Ordering-Based Semantics for Software Transactional Memory*

Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott

Technical Report #938

Department of Computer Science, University of Rochester
{spear, luked, vmarathe, scott}@cs.rochester.edu

August 2008

Abstract

It has been widely suggested that memory transactions should behave as if they acquired and released a single global lock. Unfortunately, this behavior can be expensive to achieve, particularly when—as in the natural *publication/privatization* idiom—the same data are accessed both transactionally and nontransactionally. To avoid overhead, we propose *selective strict serializability* (SSS) semantics, in which transactions have a global total order, but nontransactional accesses are globally ordered only with respect to explicitly marked transactions. Our definition of SSS formally characterizes the permissible behaviors of an STM system without recourse to locks. If all transactions are marked, then SSS, single-lock semantics, and database-style strict serializability are equivalent.

We evaluate several SSS implementations in the context of a TL2-like STM system. We also evaluate a weaker model, *selective flow serializability* (SFS), which is similar in motivation to the *asymmetric lock atomicity* of Menon et al. We argue that ordering-based semantics are conceptually preferable to lock-based semantics, and just as efficient.

Keywords: Software Transactional Memory, Serializability, Atomicity, Semantics

1 Introduction

With the proliferation of multicore processors, there is widespread recognition that traditional lock-based synchronization is too complex for "mainstream" parallel programming. *Transactional memory* attempts to address this complexity by borrowing the highly successful notion of transactions from database systems.

Transactions constrain the ways in which thread histories may legally appear to interleave. In a break from the database world, however, memory transactions are generally expected to coexist with nontransactional memory accesses. This coexistence complicates the task of specifying, for every read in the program, which values may be returned.

Many possible semantics for TM have been proposed, including *strong and weak isolation* (also known as strong and weak atomicity) [1, 17], *single lock atomicity* (SLA) [7], and approaches based on language memory models [4], linearizability [5, 15], and operational semantics [12]. Of these, SLA has received the most attention. It specifies that transactions behave as if they acquired a single global mutual exclusion lock. Unfortunately, as several groups have noted [7, 11, 18], SLA requires behavior that can be expensive to enforce, particularly when a thread *privatizes* shared data (rendering it logically inaccessible to other threads), works on it for a while (ideally without incurring transactional overheads), and then *publishes* it again [10, 11, 18].

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

Most software TM (STM) implementations today do not correctly accommodate privatization and publication, and forcing them to do so—at the boundaries of every transaction—would impose significant costs. In an attempt to reduce those costs, Menon et al. [11] have proposed a series of semantics that relax the requirement for serialization among transactions. These semantics are described in terms of locking protocols significantly more complex than SLA. While we appreciate the motivation behind this relaxation, we argue, with Luchangco [8], that locks are the wrong way to formalize transactions. We prefer the more abstract approach of language-level memory models [2, 4, 9], which directly specify permissible access orderings. We also argue that if one wishes to reduce the cost of SLA, it makes more sense to relax the globally visible ordering between nontransactional and transactional accesses of a given thread, rather than the ordering between transactions.

We argue that SLA is equivalent to the traditional database ordering condition known as *strict serializability* (SS). As a candidate semantics for STM, we suggest transactions with *selective strict serializability* (SSS), in which nontransactional accesses are globally ordered with respect to a subset of their thread's transactions. Whether this subset is all, none, or some explicitly or implicitly identified set in between, a single formal framework suffices to explain program behavior. We also propose a slightly weaker semantics, *selective flow serializability* (SFS), that orders nontransactional accesses with respect to subsequent transactions in other threads only in the presence of a forward dataflow dependence that could constitute publication.

Like Menon et al., we take the position that races between transactional and nontransactional code are program bugs, but that (as in Java) the behavior of buggy programs should be constrained to avoid "out of thin air" reads. SSS and SFS allow the programmer or compiler to eliminate races by labeling a minimal set of transactions, while still constraining behavior if the labeling is incorrect.

After a more detailed discussion of background in Section 2, we formalize our ordering-based semantics in Section 3. To the best of our knowledge, this is the first attempt to formalize privatization and publication safety without recourse to locks. We discuss implementation options in Section 4, in the context of an STM system patterned on TL2 [3]. We compare the performance of these options experimentally in Section 5, and conclude in Section 6.

2 Background

Arguably the most intuitive semantics for transactions would build on sequential consistency, providing a global total order, consistent with program order, for both transactions and nontransactional accesses. Unfortunately, most researchers already consider sequential consistency too expensive to implement, even without transactions. Among other things, it precludes standard compiler optimizations like write reordering. In light of this expense, sequentially consistent transactions appear to be a non-starter.

Several researchers have argued instead for what Blundell et al. [1] call *strong atomicity*, otherwise known as *strong isolation*. These semantics drop the requirement that a thread's nontransactional memory accesses be seen by other threads in program order. Strong isolation still requires, however, that nontransactional accesses serialize with respect to transactions; that is, that each nontransactional reference appears to happen *between* transactions, and that all threads agree as to which transactions it appears between.

Unfortunately, strong isolation still suffers from several significant costs, which make it unappealing, at least for software implementation:

Granularity: In a high-level programming language, it is not immediately clear what constitutes an individual memory access. Is it acceptable, for example, for a transaction to intervene between the read and write that underlie `x++` on a load-store machine? How about x = 0x300000003, where `x` is a `long long` variable but hardware does not provide an atomic 64-bit store?

```
// initialize node
atomic {
  PQ.insert(node)
}
```

```
atomic {
  node = PQ.extract_min()
}
// use node privately
```

Figure 1: Examples of publication (left) and privatization (right). More obscure examples, involving antidependences or even empty transactions, are also possible [11].

Instrumentation: It is generally agreed by researchers in the field that speculative execution of software transactions in the face of concurrent nontransactional accesses will require the latter to inspect and (in the case of writes) modify transactional metadata. Particularly in programs that make extensive nontransactional use of data that are temporarily private, this instrumentation can have a major performance impact.

