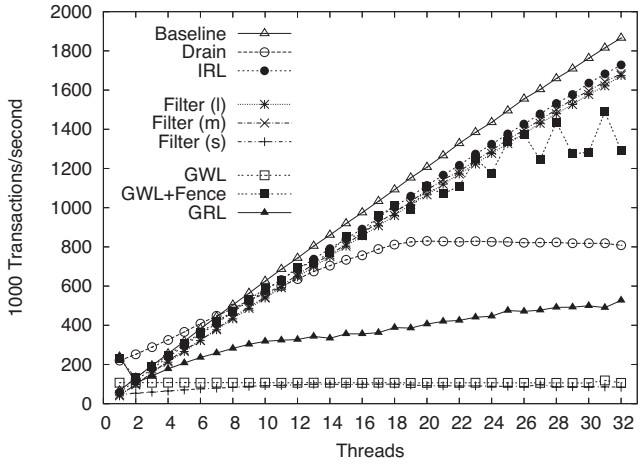

Figure 3. Shared reads, disjoint writes, 100 accesses per transaction, 20% writes, using a TL2-like runtime. Thread 0 runs inevitably at all times.

(GWL+Fence and GWL, respectively); the writer drain (Drain); inevitable read locks (IRL); and three Bloom filter mechanisms that differ in the size of the filter used and the number of hash functions. Filter (s) uses a single hash function and a 64-bit bloom filter. Filter (m) and Filter (l) both use a 4096-bit filter, with one and three hash functions, respectively.

5.1 Latency for Non-Inevitable Transactions

Our various inevitability algorithms differ in the amount of overhead they place on the critical path of non-inevitable transactions. We first consider the case when inevitability is not used by any transaction. Figure 2 compares overheads on a microbenchmark in which threads access disjoint regions of memory. Transactions are read-only with 33% probability, and otherwise perform 20% writes. Each writing transaction accesses 20 distinct locations (thus there are 4 writes), but the locations within each thread's disjoint memory region vary.

Our hope for linear scaling is not realized, due to inherent serialization among writing transactions in TL2. However, Drain serializes much earlier, due to contention for its status variable. The two atomic operations required to enter and exit the drain cause substantial bus traffic and cache misses; at any significant level of concurrency, each of these operations will result in a miss. Excluding Drain, our mechanisms introduce only modest overheads; they are all within 10% of baseline performance. This behavior matches our expectations, and confirms that supporting inevitability need not, in and of itself, be a significant source of latency.

5.2 Supporting Disjoint Writes

When all transactions' writes sets are disjoint, inevitability should ideally have no impact on scalability. In Figure 3, we show a benchmark in which every thread accesses 100 locations per transaction. Again, 33% of transactions are read-only, with the remaining transactions performing a mix of 20% writes to private buffers and 80% reads. However, all reads are to a single shared structure. Given the size of each transaction, we do not expect TL2's global timestamp to be a bottleneck, although we do expect a bottleneck in Drain.

Naturally, we expect linear scaling from the Baseline implementation, and we might hope that inevitability would speed up at least the inevitable thread without sacrificing scalability. However, our mechanisms incur various penalties that prevent this hope from being realized. With such large read sets, the absence of read instrumentation for Drain, GRL, and GWL+Fence results in substantial single-thread speedup. However, GWL forbids concurrent write commits entirely, resulting in flat performance. When the Transactional Fence is added to GWL, the periods when the inevitable transaction is blocked provide an opportunity for concurrent non-inevitable transactions to commit, raising performance. Furthermore, the GWL+TFence outperforms GRL by allowing concurrent transactions to progress up to their commit point during the inevitable transaction's execution.

IRL and the large and medium Filters perform slightly worse than Baseline, due to their additional memory ordering constraints. The workload is clearly sensitive to Filter parameters: the small Filter causes unnecessary aborts and performs dramatically worse than the other two. The distance between Baseline and these scalable mechanisms increases slightly as concurrency increases: we attribute this to the increased cache misses that result from read locks covering shared locations, and from cache misses during Filter accesses by non-inevitable transactions.

