Privatization Techniques for Software Transactional Memory*

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

Technical Report #915

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

February 2007

Abstract

Early implementations of software transactional memory (STM) assumed that sharable data would be accessed only within transactions. Memory may appear inconsistent in programs that violate this assumption, even when program logic would seem to make extra-transactional accesses safe. Designing STM systems that avoid such inconsistency has been dubbed the *privatization problem*.

We argue that privatization comprises a pair of symmetric subproblems: private operations may fail to see updates made by transactions that have committed but not yet completed; conversely, transactions that are doomed but have not yet aborted may see updates made by private code, causing them to perform erroneous, externally visible operations. We explain how these problems arise in different styles of STM, present strategies to address them, and discuss their implementation tradeoffs. We also propose a taxonomy of contracts between the system and the user, analogous to programmer-centric memory consistency models, which allow us to classify programs based on their privatization requirements. Finally, we present empirical comparisons of several privatization strategies. Our results suggest that the best strategy may depend on application characteristics.

Keywords: Transactional Memory, Privatization, RSTM, Obstruction Freedom

1 Introduction

Transactions, borrowed from the database world, have become a popular abstraction for parallel programming. Transactional memory (TM) allows the programmer to encapsulate arbitrary memory operations into a *transaction*, which is then guaranteed to be *atomic* (all of its effects appear or none of them do), *isolated* (no intermediate state is ever externally visible, thereby ensuring *serializability*), and and *consistent* (any transaction that preserves program invariants when run sequentially also does so when run in parallel).

Discussions of hardware (HTM) and software (STM) TM implementations have led to subdivision of the isolation property [4, 16]: *Strong isolation* (aka strong atomicity) guarantees that transactions appear to be isolated from non-transactional operations, in addition to other transactions. *Weak isolation* (aka weak atomicity) only guarantees isolation from other transactions; non-transactional operations may observe or affect intermediate states of a transaction.

In either case, an intuitive operational semantics of memory transactions is *single lock atomicity* [16]: "a program executes *as if* all transactions were protected by a single, program-wide mutual exclusion lock". As per mutual exclusion lock semantics, simultaneous access to the same data by transactional and nontransactional code results in a data race. Strong isolation, by definition, eliminates this race. Under weak isolation, however, programmers and (often) TM implementors must take additional steps to guarantee correctness.

*This work was supported in part by NSF grants CNS-0411127 and CNS-0615139, equipment support from Sun Microsystems Laboratories, and financial support from Intel and Microsoft.

```
1  initialize_list(L)
2  T1:                                  T2:
3  begin transaction                    begin transaction
4    node = L->head                       i_node = locate(L,i)
5    L->head = null                       if (i_node != null)
6  end transaction                          i_node->data =
7  // L is privatized                        process(i_node)
8  process(node)                        end transaction
```

Figure 1: A privatization example.

Privatization The obvious way to cope with weak isolation is to ensure that no object is accessed by transactional and nontransactional code at the same time. That is, program logic must partition objects, at each point in time, into those that are *shared*, with access mediated by transactions, and those that are *private* to some thread.[1] The *privatization problem* arises when objects move from one category to the other: the TM system must avoid violations of atomicity, isolation, or consistency during the transition.

In perhaps the simplest case, objects may be "privatized" via whole-program consensus: e.g., between these two barriers, data in row i of matrix M is accessed only by thread i. More commonly, and more problematically from the point of view of implementation, a *privatizing transaction* may modify shared state in such a way that (once the transaction commits) no future successful transaction will access a certain set of objects. In Figure 1, for example, thread $T1$ truncates list L at line 5, thereby privatizing its contents. Thereafter $T1$ should be able to process the entire list without using transactions (as in line 8). If $T1$'s transaction serializes before $T2$, $T2$'s condition on line 5 should fail.

Privatization is semantically straightforward under the single lock atomicity model. However, in this paper we shall see that conventional STM implementations are incapable of guaranteeing correct program behavior in the presence of privatization. Before we discuss this issue in more detail, it is important to understand the two motivations for privatization: performance and escape from the semantic limitations of transactions.

Performance Software transactions are often slower than non-transactional code, usually significantly so [5,10,15,20,23]. Moreover comprehensive hardware TM appears unlikely to be commercially ubiquitous anytime soon [5]. Privatization therefore presents a compelling opportunity to improve STM performance by temporarily exempting objects from the overhead of transactional access. Many STM researchers now envision a programming idiom in which code privatizes a set of objects, processes them nontransactionally, and then "publicizes" them again.[2]

Semantic Limitations STM systems are typically optimistic: they achieve concurrency by pursuing transactions in parallel and then aborting and rolling back in the event of conflict. Such systems are fundamentally incapable of handling operations that cannot safely be rolled back—interactive I/O is the canonical example. An obvious approach is to acquire a true global lock in any transaction that must perform an irreversible operation, but this has obvious consequences for scalability. Privatization enables an alternative strategy: code can splice out the objects that determine the behavior of the irreversible operation, perform that operation in a conservative fashion, and subsequently return the objects to the transactional world.

[1] A private object may be shared by more than one thread if access is mediated by some other form of synchronization (e.g. locks). Such an object is still private from the TM system's point of view. For simplicity of discussion, we consider only thread-private data in the remainder of this paper.

[2] This idiom is not trivial, of course—the privatize-process-publicize cycle is not atomic—but it appears to be reasonable for many applications.

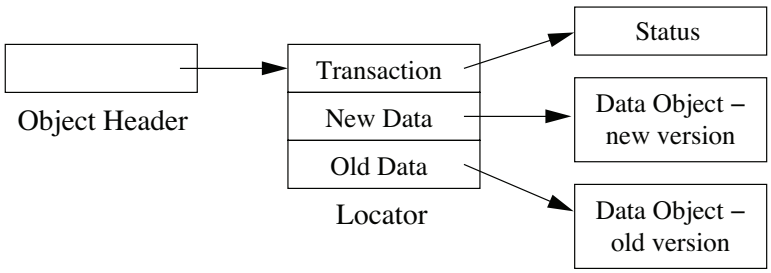

Figure 2: DSTM Metadata. Data Objects are reached through two levels of indirection. The value of *Status* determines whether *old* or *new version* is current.