Optimization obstruction: Nontransactional accesses cannot safely be reordered if they refer to locations that may also be accessed by transactions. The sequence `x = 1; y = 2` admits the possibility that a concurrent transaction will see `x == 1 && y != 2`. If the compiler is unable to prove that no such transaction (or series of transactions) can exist, it must, (a) treat `x` and `y` as volatile variables, access them in the order specified by the program, and insert a write-write memory barrier between them, or (b) arrange for the assignments to execute as a single atomic unit.[1]

In light of these costs, most STM systems provide some form of *weak isolation*, in which nontransactional accesses do not serialize with transactions. As several groups have noted [4, 11, 12, 18], the exact meaning of weak isolation is open to interpretation. Perhaps the most popular interpretation is *single lock atomicity* (SLA) [7, p. 20], which states that transactions behave as if they acquired a global mutex lock. Unfortunately, even these semantics have nontrivial cost, and are unsupported by most TM implementations, particularly for programs that publish or privatize data.

Publication (Figure 1) occurs when a thread initializes or otherwise modifies a data structure that is logically private, and then modifies shared state to make the structure accessible to other threads. *Privatization* is the reverse: a thread's modification of shared state makes some structure logically private. The appeal of privatization is the possibility that temporarily private data might be accessed without transactional overhead.

Traditionally, when a thread releases a mutual exclusion lock, all prior accesses by the thread are guaranteed to have occurred from the point of view of every other thread. Similarly, when a thread acquires a lock, all subsequent accesses by the thread are guaranteed not to have occurred. These facts suggest that the natural implementation of SLA would be *publication-* and *privatization-safe*.

The Privatization Problem. In previous work [18], we identified two dimensions of the privatization problem. In the *delayed cleanup problem* (also described by Larus and Rajwar [7, pp. 22–23]), a thread that privatizes shared data may fail to see prior updates by a transaction that has committed but has not yet written its "redo log" back to main memory.[2] In the *doomed transaction problem* (also described by Wang et al. [19, pp. 6–7]), post-privatization writes may cause a doomed transaction that has not yet aborted to see an inconsistent view of memory, allowing it fall into an infinite loop, suffer an exception, or (in the absence of run-time sandboxing) perform erroneous actions that cannot be undone.

[1] Note that while strong isolation is sometimes informally equated with making every nontransactional access execute as if it were an isolated transaction, this characterization is problematic: it would force a global total order not only between these isolated transactions and "real" programmer-specified transactions, but *among* the isolated transactions. The end result would be equivalent to the "sequentially consistent transactions" alluded to above.

[2] Conversely, in an STM system based on undo logs, a privatizing thread may see erroneous updates made (temporarily) by a transaction that has aborted but not yet cleaned up. As observed by Menon et al. [11], such reads appear to be fundamentally incompatible with the prohibition against "out of thin air" reads. We therefore assume a redo log in the remainder of this paper.

```
// initially x == 0 and x_is_public == false
T1:                          T2:                          T3:
                                                          e: j = 0
                                                          F: atomic {
                                                              t = prefetch(x)
a: x = 1
B: atomic {                  c: i = 0
    x_is_public = true       D: atomic {
}                                if (x_is_public) {           if (x_is_public) {
                                     i = x                        j = t
                                 }                            }
                             }                            }
```

Figure 2: Publication (left) in parallel with a safe (middle) or unsafe (right) use of published data. Vertical spacing is meant to suggest interleaving of operations across threads. Both `D` and `F` are assumed to serialize after `B`.

The Publication Problem. One might not initially expect a publication problem: private accesses are assumed to occur before publication, and there is no notion of cleanup for nontransactional code. Menon et al. show, however, that problems can arise if the programmer or compiler prefetches data before it is actually published (Figure 2).

Relaxing SLA. Under SLA, straightforward solutions to the privatization and publication problems [11, 18] require transactions to begin and clean up in serialization order. While heuristic optimizations may relax this requirement in some cases [10], it seems clear that the general case will remain expensive for STM.

To minimize expense, Menon et al. [11] propose to relax the ordering among transactions. Of their three successively weaker candidate semantics, *asymmetric lock atomicity* (ALA) seems most likely to enable performance improvement without precluding standard compiler optimizations. ALA transactions behave as if (1) there is a separate reader-writer lock for every datum, (2) read locks are acquired (presciently) at the beginning of the transaction, and (3) write locks are acquired immediately prior to writing. The asymmetry of reads and writes reflects the fact that (a) in most TM systems it is much easier for a reader to detect a conflict with a previous writer than vice versa, and (b) in most programs publication can be assumed to require a write in one transaction followed by a read in another.

In our view, ALA and similar proposals suffer from three important problems. First, they explain transaction behavior in terms of a nontrivial fine-grain locking protocol—something that transactions were intended to eliminate! Second, they give up one of the key contributors to the success of database transactions—namely serializability. Third, they impose significant overheads on transactions that do serialize, even in the absence of publication and privatization.

3 Ordering-Based TM Semantics

Our proposed alternative to lock-based semantics assumes a programming model in which every thread i has a memory access history H^i in which certain maximal contiguous strings of accesses are identified as (outermost) transactions. We use $\mathcal{T}$ to denote the set of all transactions. We do not consider open nesting here, nor do we permit overlapping but non-nested transactions. Moreover, from the programmer's point of view, transactions are simply atomic: there is no notion of speculation or of committing and aborting. A typical implementation will need to ensure that abortive attempts to execute a transaction are invisible; among other things, this will require that such attempts retain a consistent view of memory [5].

The goal of a semantics for TM is to constrain the ways in which thread histories may legally interleave to create a global history. In keeping with the database literature and with memory models for programming languages like Java [9] and C++ [2], we believe that the appropriate way to specify these constraints is not by reduction to locks, but rather by specifying a partial order on program operations that determines which writes may be "seen" by which reads.