When we simulated the WBR memory fence with a lightweight-sync instruction, performance of the inevitable thread improved by 60% for the large and 53% for the small and medium Filters. If we take this to be a reasonable best-case overhead for a WBR fence, then the Filters should all outperform Baseline at one thread. The impact on non-inevitable transactions is negligible (less than 1% on average) since they issue only one WBR per writing transaction.

In summary, the mechanisms we expected to scale do so, though not quite as well as Baseline. Drain consistently outperforms quiescence, and affords better scalability than GWL, because it blocks non-inevitable transactions before they acquire locations. In contrast, GWL often detects conflicts between non-inevitable and inevitable transactions after acquisition, which forces the non-inevitable transaction to abort in order to prevent deadlock.

5.3 Workload Acceleration

We lastly consider the effectiveness of inevitability as an optimization. In the large body of worklist-style algorithms that have occasional conflicts between tasks, we posit that inevitability can improve performance. In particular, when there is no priority or fairness requirement between tasks, and when tasks do not synchronize with each other, then the decision to execute some tasks inevitably can improve throughput, so long as inevitable transactions execute more quickly than their non-inevitable counterparts.

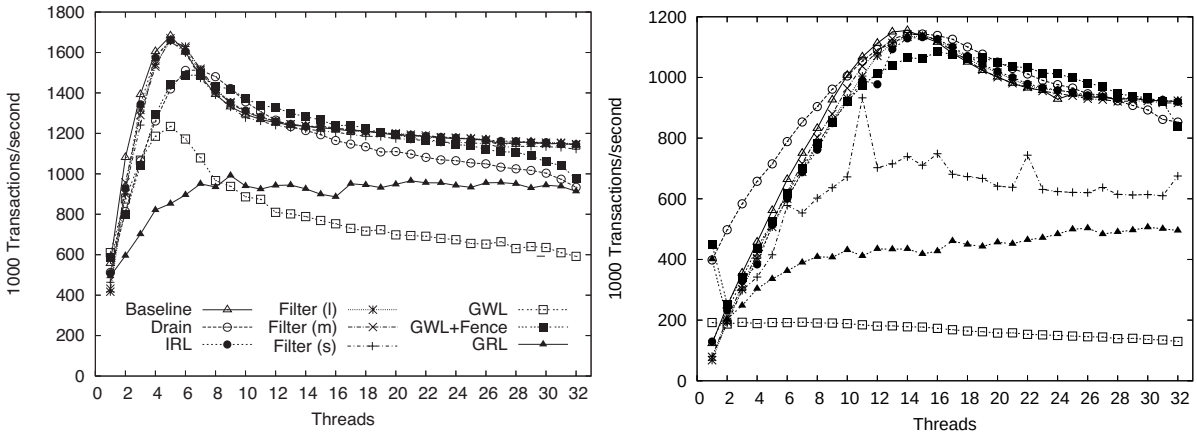

Figure 4. Scalable worklists, using a TL2-like runtime. Thread 0 runs inevitably at all times. Transactions on the left hand side are short, modeled with a 256-element hash table, while transactions on the right are larger, via a red-black tree with 1M elements.

To evaluate this claim, we concurrently execute an equal mix of insert, remove, and lookup instructions on a set, and measure the change in throughput when one thread executes all transactions inevitably. Figure 4 presents two such workloads. In the first, tasks access a 256-element hash table; in the second, a 1M-element red-black tree.

The HashTable's small transactions do not benefit from inevitability; the overhead of the drain and the TL2 timestamp dominate, resulting in both a limit on the scalability of an otherwise scalable benchmark, and a limit on the improvement afforded by inevitability. In the RBTree, however, transactions are large enough that the drain overhead does not dominate. Consequently, there is a modest but decreasing benefit to inevitability at lower thread levels, and not until 24 threads does inevitability increase latency.