In this paper, we discuss the privatization problem for optimistic STMs using variants of the Rochester Software Transactional Memory (RSTM) system as running examples. We identify two challenges to correct privatization: (1) private code must see all updates made to privatized objects by previously-committed transactions, and (2) actions in private code must not cause doomed but not-yet-aborted transactions to perform erroneous, externally visible operations.

We consider cloning, undo-logs, and redo-logs as alternative means of buffering speculative writes. We also consider both blocking and nonblocking implementations, and both visible and invisible readers (i.e., implementations in which reader transactions do or do not modify metadata to make their existence visible to writers). For each of these we consider several different approaches to privatization, leading in turn to a natural tradeoff between programming complexity and implementation efficiency. We suggest a taxonomy of contracts between the system and the user, analogous to programmer-centric memory consistency models [2], which allow us to classify programs based on their privatization requirements. Our study indicates that the best performing strategy depends on application characteristics.

In Section 2 we provide background on STM implementations in general, and on RSTM in particular. In Section 3 we consider manifestations of the privatization problem. Our strategies to ensure correctness appear in Section 4, followed by our taxonomy in Section 5. Section 6 discusses measured overheads. Conclusions and future directions appear in Section 7.

2 Background

In this section we briefly categorize several existing STM systems and discuss a naive approach to nontransactional access. In Section 3 we will show that a naive approach is incorrect.

2.1 Transactional Metadata

In general, software TM algorithms work by associating metadata with every shared variable. This per-variable metadata may be embedded within the shared variable itself (in the case of object-oriented systems) or computed based on the address of the variable. In Figures 2–4 we depict the metadata of three STMs: DSTM [11], RSTM [18], and RSTM::RedoLock [25], which is similar to TL2 [6] and uses the RSTM API. These systems also require a small amount of per-transaction metadata (a *Transaction Descriptor*) to represent whether a transaction is logically committed, logically aborted, or still in progress.

Speculative Stores When a transaction writes to shared data, the stores must be performed in a manner that can be undone in the event that the transaction eventually aborts. There are three techniques for making writes speculative:

Cloning: In DSTM and RSTM, on first write the entire object is cloned, and all speculative writes are made

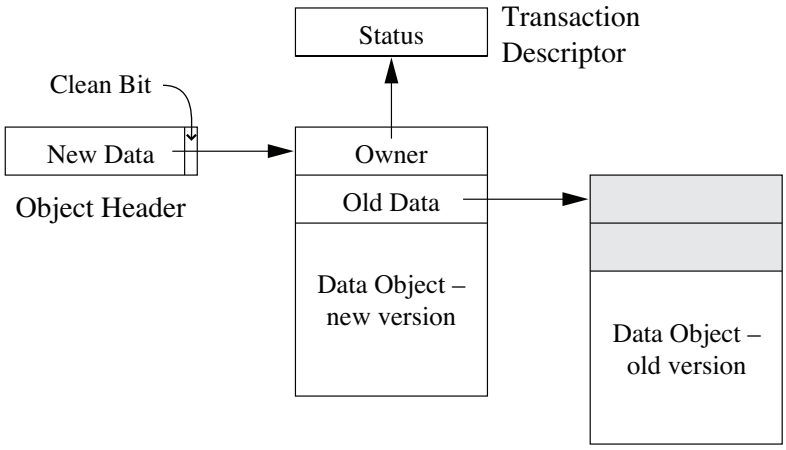

Figure 3: RSTM Metadata. Data Objects are most often reached through one level of indirection, with *old version* valid only for brief periods.

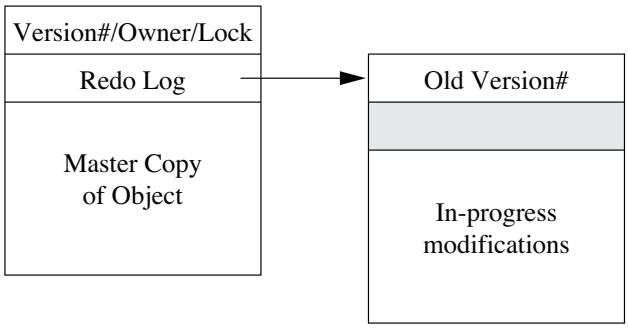

Figure 4: RSTM::RedoLock Metadata. There is no indirection header; instead speculative writes are copied back to the *master copy* on commit. TL2-PO uses similar metadata, replacing per-object version numbers with a globally consistent version timestamp.

to the clone. On commit, the clone becomes the master version of the object; it is discarded on abort.

Redo Logs: TL2 and (optionally) RSTM::RedoLock use *redo logs*. Writes are recorded in a private buffer. On commit, the corresponding locations are locked and the buffered writes are re-issued. On abort, the buffer is discarded.

Undo Logs: McRT [1, 20], Bartok [10], LibLTX [7] and (optionally) RSTM::RedoLock use *undo logs*. Writes are made in-place, and old values are logged in a buffer that is applied on abort.

2.2 The Abstract STM Algorithm

The TM algorithm is essentially independent of its metadata and speculative store mechanism: A transaction traverses memory, logging the locations it reads and speculatively storing modifications. At some point, each to-be-written location is *acquired*, an action which writes the name of the writing transaction in the object's metadata. At the end of the transaction, if all locations previously read and written have not changed, the transaction changes its state to committed with an atomic instruction. The transaction then performs whatever metadata cleanup is required (such as cleaning pointers in RSTM or applying a redo log in TL2).

2.3 Conflict Detection and Validation

When two transactions attempt to access the same location L, at least one transaction wishes to write L, and the system cannot guarantee that the reading transaction will commit first, then at least one of the transactions must abort. In the systems we consider, aborts manifest themselves in two ways:
Contention Management: When a transaction T attempts to access a location L for the first time, it may detect that another active transaction A has acquired L. In this case, T may use a contention manager [22] to decide whether to explicitly abort A (by setting A's status to *Aborted*), to self-abort, or to wait, in the hope that A will commit soon. Similarly, if T intends to write to L but the metadata lists an active reader, T must call contention management.
Validation: When a transaction R reads L, if it does not mark itself as a reader of L, then it assumes responsibility for ensuring that L is not acquired before R commits. Should this check fail, R must abort. The process of checking that all read locations do not change is termed *incremental validation*. Absent a heuristic [24] or global timestamp [6], the total cost to validate the entire read set each time a new location is encountered is $O(n^2)$ for n objects.