The standard database ordering criterion is *serializability* [13], which requires that the result of executing a set of transactions be equivalent to (contain the same operations and results as) some execution in which the transactions take place one at a time, and any transactions executed by the same thread take place in program order. *Strict serializability* imposes the additional requirement that if transaction A completes before B starts in the actual execution, then A must occur before B in the equivalent serial execution. The intent of this definition is that if external (nontransactional) operations allow one to tell that A precedes B, then A must serialize before B. For transactional memory, it seems reasonable to equate external operations with nontransactional memory accesses, and to insist that such accesses occur between the transactions of their respective threads, in program order.

More formally, we define the following strict (asymmetric, irreflexive) ordering relations:

Program order, $<_p$, is a union of disjoint total orders, one per thread. We say $a <_p b$ iff a and b are executed by the same thread, and a comes before b in the natural sequential order of the language in which the program is written. Because transactions do not overlap, if transactions A and B are executed by the same thread, we necessarily have either $\forall a \in A, b \in B : a <_p b$ or $\forall a \in A, b \in B : b <_p a$. For convenience, we will sometimes say $A <_p B$ or $B <_p A$. For $a \notin B$, we may even say $a <_p B$ or $B <_p a$.

Transaction order, $<_t$, is a total order on all transactions, across all threads. It is consistent with program order. That is, $A <_p B \implies A <_t B$. For convenience, if $a \in A$, $b \in B$, and $A <_t B$, we will sometimes say $a <_t b$.

Strict serial order, $<_{ss}$, is a partial order on memory accesses. It is consistent with transaction order. It also orders nontransactional accesses with respect to preceding and following transactions of the same thread. Formally, $\forall$ accesses $a, c \in H$, we say $a <_{ss} c$ iff at least one of the following holds: (1) $a <_t c$; (2) $\exists\, A \in \mathcal{T}$: $(a \in A \ \wedge\ A <_p c)$; (3) $\exists\, C \in \mathcal{T}$: $(a <_p C \ \wedge\ c \in C)$; (4) $\exists$ access $b \in H : a <_{ss} b <_{ss} c$. Note that this definition does *not* relate accesses performed by a given thread between transactions.

An execution with program order $<_p$ is said to be strictly serializable if there exists a transaction order $<_t$ that together with $<_p$ induces a strict serial order $<_{ss}$ that permits all the values returned by reads in the execution, as defined in the following subsection. A TM implementation is said to be strictly serializable if all of its executions are strictly serializable.

Values read

To avoid the need for special cases, we assume that each thread history H^i begins with an initial transaction T_0^i, that T_0^0 writes values to all statically initialized data, and that $\forall i > 0 : T_0^0 <_{ss} T_0^i$.

We say a memory access b *intervenes* between a and c if $a <_p b <_p c$ or $a <_{ss} b <_{ss} c$. Read r is then permitted to return the value written by write w iff (1) r and w are incomparable under both program and strict serial order or (2) $w <_p r \ \vee\ w <_{ss} r$ and there is no intervening write between w and r.

We argue that strict serializability, as defined above, is equivalent to SLA. If an execution is strictly serializable, then it is equivalent by definition to some execution in which transactions occur in a serial order consistent with program order and with nontransactional operations. This serial order is trivially equivalent to an execution in which transactions acquire and release a global lock. Conversely, if transactions acquire and release a global lock, they are guaranteed to execute one at a time, in an order consistent with program

order. Moreover SLA and strict serial order, as defined above, impose identical constraints on the ordering of nontransactional accesses relative to transactions.

3.1 Selective Strictness

We say that a TM implementation is *publication and privatization safe* with respect to a given ordering semantics if all its executions adhere to those semantics, even in programs containing data that are sometimes accessed inside transactions and sometimes outside transactions.[3]

Unfortunately, most existing TM implementations are not publication and privatization safe with respect to strict serializability, and (as we show in Section 5), modifying them to be so would incur nontrivial costs. It is not yet clear whether these costs will be considered an acceptable price to pay for simple semantics. It therefore seems prudent to consider weaker semantics with cheaper implementations. Menon et al. [11] approach this task by defining more complex locking protocols that relax the serialization of transactions. In contrast, we propose a weaker ordering, *selective strict serializability*, that retains the serialization of transactions, but relaxes the ordering of nontransactional accesses with respect to transactions. Specifically, we assume a set of *acquiring* (privatizing) transactions $\mathcal{A} \subseteq \mathcal{T}$ and a set of *releasing* (publishing) transactions $\mathcal{R} \subseteq \mathcal{T}$. $\mathcal{A}$ and $\mathcal{R}$ are not required to be disjoint, nor are they necessarily proper subsets of $\mathcal{T}$: it is permissible for all transactions, or none, to be identified as acquiring and/or releasing.

Selective strict serial order, $<_{sss}$, is a partial order on memory accesses. Like strict serial order, it is consistent with transaction order. Unlike strict serial order, it orders nontransactional accesses only with respect to preceding acquiring transactions and subsequent releasing transactions of the same thread (and, transitively, transactions with which those are ordered). Formally, $\forall$ accesses $a, c \in H$, we say $a <_{sss} c$ iff at least one of the following holds: (1) $a <_t c$; (2) $\exists\ A \in \mathcal{A} : (a \in A\ \wedge\ A <_p c)$; (3) $\exists\ C \in \mathcal{R} : (a <_p C\ \wedge\ c \in C)$; (4) $\exists$ access $b \in H : a <_{sss} b <_{sss} c$.

3.2 Asymmetric Flow

Strict serializability, whether "always on" or selective, shares a problem with the DLA semantics of Menon et al. [11]: it requires the useless but expensive guarantee illustrated in Figure 3. Specifically, $a <_p B <_t F \implies a <_{ss} F$, even if B and F are ordered only by antidependence. We can permit significantly cheaper implementations if we define a weaker ordering, *selective flow serializability*, that requires nontransactional-to-transactional ordering only when transactions are related by a true (flow) dependence:

Flow order, $<_f \subset <_t$, is a partial order on transactions. We say that $A <_f C$ if there exists a transaction B and a location l such that $A <_t B$, A writes l, B reads l, A and B occur in different threads, there is no intervening write of l, and $B = C\ \vee\ B <_t C$.

