Hardware Acceleration of Software Transactional Memory*

Arrvindh Shriraman, Virendra Marathe, Sandhya Dwarkadas, Michael L. Scott
David Eisenstat, Christopher Heriot, William N. Scherer III, and Michael F. Spear

Technical Report #887

Department of Computer Science, University of Rochester

December 2005

Abstract

Transactional memory (TM) systems seek to increase scalability, reduce programming complexity, and overcome the various semantic problems associated with locks. Software TM proposals run on stock processors and provide substantial flexibility in policy, but incur significant overhead for data versioning and validation in the face of conflicting transactions. Hardware TM proposals have the advantage of speed, but are typically highly ambitious, embed significant amounts of policy in silicon, and provide no clear migration path for software that must also run on legacy machines.

We advocate an intermediate approach, in which hardware is used to accelerate a TM implementation controlled fundamentally by software. We present a system, RTM, that embodies this approach. It consists of a novel *transactional MESI* (TMESI) protocol and accompanying TM software. TMESI eliminates the key overheads of data copying, garbage collection, and validation without introducing any global consensus algorithm in the cache coherence protocol, or any new bus transactions. The only change to the snooping interface is a "threatened" signal analogous to the existing "shared" signal.

By leaving policy to software, RTM allows us to experiment with a wide variety of policies for contention management, deadlock and livelock avoidance, data granularity, nesting, and virtualization.

1 Introduction and Background

Moore's Law has hit the heat wall. Simultaneously, the ability to use growing on-chip real estate to extract more instruction-level parallelism (ILP) is also reaching its limits. Major microprocessor vendors have largely abandoned the search for more aggressively superscalar uniprocessors, and are instead designing chips with large numbers of simpler, more power-efficient cores. The implications for software vendors are profound: for 40 years only the most talented programmers have been able to write good thread-level parallel code; now everyone must do it.

Parallel programs have traditionally relied on mutual exclusion locks, but these suffer from both semantic and performance problems. From a semantic point of view, locks are vulnerable to deadlock, priority inversion, and the inability to restore program invariants if a thread dies in a critical section. From a performance point of view, they are vulnerable to convoying and to arbitrary delays if a thread is preempted or suffers a page fault in a critical section. Most significantly, locks present the programmer with an unhappy tradeoff between concurrency and comprehensibility: coarse-grain lock-based algorithms are relatively easy to understand, (grab the One Big Lock, do what you need to do, and release it) but they preclude any significant parallel speedup. Fine-grain lock-based algorithms allow non-interfering operations to proceed in parallel, but they are notoriously difficult to design, debug, maintain, and understand.

*This work was supported in part by NSF grants CCR-0204344, CNS-0411127, and CNS-0509270, by an IBM Faculty Partnership Award, and by financial and equipment support from Sun Microsystems Laboratories.

Ad hoc *nonblocking* algorithms [18, 19, 30, 31] solve the semantic problems of locks by ensuring that forward progress is never precluded by the state of any thread or set of threads. If two concurrent operations conflict and one is stalled (due, say, to preemption) the other can typically deduce the status of the first, and either back it out of the way or push it through to completion. Ad hoc nonblocking algorithms solve the semantic problems of locks, and provide performance comparable to fine-grain locking, but each such algorithm tends to be a publishable result.

Clearly, what we want is something that combines the semantic advantages of ad hoc nonblocking algorithms with the conceptual simplicity of coarse-grain locks. Transactional memory promises to do so. Originally proposed by Herlihy and Moss [14], transactional memory (TM) borrows the notions of atomicity, consistency, and isolation from database transactions. In a nutshell, the programmer or compiler labels sections of code as *atomic* and relies on the underlying system to ensure that their execution is *linearizable* [12], consistent, and as highly concurrent as possible.

Once regarded as impractical, in part because of limits on the size and complexity of 1990s caches, TM has in recent years enjoyed renewed attention. Unfortunately, it is not yet clear to us that full-scale hardware TM will provide the most practical, cost-effective, or semantically acceptable implementation of transactions. Specifically, hardware TM proposals suffer from three key limitations:

1. They are architecturally ambitious—enough so that commercial vendors will require very convincing evidence before they are willing to make the investment.