Note that when a transaction is explicitly aborted following contention management, it may continue to execute for a brief period until it checks its state and notices the abort. Similarly, a transaction that fails validation may have been in a "doomed" state for some period since its last successful validation. For example, if transaction R starts reading object O, and then transaction W acquires O, makes changes, and commits, then until R validates it is running in a doomed state. Indirection-based systems avoid errors by ensuring that R only observes committed, immutable objects during execution, thereby preventing R from reading part of O and part of the clone of O made by W. Indirection-free systems can ensure correctness by performing *postvalidation*: on every read of a transactional object, the runtime verifies that the object is consistent with past reads before returning a value to user code.

2.4 A Single-Threaded Approach to Privatization

If only one thread is active, then private access to transactional data is, in many systems, a simple operation that depends only on the metadata layout. In indirection-free systems, locations can be accessed nontransactionally without overhead. In RSTM, private accesses to sharable data require an indirection, but are otherwise unencumbered. In DSTM, the overhead is higher: two indirections are required, plus an intervening check of the status of the last acquiring transaction. (This overhead can be reduced, as in ASTM [17], by simplifying the metadata on the first private or read-only reference.) We now turn our focus to privatization in the face of concurrency, demonstrating how accesses that are safe in the single-threaded case can fail, and then presenting thread-safe alternatives.

3 The Privatization Problem

In this section, we present examples that demonstrate erroneous externally visible effects from naive, single-thread style access to privatized data.

3.1 Transactions Cause Incorrect Private Operation

Figure 5 depicts a simple transaction that privatizes a list suffix and prints that suffix. Using incremental validation, a thread executing this code will validate its read set any time an object is read or written for the first time within a transaction, and again at the end of the transaction. Thus line 2 will cause two read-induced validations, line 4 will cause one read-induced validation, line 6 will cause a write-induced

```
function excise_and_print(int v)
 1  begin transaction
 2    prev = list_head; curr = prev->next
 3    while curr->val != v
 4      prev = curr; curr = curr->next
 5      if curr == NULL goto 7
 6    prev->next = NULL
 7  end transaction
 8  while curr != NULL
 9    print curr->val
10    curr = curr->next
```

Figure 5: Pseudocode to privatize the list suffix starting from element v and then print the suffix.

validation, and line 7 will cause a final validation. In systems that use in-place update, postvalidation is required whenever a field is read transactionally (lines 2, 3, and 4).

Now consider a program with two threads: thread $T1$ calls excise_and_print(5) and thread $T2$ calls excise_and_print(7). The list is initialized to $\{1 \ldots 10\}$. If $T1$ executes before $T2$, the correct output is $T1 : \{5, 6, 7, 8, 9, 10\}, T2 : \{\}$. If $T2$ executes first, the correct output is $T1 : \{5, 6\}, T2 : \{7, 8, 9, 10\}$. We now demonstrate that a race within the TM runtime will cause incorrect output, regardless of whether the TM uses indirection, redo logs, or undo logs.

Redo Log: $T1$ Does Not Observe Committed Out-Of-Place Writes In systems that employ redo logs, there is a delay between when a successful transaction commits and when that transaction's modifications become visible to other threads. Let us consider the situation in which $T2$ executes first and commits. Within the end transaction code (line 7), $T2$ must redo all transactional writes after logically committing. Let us suppose that $T2$ is delayed in this task, and thus the write from line 6 (setting node 6's next pointer to NULL) is delayed. If $T1$ executes at this point, it will not read node 7 transactionally, and when it reaches line 10 with curr == node 6 the loop will not terminate. In this case the incorrect output $T1 : \{5, 6, 7, 8, 9, 10\}$ $T2 : \{7, 8, 9, 10\}$ will be observed.

Undo Log: $T1$ Observes Aborted In-Place Writes Conversely, in systems that employ undo logs there exists a delay between when a failed transaction aborts and when its speculative writes are rolled back. Let us consider the same situation as above, but when $T2$ reaches line 7, it delays before its pre-commit validation. Since $T2$ made changes in-place, on line 6 node 6's next pointer was set to NULL. If $T1$ executes to completion at this point, then when it reaches line 7, its commit will logically force $T2$ to abort and retry. When $T2$ does so, it will conclude that 7 is not in the list and will complete without generating output. However, if $T1$ completes the function before $T2$ rolls back its state, the final output $T1 : \{5, 6\}$ $T2 : \{\}$ will be incorrectly generated.

Indirection: $T1$ Reads Logically Unreachable Clone As discussed in Section 2.4, a correct single-thread approach to privatization in indirection-based systems is to assume that the last transaction committed, and use the corresponding object (for RSTM, the object reached with one indirection). This choice leaves $T1$ vulnerable to the exact same error as undo log-based STM implementations. When $T1$ reaches line 10 with curr = node 5, it will choose the wrong version of node 6: rather than use the old, immutable version, $T1$ will read the private copy of node 6 that is under active modification by $T2$. On the next iteration, $T1$ will conclude that there is no node 7, and will stop, generating the incorrect output $T1 : \{5, 6\}$ $T2 : \{\}$.

```
function init()
1   D->binary_data = get_data()
2   C->is_private = false
3   C->next = D
4   A->next = C

function privatize()          // T1
5   begin transaction
6     B = A->next
7     A->next = NULL
8   end transaction
9   B->is_private = true
10  modify(B->next->binary_data)

function custom_transaction   // T2
11  begin transaction
12    z = A
13    y = z->next
14    x = y->next
15    complex_operation(y, x)
16  end transaction

function complex_operation(y, x)
    ...
98  if (y->is_private)
99    print(x->binary_data)
    ...
```

Figure 6: Pseudocode for concurrent transactional and nontransactional accesses with a potentially unsafe function call.

3.2 Private Operations Cause Incorrect Transactional Actions

We now turn our attention to private actions that cause transactions to perform erroneous operations that are externally visible. We assume that private actions do not acquire the locations that they access privately (to do so would require a heavyweight atomic read-modify-write on every location, as well as bookkeeping to release all privately acquired objects should subsequent de-privatization be required). This characteristic permits "doomed" transactions (transactions that are destined to fail but that have not yet detected that they cannot succeed) to read privatized data.