Selective flow serial order, $<_{sfs} \subset <_{sss}$, is a partial order on memory accesses. It is consistent with transaction order. It does not order nontransactional accesses with respect to a subsequent releasing transaction B, but rather with respect to transactions that have a flow dependence on B. Formally, $\forall$ accesses $a, c \in H$, we say $a <_{sfs} c$ iff at least one of the following holds: (1) $a <_t c$; (2) $\exists\ A \in \mathcal{A} : (a \in A\ \wedge\ A <_p c)$; (3) $\exists\ B \in \mathcal{R}, C \in \mathcal{T} : (a <_p B <_f C\ \wedge\ c \in C)$; (4) $\exists$ access $b \in H : a <_{sfs} b <_{sfs} c$.

The sets of values that reads are permitted to return under selective strict and flow serial orders are defined the same as under strict serial order, but with $<_{sss}$ and $<_{sfs}$, respectively, substituted for $<_{ss}$. These

[3] An implementation that was not publication and privatization safe might still be correct if used in a system that statically partitioned private and transactionally shared data. In such a system, strict serializability might still be a stronger semantics than "ordinary" serializability, if threads could use system calls or lock-based or nonblocking data structures to interact outside transactions.

```
// initially x == 0, x_is_public == false, and T3_used_x == false
T1:                          T2:                              T3:
                                                              e: j = 0
                                                              F: atomic {
                                                                  t = prefetch(x)
a: x = 1
B: atomic {                  c: i = 0
    x_is_public = true       D: atomic {
    k = T3_used_x                if (x_is_public) {
}                                    i = x                        T3_used_x = true
                                 }                                j = t
                             }                                }
```

Figure 3: "Publication" by antidependence. If `B` is a releasing transaction, selective strict serializability guarantees not only that the write of `x` is visible to `D` (`i == 1`, which makes sense), but also that if `k == false`, then `B` must have serialized before `F`, and thus `j` must equal `1` as well. Unfortunately, it is difficult for an STM implementation to notice a conflict between `A` and `F` if the former commits before the latter writes `T3_used_x`, and undesirable to outlaw the prefetch of `x`.

induce corresponding definitions of SSS and SFS executions and TM implementations. In comparison to ALA, SFS is not defined in terms of locks, and does not force ordering between nontransactional accesses and unrelated transactions (Figure 4).

Like ALA, SFS can lead to apparent temporal loops in racy programs. In Figure 3, for example, both semantics allow `k == false` and `j == 0`, even if `B` is a releasing transaction. Naively, this output would seem to suggest that $a < B < F < a$. ALA breaks the loop by saying that B and F are not really ordered. SFS breaks the loop by saying that a and F are not really ordered. Which of these is preferable is perhaps a matter of taste.

4 Implementing SSS and SFS Systems

Both the doomed transaction problem and the undo log variant of the delayed cleanup problem (described in footnote 2) involve abortive attempts to execute transactions. Since these attempts play no role in the (user-level) semantics of Section 3, we need to extend our formalism. For discussion of implementation-level issues, we extend thread histories (as in our previous work [15]) to include begin, commit, and abort operations. None of these takes an argument. Begin and abort return no value. Commit returns a Boolean indication of success. Each thread history is assumed to be of the form ((read | write)* begin (read | write)* (commit | abort))* (read | write)* (for simplicity, we assume that nested transactions are subsumed into their parent). A transaction comprises the sequence of operations from begin through the first subsequent commit or abort in program order. A transaction is said to *succeed* iff it ends with a commit that returns true.

With this extended definition, for $<_g \in \{<_{ss}, <_{sss}, <_{sfs}\}$, a read r is permitted to return the value written by a write w iff (1) w does not belong to an unsuccessful transaction, and r and w are incomparable under both $<_p$ and $<_g$; (2) w does not belong to an unsuccessful transaction, $w <_p r$ or $w <_g r$, and there is no intervening write between w and r; or (3) w and r belong to the same transaction, $w <_p r$, and there is no intervening write between w and r. A memory access b intervenes between a and c if $a <_p b <_p c$ or $a <_g b <_g c$ and (a) b and c belong to the same transaction, or (b) neither a nor b belongs to an unsuccessful transaction. These rules are roughly equivalent to those of Guerraoui and Kapałka [5], but simplified to merge request and reply events and to assume that histories are complete (i.e., that every transaction eventually commits or aborts), and extended to accommodate nontransactional accesses. In

```
// initially x == 0, y == 0, and x_is_public == false
T1:                              T2:                              T3:
                                                                  e: j = 0
                                                                  F: atomic {
                                 c: i = 0                              ...
                                 D: atomic {
                                     t = prefetch(x)
a: x = 1
B: atomic {
    x_is_public = true
}                                    if (x_is_public) {
                                         i = y = t
                                     }
                                 }                                    j = y
                                                                  }
```

Figure 4: Unnecessary ordering in ALA. When `B` commits and `D` reads `x_is_public`, ALA forces `D` to abort (as it should). When `D` commits and `F` reads `y`, the implementation will likewise force `F` to abort, though it could reasonably serialize after `D`. With SFS, the programmer would presumably mark `B` but not `D` as a releasing transaction, and `F` would not have to abort.

particular, we maintain their requirement that transactions appear to occur in serial order ($<_t$), and that writes in unsuccessful transactions are never externally visible.

We assume that every correct STM implementation suggests, for every execution, a (partial or total) *natural order* $<_n$ on transactions that is consistent with some $<_t$ that (together with $<_p$) can explain the execution's reads. For the implementation to ensure SSS semantics, it must provide publication and privatization safety only around selected releasing and acquiring transactions, respectively. To ensure SFS semantics, it must provide the same privatization safety, but need only provide publication safety beyond selected flow-ordered transactions.