Again, the GWL performs worse than GRL. As before, most of GRL's scalability is due to non-inevitable transactions completing while the inevitable transaction is blocked. In GWL, however, non-inevitable transactions can slow down an inevitable transaction, since they access metadata concurrently with the inevitable transaction. Since all non-inevitable transactions block in GRL, no such interference occurs. Adding the Transactional Fence to GWL eliminates this effect. However, as in GRL, the fence prevents fast-running inevitable transactions from running often enough to improve performance at high thread counts.

In additional experiments, we confirmed that this effect is even more pronounced in workloads with less inherent concurrency—e.g., with large transactions that frequently conflict. Here single-thread performance is paramount, and the use of inevitability to raise that performance can have a dramatic impact. On a RandomGraph [14] workload, where all transactions conflict and each transaction reads hundreds of locations, we observed over two orders of magnitude improvement at one thread; at higher thread levels, contention for the drain increased, eventually reducing performance to baseline levels at 16 threads. In a scalable LinkedList workload with large read sets, inevitability raised throughput on 1–32 threads to a constant level 50% higher than the peak performance without inevitability.

5.4 Dependence on Workload Characteristics

Our results, though preliminary, suggest that the best inevitability mechanism depends on the offered workload:

Library or system calls with unpredictable write sets: GRL is the only option in this case. It sacrifices most concurrency in the application.

Short inevitable transactions that are likely to conflict with non-inevitable transactions: Here GWL is attractive. It requires, however, that the read and write sets of precompiled functions be predicted, and it limits scalability if there are concurrent nonconflicting writers.

Long but rare inevitable transactions that call library code with unpredictable read sets: GWL+Fence should perform well in this case. The cost of the fence is offset by the improved performance of the inevitable transaction, and when the workload does not contain concurrent disjoint writers, the impact on scalability will be limited.

Long, frequent inevitable transactions that run with long non-inevitable transactions, and rarely conflict with them: Drain seems best for these. It allows calls to precompiled code with unpredictable read sets, but introduces a scalability bottleneck if there are many short-running nonconflicting writers.

Short, frequent inevitable transactions that run with short non-inevitable transactions, and rarely conflict with them: Both IRL and Filter should work well here. Both are well-suited to workloads with frequent inevitable transactions and concurrent, nonconflicting writer transactions. They both afford good scalability, but require that the read and write sets of library code be predicted. The choice of mechanism should depend on the timing of the call to `begin_inevitable`. If the call is made late, the expense of making existing reads inevitable should be lower for Filter, since the cost of a single memory fence can be amortized across all read locations, and there are no atomic operations. Filter should also be preferred if the inevitable transaction's read set overlaps with other transactions' reads, since IRL's read locks will cause non-inevitable transactions to incur cache misses.

6. Conclusions

In this paper we have presented several mechanisms to implement inevitable transactions. Using these mechanisms, programmers can easily incorporate I/O, system calls, and other irreversible operations into STM-based applications. While no single mechanism achieves both low contention-free overhead and high scalability at high levels of concurrency, these mechanisms make it practical to develop realistic transactional workloads. In particular, inevitability provides a simple and effective way to allow I/O in transactions and, contrary to conventional wisdom, need not sacrifice scalability

for workloads that have few write conflicts between inevitable and non-inevitable transactions.

The Bloom Filter mechanism underscores the importance of designing memory systems with lower write-before-read fence overhead. Even with current overheads, the Filter seems like best way to provide inevitability without sacrificing scalability. When speed within inevitable transactions is paramount, the Drain appears to be best: it gives high single-thread performance and low overhead when conflicts are infrequent, but introduces a scalability bottleneck. We suspect that it is most suitable as a performance optimization for workloads with low contention and no condition synchronization. Alternatively, GWL+Fence provides better scalability and comparable throughput for inevitable transactions with little impact on non-inevitable transactions, at the cost of delays at the point when a transaction becomes inevitable. Ideally, future STM systems might employ static analysis or profiling to identify the ideal inevitability mechanism for a given application.