Consider the code in Figure 6. The init function is called to initialize a three-element list. Then thread $T1$ executes privatize() while thread $T2$ executes custom_transaction(). Transactional reads of new locations cause incremental validation on lines 6, 12, 13, and 14. Lines 7, 8, and 16 also trigger incremental validation (due to a write to new location or end of transaction). TMs that do not use indirection will postvalidate the reads on lines 6, 13, 14, 98, and 99.

Indirection: $T2$ Reads Private Write Suppose that $T2$ executes custom_transaction() and calls complex_operation() (code that is sometimes called from outside transactions as well). $T2$ completes line 97 and then

is delayed. At this time, $T1$ begins privatize() and executes through line 10 before $T2$ resumes. Since indirection-based TMs assume that readable objects are immutable, $T2$ does not perform postvalidation on y, and since y has already been accessed, $T2$ does not validate its read set. Consequently, $T2$ does not detect that A has changed, and that it is doomed. In its doomed state, $T2$ reads y->is_private and sees the write made by $T1$ on line 9. $T2$ continues to line 99 and incorrectly prints x->binary_data. Clearly it is incorrect for $T2$ to perform this write. Furthermore, it is possible that $T1$ is in the process of modifying the data $T2$ is printing, so that the printed data does not correspond to any correct state for object B.

No Indirection: Post-validation Does Not Detect Invalid Read Since custom_transaction does not write to shared memory, we consider systems based on either undo log and redo log. These systems follow the same protocol for incremental validation as indirection-based systems, and thus it is sufficient to show that postvalidation does not prevent these systems from also executing line 99. Recall that neither private actions nor transactional reads modify metadata. Since $T1$ only reads B within a transaction, and only accesses C outside of a transaction, $T1$ has no cause to modify the metadata of B or C, and thus postvalidation of y and x on lines 98 and 99 will not detect that the objects have been changed. Consequently, postvalidation will not prevent the transaction from executing line 99.

3.3 Incomplete Solutions

We briefly consider techniques that can resolve some, though not all, dimensions of the privatization problem.

Visible Readers The problem of private operations causing incorrect transactional actions can be avoided in a TM that implements visible reads with post-validation. In such a system, transactions write themselves as readers of every object accessed transactionally, and writers of an object must explicitly abort all readers before acquiring an object. Since an object cannot change without its readers being explicitly aborted, postvalidation does not check that the object version is consistent, but instead simply tests that the reading transaction remains active. In our example, when $T1$ commits (line 8), it is certain that $T2$'s state is aborted, and that on $T2$'s next read of a shared word, $T2$ will detect that it has aborted and will cease operation.

Indirection Another partial solution is for indirection-based systems to require that on the first nontransactional access of an object, the thread does a full inspection of object metadata. This avoids the problem of private operations reading logically unreachable clones, though it does nothing to avoid the problem of private operations causing incorrect transactional actions.

Managed Code With sufficient compiler and runtime support, one can *sandbox* transactions to ensure they have no externally visible effects. The costs of sandboxing are not well documented, but appear significant. In particular, the system would need to: instrument every write that cannot be statically proven to reference already-acquired transactional data; instrument every indirect jump and potentially irreversible system call; install handlers for various run-time exceptions (e.g., divide-by-zero); and arrange for periodic validation, triggered by timers, to interrupt infinite loops.

3.4 Untenable Solutions

Additional techniques, though correct, appear to be unacceptable in practice.

Explicit Copying Without modifying TM runtimes at all, safe privatization could be achieved through explicit copies. In effect, the memory space of an application would be partitioned into one shared heap, only accessed transactionally, and per-thread private heaps. Privatizing transactions would simply copy data from the shared heap to the appropriate private heap, commit a transaction to delete the shared data, and then operate on the private heap.

For large or complex data structures, the cost of creating the private buffer may be unacceptable. For example, if an application needed to privatize a large tree before searching for k elements, the $O(n)$ cost of copying would dwarf the $O(k \log n)$ cost of searching.

Transactional "Private" Access Similarly, we could require private code to continually read and write the metadata of privatized objects in a manner consistent with ongoing transactions. Such code would need to acquire any to-be-written objects, releasing them again at some point after the write. It would also need to perform modifications out-of-place (in redo logs or clones) if required by the STM system. Finally, it would need either to postvalidate its reads or to register in the object's metadata as a visible reader. Any transaction encountering data acquired or visibly read by private code would self-abort.

Given these conventions, private code could safely perform I/O and other irreversible operations. Its contention management could be somewhat simpler than that of true transactions. It could also skip post-access validation of anything for which it became a visible reader, since it could never be aborted. It seems likely, however, that the overhead of acquires, visible reading, and out-of-place updates would eliminate most of the performance advantage one might normally expect from privatization.

4 Solutions to the Privatization Problem

Our examples in Section 3 expose the complex interactions between privatized and nonprivatized code in an STM, and demonstrate that privatization requires some degree of additional synchronization above that provided by existing transactional frameworks. We now consider mechanisms using traditional or transactional constructs that resolve the privatization problem.

4.1 Partitioning by Consensus

Before exploring general solutions to the privatization problem, we mention a programmatic solution of interest. If all threads agree, outside any transaction, that certain data is private, then the thread that "owns" that data can use it safely. The simplest way to obtain such agreement is via a conventional barrier. Our largest existing benchmark for RSTM, an implementation of Delaunay Mesh triangulation,[3] begins by bin-sorting its data set into geometrically partitioned regions. It then spends the overwhelming fraction of its time in a phase that triangulates, in parallel, the points in each geometric region. This phase is delimited by conventional barriers that guarantee private access is safe. Additional, briefer barrier-delimited private phases occur during a "stitching up" operation that joins triangulated regions.

Privatization is safe in the mesh application because the barrier marks a point at which all transactional metadata is in a clean and stable state. The implementation of transactions is immaterial: during a barrier-delimited private phase, no thread performs a transaction. Correctness would still be maintained even with a mix of transactional and nontransactional actions during a barrier phase, so long as program logic ensures that no individual object is accessed both transactionally and nontransactionally. By extension, it would also be maintained in any program whose control flow guarantees that all threads have reached consensus, *outside any transaction*, as to the identity of private data.