4.1 Preventing the Doomed Transaction Problem

The doomed transaction problem occurs when an STM implementation admits an execution containing a failed transaction D, a nontransactional write w, an acquiring transaction $A <_p w$, and a natural transaction order $<_n$ such that any $<_t$ consistent with $<_n$, when combined with $<_p$, induces a global (SS, SSS, SFS) order $<_g$ that fails to explain a value read in D—specifically, when there are dependences that force $D <_t A$, but there exists a read $r \in D$ that returns the value written by w, despite the fact that $D <_g A <_g w$.

In managed code, it appears possible to use run-time sandboxing to contain any erroneous behavior in doomed transactions [6, 11]. For unmanaged code, or as an alternative for managed environments, we present three mechanisms to avoid the inconsistencies that give rise to the problem in the first place.

Quiescence A *transactional fence* [18] blocks the caller until all active transactions have committed or aborted *and cleaned up*. This means that a fence f participates in the natural order $<_n$ on transactions and in program order $<_p$ for its thread. Since $D <_t A <_t f <_p w$, and f waits for D to clean up, we are guaranteed that the implementation respects $D <_g w$.

Polling Polling for the presence of privatizers can tell a transaction when it needs to check to see if it is doomed. This mechanism requires every privatizing transaction to increment a global counter that is polled by every transaction, on every read of shared data. When a transaction reads a new value of the counter,

```
TxBegin(desc)                              TxRead(&addr, &val, desc)
  ...                                        ... // read value consistently
  desc->priv_cache = priv_count              t = priv_count
                                             if (t != desc->priv_cache)
Acquire()                                      validate()
  fai(&priv_count)                             desc->priv_cache = t
```

Figure 5: Polling to detect doomed transactions.

it validates its read set and, if doomed, aborts before it can see an inconsistency caused by a private write. Pseudocode for this mechanism appears in Figure 5.

Like a transactional fence, the increment c of `priv_count` participates in $<_n$. Suppose D contains a read r that sees (incorrectly) a value written by w, with $D <_t A <_t c <_p w$. Since c increments `priv_count` and D reads `priv_count` prior to every read, D must abort before r, a contradiction.

Timestamp Polling In a timestamp-based STM like TL2 [3], every writer increments a global timestamp. If all transactions are writers (and hence all update the global timestamp), polling this timestamp prevents the doomed transaction problem, using the same argument as above. For privatization by antidependence, it suffices to observe that while A may not increment the global timestamp, A is ordered after some other transaction W ($W <_n A$) that committed a write in order for A's privatization to succeed. If D is doomed because of the privatization (and still active), it must read a value written by W. W's increment of the global timestamp is sufficient to force a polling-based abort in D prior to any use of an inconsistent value.

4.2 Preventing the Delayed Cleanup Problem

The delayed cleanup problem occurs when an STM implementation admits an execution containing a successful transaction D whose cleanup is delayed,[4] a nontransactional read r, an acquiring transaction $A <_p r$, and a natural transaction order $<_n$ such that any $<_t$ consistent with $<_n$, when combined with $<_p$, induces a global (SS, SSS, SFS) order $<_g$ that fails to explain the value read by r—specifically, when there are dependences that force $D <_t A$, but r returns the value from some write w despite an intervening write $w' \in D$. We propose two mechanisms to avoid this problem.

Quiescence As before, let A be immediately followed by a transactional fence f, and let D commit before A. Since $D <_t A <_t f <_p r$, and f waits for D to clean up, we are guaranteed that the implementation respects $D <_g r$.

Optimized Commit Linearization Menon et al. describe a commit linearization mechanism in which committing transactions acquire a unique linearization number from a global counter. Unfortunately, forcing read-only transactions to modify shared data has a serious performance cost. To avoid this cost, we propose an alternative implementation of commit linearization in Figure 6. The implementation is inspired by the classic ticket lock: writer transactions increment the global timestamp, clean up (in order), and then increment a second `cleanups` counter. Readers read the timestamp and then wait for `cleanups` to catch up.

We argue that this mechanism is privatization safe with respect to (even non-selective) strict serializability. If writer D precedes writer A in natural order but has yet to clean up, then D will not yet have updated the `cleanups` counter, and A's `TxCommit` operation will wait for it. Any subsequent read in A's thread can be guaranteed that D has completed.

[4] As noted in footnote 2, undo log-based STMs have an analogous problem, which we ignore here.

```
TxBegin(desc)
  ...
  start = timestamp
  while (cleanups < start)
    yield()
```

```
TxCommit(desc) // not read only
  acquire_locks()
  my_timestamp = fai(timestamp)
  if (validate())
    // copy values to shared memory
  else
    must_restart = true
  release_locks()
  while (cleanups != (my_timestamp - 1))
    yield()
  cleanups = my_timestamp
```

Figure 6: An implementation of commit linearization in which read-only transactions do not update shared metadata.

```
TxCommit(desc) // not read only
  commit_fence[my_slot]++
  acquire_locks()
  my_timestamp = fai(timestamp)
  if (validate())
    // copy values to shared memory
  else
    must_restart = true
  release_locks()
  commit_fence[my_slot]++
```

```
Acquire()
  num = commit_fence.size
  for (i = 0 .. num)
    local[i] = commit_fence[i]
  for (i = 0 .. num)
    if (local[i].is_odd())
      while (commit_fence[i] == local[i])
        yield();
```

Figure 7: The commit fence.

Suppose that reader A privatizes by antidependence. If D increments the global timestamp before A begins, A must wait in `TxBegin` for D to clean up, avoiding the problem. If D is still active when A begins, there must exist some other writer W ($W <_n A$) that committed a write in order for A's privatization to succeed. Clearly $D \neq W$, or else D's write of the datum in the antidependence would have forced A to abort. Moreover, since the program is race free, then $D <_n W$. For D to still be active when A begins we must have W still active when A begins, a contradiction, since W writes a datum that A reads, and A does not abort.

Commit Fence Our commit fence mechanism combines the best features of the transactional fence and commit linearization. As in the transactional fence, there is no single global variable that is accessed by all committing writer transactions. As in commit linearization, only committing transactions can cause a privatizer to delay. Pseudocode for the mechanism appears in Figure 7.