References

[1] L. Baugh and C. Zilles. An Analysis of I/O and Syscalls in Critical Sections and Their Implications for Transactional Memory. In *Proc. of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[2] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[4] D. Dice and N. Shavit. Understanding Tradeoffs in Software Transactional Memory. In *Proc. of the 2007 Intl. Symp. on Code Generation and Optimization*, San Jose, CA, Mar. 2007.

[5] P. Felber, C. Fetzer, U. Müller, T. Riegel, M. Süßkraut, and H. Sturzrehm. Transactifying Applications using an Open Compiler Framework. In *Proc. of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[6] K. Fraser. Practical Lock-Freedom. Technical Report UCAM-CL-TR-579, Cambridge Univ. Computer Laboratory, Feb. 2004.

[7] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabju, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proc. of the 31st Annual Intl. Symp. on Computer Architecture*, page 102. IEEE Computer Society, June 2004.

[8] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable Memory Transactions. In *Proc. of the 10th ACM SIGPLAN 2006 Symp. on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[9] T. Harris, M. Plesko, A. Shinar, and D. Tarditi. Optimizing Memory Transactions. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[10] M. P. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd Annual ACM Symp. on Principles of Distributed Computing*, Boston, MA, July 2003.

[11] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Annual Intl. Symp. on Computer Architecture*, San Diego, CA, May 1993. ACM Press.

[12] O. S. Hofmann, D. E. Porter, C. J. Rossbach, H. E. Ramadan, and E. Witchel. Solving Difficult HTM Problems Without Difficult Hardware. In *Proc. of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[13] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. A Scalable Transactional Memory Allocator. In *Proc. of the 2006 Intl. Symp. on Memory Management*, Ottawa, ON, Canada, June 2006.

[14] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[15] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Nonblocking Software Transactional Memory. In *Proc. of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[16] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proc. of the 34th Intl. Symp. on Computer Architecture*, San Diego, CA, June 2007.

[17] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. Hosking, R. Hudson, E. Moss, B. Saha, and T. Shpeisman. Open Nesting in Software Transactional Memory. In *Proc. of the 12th ACM SIGPLAN 2007 Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[18] M. Olszewski, J. Cutler, and J. G. Steffan. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. In *PACT '07: Proc. of the 16th Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT 2007)*, Brasov, Romania, Sept. 2007.

[19] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. In *Proc. of the 11th ACM SIGPLAN 2006 Symp. on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[20] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proc. of the 24th Annual ACM Symp. on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[21] N. Shavit and D. Touitou. Software Transactional Memory. In *Proc. of the 14th Annual ACM Symp. on Principles of Distributed Computing*, Ottawa, ON, Canada, Aug. 1995.

[22] Y. Smaragdakis, A. Kay, R. Behrends, and M. Young. Transactions with Isolation and Cooperation. In *OOPSLA '07: Proc. of the 22nd Annual ACM SIGPLAN Conf. on Object Oriented Programming Systems and Applications*, Montreal, Quebec, Canada, Oct. 2007.

[23] M. F. Spear, V. J. Marathe, L. Dalessandro, and M. L. Scott. Privatization Techniques for Software Transactional Memory. Technical Report TR 915, Department of Computer Science, Univ. of Rochester, Feb. 2007.

[24] M. F. Spear, A. Shriraman, L. Dalessandro, S. Dwarkadas, and M. L. Scott. Nonblocking Transactions Without Indirection Using Alert-on-Update. In *Proc. of the 19th ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

[25] F. Tabba, C. Wang, J. R. Goodman, and M. Moir. NZTM: Nonblocking Zero-Indirection Transactional Memory. In *Proc. of the 2nd ACM SIGPLAN Workshop on Transactional Computing*, Portland, OR, Aug. 2007.

[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 *Proc. of the 2007 Intl. Symp. on Code Generation and Optimization*, San Jose, CA, Mar. 2007.

[27] V. Ying, C. Wang, Y. Wu, and X. Jiang. Dynamic Binary Translation and Optimization of Legacy Library Code in an STM Compilation Environment. In *Proc. of the 2006 Workshop on Binary Instrumentation and Applications*, San Jose, CA, Oct. 2006.