4.2 Pessimistic Concurrency Control (PCC)

PCC forces a transaction to acquire reader-writer locks on objects accessed by transactions. The lock protocol is inherently blocking. Suppose, for example, that transaction $T1$ wishes to acquire a write lock on

[3] This application was inspired by, but independent of, the work of Kulkarni et al. [14].

object O, and that $T2$ already holds a read lock on O. $T1$ may attempt to abort $T2$, but $T1$ cannot continue until $T2$ has either committed or explicitly acknowledged that it has aborted *and cleaned up*. As a result, if a transaction commits it is guaranteed that all prior transactions and all conflicting transactions have completed any necessary cleanup operations.

Since the privatization problem occurs precisely when this guarantee cannot be made, pessimistic locking avoids the problems described in Section 3. Unfortunately, prior work suggests [20] that PCC entails unacceptable overhead, primarily due to the overhead of reader locking.

4.3 Explicit Fences

Given the preceding discussion, we turn now to what we consider the interesting case: privatizing transactions with optimistic concurrency control. In this section we present two solutions that introduce blocking, but only for transactions that make part of shared memory private to the executing thread, and that wish to subsequently operate on that data without the overhead of transactions.

The Transactional Fence In Section 4.1, we noted that barriers solve the privatization problem by ensuring that every thread has "agreed" to privatization while outside any transaction. We can similarly assure the safety of a privatizing transaction by waiting, at the privatizer's commit point, for all active transactions to commit or to abort, and to finish any necessary cleanup. We call this wait a *transactional fence*.

The transactional fence mechanism already exists in certain STM systems designed for unmanaged languages, among them RSTM, FSTM [9, Section 5.2.3], and McRT [13]. Lacking automatic garbage collection, these systems need the functionality of the fence to ensure that explicitly deleted data can safely be reclaimed. In this vein, the transactional fence also resembles the *epoch* mechanism of RCU synchronization [19], which avoids overhead in readers by waiting to reclaim old versions of shared data until some distinguished event (e.g., return from the kernel to user space) guarantees that all readers that may have been using the data have completed. Our contribution is to note that a transactional fence suffices to solve the privatization problem as well: following such a fence, threads are guaranteed that private data will reflect all updates performed by prior transactions, and will never subsequently be viewed by a doomed transaction.

The Validation Fence A major shortcoming of the transactional fence is that it can induce unnecessary delays. Let us consider a program with two threads (T_1 and T_2) accessing a single list. T_1 reads the first element of the list as the first step of a complex read-only operation on the first K elements of the list. Meanwhile, T_2 privatizes the sublist beginning at element $K + 2$. There is no reason why T_2 should wait until T_1 completes its transaction. Instead, T_2 need only wait until T_1 reaches its next validation point. As soon as T_1 begins validation, T_2 can be sure that T_1 will not observe T_2's privatized data–either T_2 and T_1 conflict, in which case T_1 will abort, or else they do not conflict, in which case the validation will succeed, affirming that T_1 does not have a reference to T_2's private data.

We use the term *validation fence* to describe an operation in which the caller waits until every thread T has either been outside a transaction or has validated its own (T-local) read set. We expect a validation fence to have significantly lower latency than a transactional fence, particularly when there are long-running transactions that do not conflict with the privatizer. As just described, however, the validation fence ensures only that doomed transactions will never incorrectly observe post-privatization actions. The privatizer is not guaranteed that all preceding committed and aborted transactions have finished cleaning up their speculative writes.

To prevent incorrect reads in private operations, the privatizing thread must inspect (and possibly clean) the metadata associated with an object when it accesses that object for the first time outside of a transaction. (Since determining the first access is in general uncomputable, the compiler may need to make conservative assumptions or, if metadata inspection is expensive, insert run-time first-access checks.) When metadata

indicates that the object is dirty (either due to in-place modification by an aborted transaction or uncommitted out-of-place modification by a committed transaction), the privatizing thread either steals cleanup responsibility or waits for the appropriate thread to cleanup, depending on the underlying TM implementation. For indirection-based systems, cleanup usually entails a single CAS of an object header, after which all dereferences of that header can be performed without testing or branching. In direct-update systems, the private thread may have to apply a redo or undo log.

Interestingly, a *pair* of consecutive validation fences provides the functionality of a full transactional fence. The first validation fence ensures that all conflicting transactions will commit or abort in future without accessing any new data in a conflicting fashion. The second guarantees that conflicting transactions have already cleaned up their speculative writes. This approach certainly imposes higher initial latency on privatizing threads, but eliminates the need for metadata inspection thereafter.

Several different validation strategies have been implemented in extant STM systems [6,8,11,20,21,24]. Correspondingly, the validation fence has varying costs in these systems. In systems that perform validation only at commit time or periodically [8, 20] the validation fence can be expected to be as expensive as a full transactional fence. The validation fence should also be as expensive as a transactional fence in TL2 [6], where version numbers are used to postvalidate objects on every access, and full validation happens only at commit time. On the other hand, systems that validate the entire read set on every new object access [11, 21] will benefit significantly from the validation fence. Finally, privatizers should benefit from the *global commit counter* heuristic of Spear et. al. [24], which retains the correctness guarantees of full validation on every new object access, but avoids most of the work when no transaction (including a privatizer) has committed since the last such validation.

4.4 Nonblocking Privatization

While fences limit the amount of metadata manipulation required for private data, they introduce blocking that may be undesirable in an otherwise nonblocking STM. In this section, we propose a construct for obstruction-free privatization.

In the previous section, we observed that a single validation fence cannot ensure the cleanup of speculative writes, but that metadata inspection on first private access can. Should we discard the validation fence, metadata inspection must be performed on *every access*. Suppose transaction T wishes to acquire object O. Suppose further that T is delayed immediately before it issues its acquiring compare-and-swap (CAS) instruction, at which time thread P privatizes the entire heap and begins writing O. Since we do not require P to modify O's metadata, T's `CAS` will succeed. In any correct TM, T will subsequently abort before writing to O, but in indirection-based TMs there exists a window where nontransactional dereferences of O's header will point to T's version, instead of the version written by P. Metadata inspection on every private access is therefore necessary. When a private thread discovers that an access is invalid, it must assume responsibility for cleanup to ensure nonblocking progress.