The commit fence ensures that any transaction sets an indicator before acquiring locks, and unsets the indicator after releasing locks. At its acquire fence, a privatizing transaction samples all transaction's indicators, and waits until it observes every indicator in an unset state. This commit fence c provides privatization safety as above: if D accesses data privatized by A, and if $D <_t A$, then D must update the commit fence before A completes its commit sequence. Since $A <_p c$, and c observes D's in-flight modifications, c will not return until D completes, and thus $D <_g c$.

Unlike the full transactional fence, this mechanism does not prevent the doomed transaction problem. Like the transactional fence, it can cause an acquirer A to wait on a committing, nonconflicting transaction B even when $A <_t B$. However, as in commit linearization, A will only block for committing transactions, never for in-flight transactions.

```
TxBegin(desc)
  ...
  desc->pub_cache = pub_count

TxRead(&addr, &val, desc)
  ... // read value consistently
  if (pub_count != desc->pub_cache)
    abort()
```

```
TxCommit(desc)
  ... // acquire locks, validate
  if (pub_count != desc->pub_cache)
    abort()
  ...

Release()
  fai(&pub_count)
```

Figure 8: Polling to detect publication.

```
TxStart(desc)
  ...
  desc->start = timestamp;

TxCommit(desc)
  ... // acquire locks
  endtime = get_timestamp();
  if (validate())
    // copy values to shared memory
    foreach (lock in writeset)
      lock.releaseAtVersion(endtime)
    ...
  ...
```

```
TxRead(addr* addr, addr* val, desc)
  // addr not in write set
  orec o = get_orec(addr);
  if (o.locked ||
      o.timestamp > desc->start)
    abort()
  ... // read value
  if (o != get_orec(addr))
    abort()
  ... // log orec, return value
```

Figure 9: TL2-style timestamps.

4.3 Preventing Publication Errors

Under SSS, publication safety can be expressed as the condition that if $w <_p R <_t T$, where $R \in \mathcal{R}$, then $w <_{sss} T$, even if T reads the datum written by w. We propose two implementations of a release mechanism that guarantee this condition.

Quiescence Placing a transactional fence between a nontransactional access and a subsequent publishing transaction prevents the publication problem. Suppose $w <_p R$, where w is a nontransactional write of datum d and $R \in \mathcal{R}$. The publication problem manifests when some transaction T prefetches d before w writes it, but $R <_n T$. If T begins before w, a fence between w and R forces T to complete before R begins, so $R \not<_n T$.

Polling Polling may also be used, as shown in Figure 8, to prevent the publication problem. Instead of having a publisher wait for all active transactions to complete, each active transaction T checks at each read (and at `TxCommit`) to see whether a release operation l has occurred since T began execution. If $l <_p R <_n T$ and T is successful, we are guaranteed that T was not active at the time l occurred, and T could not have prefetched any published datum.

4.4 Preventing Flow Publication Errors

Flow-serializable publication safety requires only that if $w <_p R <_f T$, where $R \in \mathcal{R}$, then $w <_{sfs} T$. That is, the existence of R need not cause T to abort unless T reads something R wrote. Menon et al. use timestamps to achieve ALA semantics. Their timestamps, however, are not TL2 timestamps, as they are assigned at begin time, even for read-only transactions. We briefly argue that TL2 timestamps [3] provide SFS (Figure 9).

Let us assume (due perhaps to compiler reordering) that T races with R to prefetch d. This indicates that T could not start after R, and thus $T.\texttt{start} \leq R.\texttt{start}$. If it also holds that $R <_f T$, then when R acquires timestamp t at commit time, $t > R.\texttt{start} \geq T.\texttt{start}$. R subsequently writes t into all lock words that it holds. If $R <_f T$, T must read some datum written by R. During T's call to `TxRead` for d, the test of the timestamp will cause T to abort, restart, and re-read d after R. Alternately, if T reads the lock covering d before R commits, then either $T <_t R$, or T will abort at its next validation. In either case, T cannot simultaneously be ordered before w and after R.

We note that TL2 timestamps reduce the scalability of non-publishing, non-privatizing transactions when compared to *scalable time bases* [14]. They also enforce more ordering than Menon's timestamps, which do not abort a transaction T that reads a value written by a publisher who started (but did not commit) before T began. Finally, TL2-style timestamps preclude partial rollback for closed nested transactions: If B reads a value that C writes, and B is nested within A, then the requirement for B to order after C necessitates that A order after C as well. Even if all accesses prior to B in A do not conflict with C, A must restart to acquire a start time compatible with ordering after C.