On the other side of the interaction, in the absence of a validation fence, a doomed transaction may see writes performed by private code, causing erroneous behavior. Our solution to this problem is to perform full validation (of the entire read set) on each transactional access. To keep costs under control, we suggest a heuristic similar to the *global commit counter* proposed in our earlier work [24]. Whenever a privatizing thread commits, rather then execute a fence we require the thread to atomically increment a global privatization counter (pcount). We then augment the TM validation mechanism so that transactions poll pcount as part of per-access postvalidation. Whenever pcount changes, threads must perform a full validation. Furthermore, threads must poll pcount both before and after full validation, and if pcount changes, they must re-validate. The key observation is that if pcount does *not* change, then no transaction has privatized anything since the last full validation in the transactional thread, so the location just accessed was valid.

Revalidation is unfortunate, but necessary in the face of multiple privatizers. Consider a transaction T that has read objects $\{O_1 \ldots O_k\}$. If privatizer P_1 privatizes object $O_k + 1$, T will validate on its next access to shared memory. After this validation begins, and T has validated $\{O_1 \ldots O_{k/2}\}$, another privatizer P_2 may privatize O_1 and commit. If T does not revalidate, it will not detect that it is doomed, and may incorrectly access memory that has been privatized.

Nonblocking privatization bears considerable overhead: private code pays more for every access, and transactions validate more often. Traditionally, validation of a transaction T's read set must occur whenever a new object is encountered, resulting in $O(n^2)$ overhead for n objects. With nonblocking privatization, the overhead rises to $O(n^2+nP)$, where P is the average number of threads who privatize during T's operation.

We argue that nonblocking privatization is obstruction-free. First we note that to the privatizing thread we have only added a single atomic increment, which is certainly nonblocking given the hardware instructions already used to implement nonblocking STM. Furthermore, we note that in the absence of privatizing threads, a transaction will issue exactly D additional checks of pcount, where D is the number of accesses to shared data within the transaction. This additional work is bounded, and thus in the absence of other conflicts there is no obstacle to the transaction completing.

Lastly, we note that with visible readers, the pcount variable is unnecessary. Privatizing transactions will explicitly abort readers of an object that becomes private. Consequently, with visible readers per-access validation need only verify that the transaction is still *Active*. Privatizing transactions will abort readers explicitly. This suggests that in the face of nonblocking privatization, the tradeoff between invisible and visible readers [24] may shift away from invisible readers, especially given abundant privatizing transactions.

5 A Taxonomy for Single Lock Atomicity

As part of her Ph.D. research some 17 years ago, Sarita Adve proposed that the bewildering array of hardware-centric memory consistency models be reconceptualized as *programmer-centric* contracts between the application and the underlying system [2]. This approach is now standard in the field: programmers may view their system as sequentially consistent provided that they follow a specified programming model—generally some variant of data-race freedom [3].

In a similar vein, we propose that the privatization problem be conceptualized as a contract under which programmers may view transactions as strongly isolated provided that they follow a specified programming model. And just as increasingly restrictive programmer-centric memory models (DRF0 $\rightarrow$ DFR1 $\rightarrow$ PLpc) allow the underlying system to implement more memory optimizations, so do increasingly restrictive programmer-centric sharing models allow the underlying STM system to minimize run-time overheads.

Based on the discussion in preceding sections, we propose the following set of sharing models. We list them from most-restrictive/lowest-STM-overhead to least-restrictive/highest-STM-overhead. Note, however, that in contrast to the typical case with memory models, performance will not necessarily be maximized by using the most restrictive sharing model, since this may limit the programmer's choice of algorithms.

Static Partition Every data object is transactional or nontransactional for the full life of the program. Transactional objects cannot be accessed by nontransactional code. Privatization, whether for performance or for irreversible operations (e.g., I/O), must be achieved by means of copying, as described in Section 3.4.

Partition within Global Consensus Phases Every data object is either transactional or nontransactional at any particular point in time. Objects move from one category to the other only as a result of full-program consensus, achieved outside transactions. Typically, as in our Delaunay mesh application, barriers will be used to delineate private phases.

Privatizing Transactions (Explicit or Implicit) Data objects may be privatized by any committed transaction. Program logic must ensure that the transactional/private status of every object is well defined at all

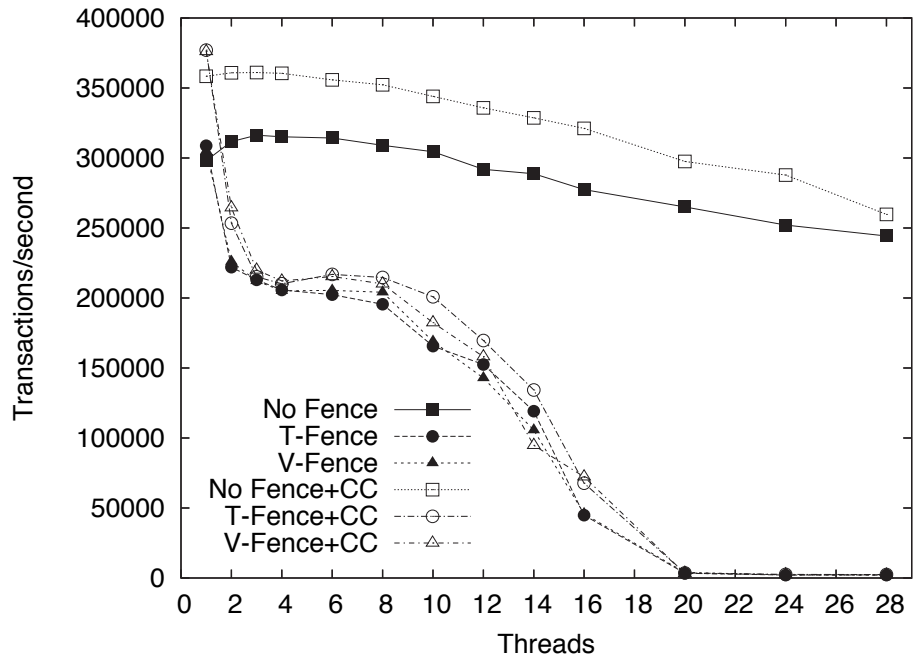

Figure 7: Privatization microbenchmark throughput for a bounded buffer with 10 elements. "CC" indicates that the global commit counter was used to reduce validation overheads.

times—i.e., that this status is never the subject of a race. The implementation may ensure the safety of privatization via pessimistic concurrency control (Section 4.2), transactional or validation fences (Section 4.3), or nonblocking privatization (Section 4.4). Fences, if present, may be explicit (invoked by the program) or implicit (invoked automatically by the library or compiler-generated code on the first use of a sharable object outside a transaction).

Strong Isolation Transactions appear to execute atomically and to serialize not only with each other, but with nontransactional reads and writes as well. For performance reasons some data (e.g., stack locals) may be inherently unsharable, but sharable data can always be accessed safely in nontransactional code, even in the presence of concurrent transactional accesses, without breaking isolation. This model may be implemented by hardware, or, at considerable cost, either by transactional "private" access, as described in Section 3.4 or through extensive compiler integration [12].

Just as there is no consensus on the "right" programmer-centric memory model, we do not expect to see agreement soon on the "right" sharing model for privatization. The choice is likely to depend on the programming language, the underlying STM (or HTM), and the expected expertise of programmers. It also seems likely that refinements of the models presented here will emerge from future work.

6 Evaluation

As a preliminary step toward evaluation of tradeoffs in the implementation of privatization, we have used a simple microbenchmark to compare the transactional fence, the validation fence, and nonblocking privatization—the three implementations we identified as belonging to the *privatizing transactions* category in the taxonomy of Section 5. Though preliminary, these results suggest that the tradeoffs are both workload dependent and non-trivial.

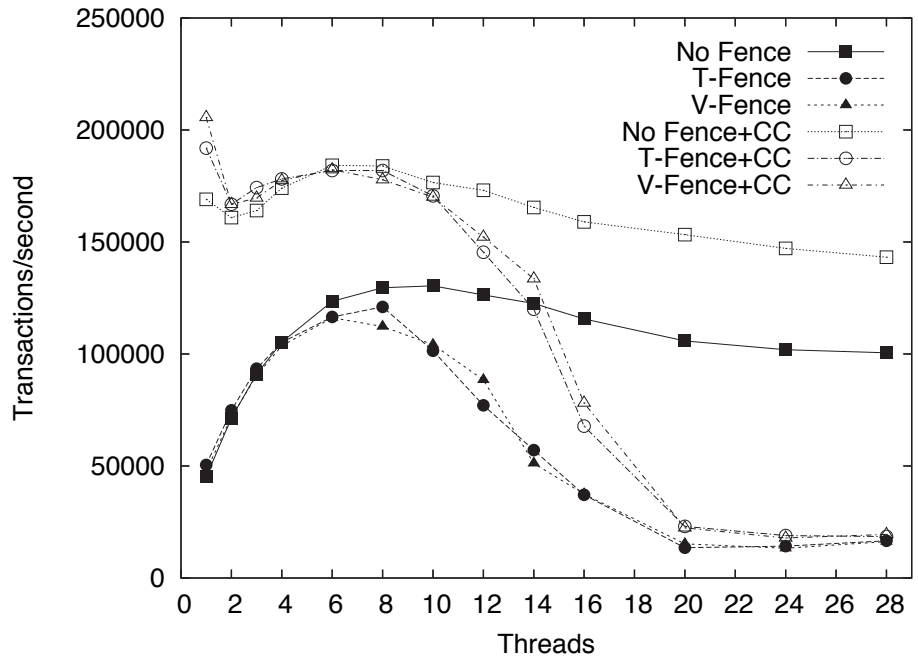

Figure 8: Privatization microbenchmark throughput for a bounded buffers of 100 elements. "CC" indicates that the global commit counter was used to reduce validation overheads.

As noted in Section 4, we have also experimented with a large-scale implementation of Delaunay triangulation. Because this benchmark uses barriers to separate private and transactional phases, it does not require a system-level solution to the privatization problem. At the same time, because it spends the bulk of its time in private work (over 98% with more than 1000 points), it is critically sensitive to per-access overhead in private code. In particular, it suffers a roughly $2\times$ slowdown with indirection-based STM, and a roughly $2\times$ reduction in the number of points it can handle before falling out of the cache (at which point it suffers an additional $6\times$ slowdown).

Our microbenchmark captures a simple producer-consumer scenario: producer transactions add an element to a shared buffer; when the buffer is full, a consumer privatizes its entire content. The buffer is implemented as a list of N tokens; a privatizing thread simply truncates the list within a transaction and commits. The privatizing thread thereafter traverses the privatized list and `frees` its nodes. This choice of consumer behavior was made for its simplicity.

Our experiments were conducted on a 16 processor cache-coherent SunFire 6800. We experimented with both nonblocking (indirection-based) RSTM and blocking (indirection-free) RSTM::RedoLock, both with and without the *global commit counter* [24], and with both eager and lazy acquire.[4] In all cases we used the `Polka` contention manager [22]. We do not present results for eager acquire, since lazy acquire consistently out-performed it. We also present results for invisible readers only; they outperformed visible readers in similar prior experiments. Finally, since the results for the nonblocking and redo-lock RSTM systems were qualitatively similar (redo-lock was generally 20% faster, but with the same scalability), we present results for the nonblocking version only: it is conceptually more compatible with nonblocking privatization. Experimenting with various buffer sizes N, we found values of 10, 100, and 1000 to yield dramatically

[4]Eager acquire performs contention management at the time of the first conflicting access; lazy acquire delays until commit time.

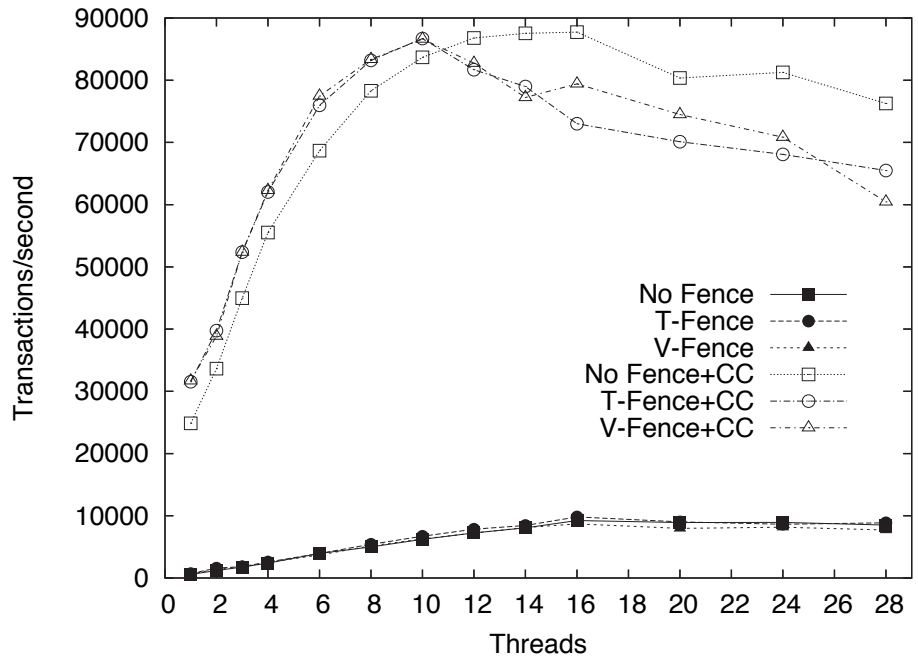

Figure 9: Privatization microbenchmark throughput for a bounded buffer of 1000 elements. "CC" indicates that the global commit counter was used to reduce validation overheads.