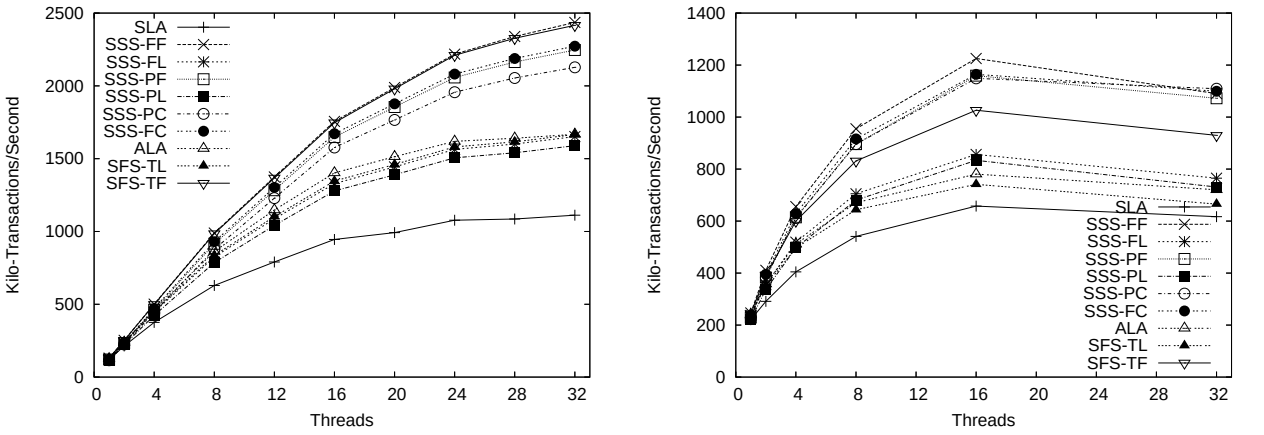
5 Evaluation

In this section, we evaluate the role that selective semantics can play in reducing transaction latency without compromising correctness. We use targeted microbenchmarks to approximate three natural idioms for privatization and publication. All experiments were conducted on an 8-core (32-thread), 1.0 GHz Sun T1000 (Niagara) chip multiprocessor running Solaris 10. All benchmarks were written in C++ and compiled with g++ version 4.1.1 using –O3 optimizations. Data points are the average of five trials.

5.1 STM Runtime Configuration

We use a word-based STM with 1 million ownership records, commit-time locking, and buffered updates. Our STM uses timestamps to avoid validation overhead, and unless otherwise noted, the timestamps employ a scalable time base scheme [14] to safely allow nonconflicting transactions to ignore some timestamp-based aborts. From this STM, we derive 10 runtimes:

- SLA. Uses the start and commit linearization of Menon et al., and polls the global timestamp on reads to avoid the doomed transaction problem.
- SSS-FF. Uses transactional fences before publishing transactions and after privatizing transactions.
- SSS-FL. Uses our commit linearization for privatization, polls the global timestamp to avoid the doomed transaction problem, and uses transactional fences before publishing transactions.
- SSS-PF. Uses polling for publication safety, and transactional fences for privatization.
- SSS-PL. Uses polling for publication safety, commit linearization for privatization, and polling to avoid the doomed transaction problem.
- SSS-PC. Uses polling for publication safety, commit fences for privatization, and polling to avoid the doomed transaction problem.
- SSS-FC. Uses commit fences for privatization, polls the global timestamp to avoid the doomed transaction problem, and uses transactional fences before publishing transactions.



(a) Nontransactional phase of a phased privatization workload, modeled as a low contention red-black tree.

(b) Worklist privatization, using transactional producers and a single private consumer.

Figure 10: Privatization microbenchmarks.

- ALA. Uses TL2-style timestamps for publication safety, and commit linearization with polling of the timestamp for privatization.
- SFS-TL. Identical to ALA, but maintains a privatization counter, separate from the timestamp, to avoid the doomed transaction problem.
- SFS-TF. Uses TL2-style timestamps for publication safety, and transactional fences for privatization.

5.2 Phased Privatization

In some applications, program structure, such as barriers and thread join points, ensures that all threads agree that a datum is public or private [16]. Since these phase boundaries are ordered globally, and with respect to $<_t$, no additional instrumentation is required for correctness on acquiring and releasing transactions. However, as in applications with no privatization or publication, ALA or SLA semantics cause ordering latency for all transactions.

We model the transactional phase of a phased workload with a low-contention red-black tree (Figure 10(a)). Threads use 20-bit keys and perform 80% lookups, 10% inserts, and 10% removes. We ensure steady state by pre-populating the tree to 50% full.

Since SLA serializes all transactions, it consistently underperforms the other runtimes. Similarly, ALA and other mechanisms that use commit linearization fail to scale as well as mechanisms that do not impose additional ordering on all writers at commit time. However, the SSS-FL, SSS-Pl, and SFS-TL curves show that our optimized mechanism for commit linearization, which does not force read-only transactions to increment a global shared counter, achieves better throughput when writing transactions are rare.[5]

Our test platform clearly matters: the Niagara's single shared L2 provides low-latency write misses for the variables used to provide commit linearization, preventing additional slowdown as the lines holding the commit and cleanup counters bounce between cores. At the same time, the Niagara's single-issue cores cannot mask overheads due to polling for publication safety. Thus SSS-FF and SFS-TF perform best. Since commit fences require polling to prevent the doomed transaction problem, SSS-FC performs worse than SSS-FF.

[5]Higher writer ratios show the same separation, with less difference between commit linearization and SLA.

5.3 Worklist Privatization

The privatization problem was first discovered in worklist-based applications, where transactions cooperatively create a task and enqueue it into a nontransactional worklist. When the task is removed from the worklist, the data comprising the task are logically private. Abstractly, these workloads publish by sharing a reference to previously private data, and privatize by removing all shared references to a datum. In the absence of value speculation, these applications admit the privatization problem, but not the publication problem.

To evaluate selective semantics for worklists, we use a producer/consumer benchmark, in which multiple threads cooperatively produce tasks, and then pass the tasks to a consumer thread. We model tasks as red-black trees holding approximately 32 6-bit values, and build tasks using an equal mix of insert, remove, and lookup operations on initially empty trees. Once a tree is found to hold a distinguished value, it is privatized and sent to a nontransactional consumer thread. For the experiment in Figure 10(b), the consumer is fast enough that even 32 producers cannot oversaturate it.

Mechanisms that impose excessive ordering (SLA and ALA) or use commit linearization (SSS-FL, SSS-PL, and SFS-TL) perform worst. Furthermore, since transactions are small, and since privatizing a task does not prevent other producers from constructing a new task, the overhead of a transactional fence (SSS-FF and SSS-PF) at privatization time is as low as the commit fence (SSS-PC and SSS-FC). TL2-style timestamps (ALA, SFS-TL, and SFS-TF) decrease scalability. Again, due to the architecture of the Niagara CPU, polling for publication (SSS-PF, SSS-PL, and SSS-PC) or doomed transaction safety (SLA, SSS-FL, SSS-PL, SSS-PC, SSS-FC, ALA, and SFS-TL) increases latency slightly.

5.4 Indirect Publication

When the value of a shared variable determines whether another location is safe for private access, both the publication and privatization problems can arise. This programming idiom is analogous to locking: the period of private use corresponds to a critical section. We explore it with extensible hashing.