different results, particularly for transactional and validation fences. All results are the average of three runs.

Single Threaded Performance and Scalability Figures 7–9 show throughput in transactions per second for the configurations mentioned above. Fences are clearly faster in single threaded runs: the nonblocking scheme incurs extra overhead on every private access (to verify that a doomed transaction has not just acquired the object) and on every transactional access (to verify that private code has not just modified the object). Moreover, with only one active thread, the fence-based schemes incur no waiting overhead. At the same time, the fence-based schemes, as one would expect, do not scale as well as the nonblocking scheme. Their performance deteriorates particularly badly beyond 16 processors, due to preemption of threads for which a privatizer must wait. The global commit counter improves performance in all cases—dramatically so for long transactions.

Transaction Length Our results also demonstrate that the fence schemes are surprisingly expensive for short transactions: the fence becomes a contention hotspot as transactions repeatedly update their state. Note however that with increasing transaction length the difference between performance of the fence and nonblocking schemes progressively reduces. Thus, with substantially large transactions and nontransactional work, scalability problems are mitigated in the fence-based schemes, whereas the overhead of metadata inspection on every transactional access is exacerbated in the nonblocking scheme. With 1000 elements, the fence-based schemes outperform the nonblocking scheme up to 10 threads, after which the fences' overheads exceed the cost of per-access validation.

7 Conclusions

In this paper we characterized the privatization problem, described the subtle errors it can introduce across a variety of STM algorithms, and proposed several techniques to ensure the safety of operations on privatized data. In a simple microbenchmark, we found that variations in workload characteristics, including working set, contention, and preemption, can dramatically shift the relative performance of privatization techniques.

As transactional memory moves into the programming mainstream, privatization is likely to play an important role, allowing programmers (a) to escape the semantic limitations of transactions (e.g., with regard to I/O) and (b) to use application-specific knowledge to improve performance by avoiding STM overhead on private computation. We believe that programmer-centric sharing models can facilitate these efforts, by allowing programmers to think of transactions as strongly isolated so long as they follow the rules.

In a broader context, our experimentation with several variants of RSTM confirms the value of indirection-freedom. This value only increases in the presence of privatization: the relative cost of indirection increases in the absence of other overheads. Our experience with the Delaunay mesh confirms the usefulness of RSTM as a platform for experimentation with STM implementation techniques. At the same time, it confirms that ease of use—the principal argument for transactions—will eventually require language and compiler support.

Acknowledgments

We are grateful to colleagues at Microsoft Research for calling the privatization problem to our attention.

References

[1] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proc. of the SIGPLAN 2006 Conf. on Programming Language Design and Implementation*, Ottawa, ON, Canada, June 2006.

[2] S. V. Adve and M. D. Hill. Weak Ordering—A New Definition. In *Proc. of the 17th Intl. Symp. on Computer Architecture*, pages 2-14, Seattle, WA, May 1990.

[3] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *Computer*, 29(12):66-76, Dec. 1996.

[4] 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.

[5] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid Transactional Memory. In *Proc. of the 12th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2006.

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

[7] R. Ennals. Software Transactional Memory Should Not Be Lock Free. Technical Report IRC-TR-06-052, Intel Research Cambridge, 2006.

[8] K. Fraser and T. Harris. Concurrent Programming Without Locks. Submitted for publication, 2004. Available as `research.microsoft.com/~tharris/drafts/cpwl-submission.pdf`.

[9] K. Fraser. Practical Lock-Freedom. Ph. D. dissertation, UCAM-CL-TR-579, Computer Laboratory, University of Cambridge, Feb. 2004.

[10] 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.

[11] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proc. of the 22nd ACM Symp. on Principles of Distributed Computing*, pages 92-101, Boston, MA, July 2003.

[12] B. Hindman and D. Grossman. Atomicity via Source-to-Source Translation. In *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, San Jose, CA, Oct. 2006.

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

[14] M. Kulkarni, L. P. Chew, and K. Pingali. Using Transactions in Delaunay Mesh Generation. In *Workshop on Transactional Memory Workloads*, Ottawa, ON, Canada, June 2006. Held in conjunction with PLDI 2006.

[15] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid Transactional Memory. In *Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

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

[17] 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.

[18] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the Overhead of Software Transactional Memory. In *ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006. Held in conjunction with PLDI 2006. Expanded version available as TR 893, Dept. of Computer Science, Univ. of Rochester, Mar. 2006.

[19] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russel, D. Sarma, and M. Soni. Read-Copy Update. In *Proc. of the Ottawa Linux Symp.*, July 2001.

[20] 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 Symp. on Principles and Practice of Parallel Programming*, New York, NY, Mar. 2006.

[21] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proc. of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July 2004.

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

[23] A. Shriraman, V. J. Marathe, S. Dwarkadas, M. L. Scott, D. Eisenstat, C. Heriot, W. N. Scherer III, and M. F. Spear. Hardware Acceleration of Software Transactional Memory. In *ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006. Held in conjunction with PLDI 2006. Expanded version available as TR 887, Dept. of Computer Science, Univ. of Rochester, Dec. 2005, revised Mar. 2006.

[24] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict Detection and Validation Strategies for Software Transactional Memory. In *Proc. of the 20th Intl. Symp. on Distributed Computing*, Stockholm, Sweden, Sept. 2006.

[25] 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 Annual ACM Symp. on Parallelism in Algorithms and Architectures*, San Diego, CA, June 2007.