In Figure 11, transactions perform 8 puts into a set of 256 hash tables, where each table uses per-bucket sorted linked lists. If transaction T encounters a bucket containing more than 4 elements, it sets a local flag. After committing, T privatizes and then rehashes any flagged hash tables, doubling the bucket count (initially 8). In order to maintain a steady rate of privatization, if a table's bucket count reaches 2^{13}, the table is privatized and passed to a worker thread W. W replaces the table with an empty 8-bucket table.

Scalability is low, because privatization for rehashing essentially locks the hash tables. Even with 256 tables, the duration of rehashing is significantly longer than the duration of several 8-put transactions, and thus even at two threads, when one thread privatizes a table for rehashing, the other thread is likely to block while attempting to access that table.

The different mechanisms vary only when there is preemption (at 32 worker threads, since there is an additional thread for performing table destruction). At this point, all privatization mechanisms risk waiting for a preempted transaction to resume. The effect is worst for the transactional fence (SSS-FF, SSS-PF, SFS-TF), since it must wait for all active transactions. Our commit linearization (SSS-FL, SSS-PL, ALA, SFS-TL) fares much better, since threads only wait for previously committed writers, who necessarily are at the very end of their execution. SLA linearization lies somewhere in between. Unlike transactional fences (which also avoid the doomed transaction problem), its use of a global timestamp avoids waiting for logically "later" transactions that are still in progress, but unlike commit linearization, it also must wait on "earlier" transactions that have not reached their commit point. The commit fence (SSS-PC, SSS-FC) performs slightly worse than SLA, indicating that waiting on "later" transactions is more expensive than waiting for "earlier" transactions.

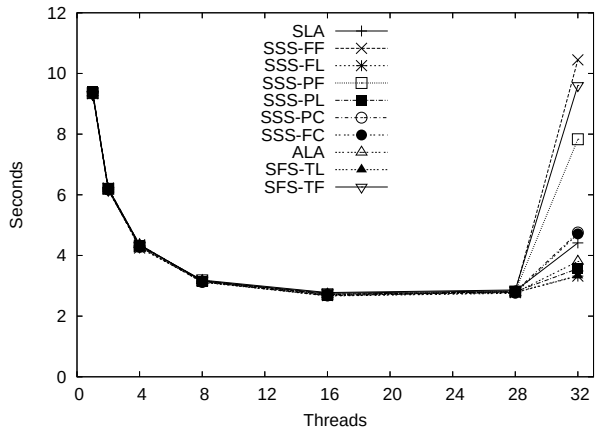


Figure 11: Indirect publication, modeled as extensible hashing with 256 tables. With time on the y axis, lower values are better.

The low latency of fence-based publication appears to be an artifact of the indirect publication idiom. At a fence, the releasing transaction waits for all concurrent transactions to commit or abort and clean up. Since some hash tables are effectively locked, most concurrent threads will execute "restart" transactions to spin-wait for the tables to become public. In our lazy STM, such transactions do not hold locks and can rapidly restart, preventing delays at the publication fence.

6 Conclusions

In this paper we argued that TM semantics should be specified in terms of permissible memory access orderings, rather than by recourse to locks. In our specification, traditional strict serializability (SS) takes the place of single lock atomicity (SLA). To reduce the cost of publication and privatization, we proposed *selective* strict serializability (SSS), which enforces a global order between transactional and nontransactional accesses of a given thread only when transactions are marked as acquiring or releasing. We also proposed a weaker selective flow serializability (SFS), that enforces release ordering only with respect to transactions that read a location written by the releasing transaction. We described several possible implementations of both SSS and SFS, with informal correctness arguments.

Preliminary experiments suggest several performance-related conclusions: (1) By imposing the cost of publication and privatization only when they actually occur, selective ordering of nontransactional accesses can offer significant performance advantages. (2) Given selectivity, there seems to be no compelling argument to relax the serial ordering of transactions. Moreover we suspect that requiring annotations will ultimately help the programmer and compiler to generate race-free code. (3) At the same time, the additional relaxation of SFS (and, similarly, ALA), offers little if any additional benefit. Since SSS is simpler to explain to novice programmers, permits true closed nesting, and is orthogonal to the underlying STM, we currently see no reason to support more relaxed semantics, whether they are defined in terms of prescient lock acquisition or memory access ordering.

References

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

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

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

[5] R. Guerraoui and M. Kapa. On the Correctness of Transactional Memory. In *Proc. of the 13th ACM Symp. on Principles and Practice of Parallel Programming*, Salt Lake City, UT, Feb. 2008.

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

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

[8] V. Luchangco. Against Lock-Based Semantics for Transactional Memory. In *Proc. of the 20th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008. Brief announcement.

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

[10] V. J. Marathe, M. F. Spear, and M. L. Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *Proc. of the 2008 Intl. Conf. on Parallel Processing*, Portland, OR, Sept. 2008.

[11] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. Practical Weak-Atomicity Semantics for Java STM. In *Proc. of the 20th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, Munich, Germany, June 2008.

[12] K. F. Moore and D. Grossman. High-Level Small-Step Operational Semantics for Transactions. In *Conf. Record of the 35th ACM Symp. on Principles of Programming Languages*, San Francisco, CA, Jan. 2008.

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

[14] T. Riegel, C. Fetzer, and P. Felber. Time-based Transactional Memory with Scalable Time Bases. In *Proc. of the 19th Annual ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.

[15] M. L. Scott. Sequential Specification of Transactional Memory Semantics. In *Proc. of the 1st ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006.

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

[17] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing Isolation and Ordering in STM. In *Proc. of the SIGPLAN 2007 Conf. on Programming Language Design and Implementation*, San Diego, CA, June 2007.

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

[19] C. Wang, W.-Y. Chen, Y. Wu, B. Saha, and A.-R. Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *Proc. of the Intl. Symp. on Code Generation and Optimization*, San Jose, CA, Mar. 2007.