2. They embed important policies in silicon—policies whose implications are not yet well understood, and for which current evidence suggests that no one static approach may be acceptable.

3. They provide no obvious migration path from current machines and systems: programs written for a hardware TM system will not run on legacy machines.

Moir et al. [20] have recently proposed solutions to the first and third of these problems. In their *Hybrid Transactional Memory* (HyTM) system, hardware makes a "best effort" attempt to complete transactions, falling back to software when necessary. They envision a spectrum of increasingly ambitious hardware implementations with a common programming model—one that works correctly (if more slowly) on legacy machines. Unfortunately, instances of this approach still embed significant policy in silicon. The initial version, for example, assumes that updates are made at the granularity of individual words, and that ownership of these words is represented by a very specific form of hash table. Likewise it assumes that potential conflicts should be detected as early as possible, forcing transactions to abort and retry in software. While we agree with Moir et al. on the importance of backward compatibility and the usefulness of hardware/software hybrids, we carry the emphasis on software further.

Moir et al. cast software as a "fallback option"—transactions complete entirely in hardware whenever possible, performing whatever minimal checking is necessary to avoid conflicting with concurrent software transactions. We propose that hardware serve simply to optimize the performance of transactions that are controlled fundamentally by software. This allows us, in almost all cases, to cleanly separate policy and mechanism. The former is the province of software; the latter is supported by hardware in cases where we can identify an opportunity for significant performance improvement.

We present a system, RTM, that embodies this software-centric hybrid strategy. RTM comprises a *Transactional MESI* (TMESI) coherence protocol and a modified version of our RSTM software TM [8]. TMESI extends traditional snooping coherence with a "threatened" signal analogous to the existing "shared" signal, and with several new instructions and cache states. One new set of states allows transactional data to be hidden from the standard coherence protocol, until such time as software permits it to be seen. A second set allows metadata to be tagged in such a way that invalidation forces an immediate abort.

In contrast to most software TM systems, RTM eliminates, in the common case, the key overheads of data copying, garbage collection, and consistency validation. In contrast to most hardware proposals, it requires no global consensus algorithm in the cache coherence protocol, no snapshotting of processor state, and no bus messages beyond those already required for MESI. Nonspeculative loads and stores are permitted in the middle of transactions—in fact they constitute the hook that allows us to implement policy in software. Among other things, we rely on software to determine the structure of metadata, the granularity of concurrency and sharing (e.g., word vs. object-based), and the degree to which conflicting transactions are permitted to proceed speculatively in parallel. (We permit, but do not require, read-write and write-write sharing, with delayed detection of conflicts.) Most important, we employ a software *contention manager* [27, 28] to arbitrate conflicts and determine the order of commits.

Because conflicts are handled in software, speculatively written data can be made visible at commit time with only a few cycles of entirely local execution. Moreover, this data (and a small amount of nonspeculative metadata) is *all* that must remain in the cache for fast-path execution: data that were speculatively *read* or *nonspeculatively* written can safely be evicted at any time. Like HyTM, RTM falls back to a software-only implementation of transactions in the event of overflow (or at the discretion of the contention manager), but in contrast not only to HyTM, but to TLR, LTM, VTM, and LogTM as well, it can accommodate "fast path" execution of dramatically larger transactions with a given size of cache.

TMESI is intended for implementation either at the L1 level of a CMP with a shared L2 cache, or at the L2 level of an SMP with write-through L1 caches. We believe that similar extensions could be devised for directory-based coherence protocols. TMESI could also be used with a variety of other TM software. We do not describe such extensions here.

Section 2 surveys the design space for TM systems. Section 3 then turns to an overview our RTM hybrid system, including its programming model, its software control, the costs it needs to address, and the hardware it uses to do so. Section 4 describes that hardware (TMESI) in detail, including its instructions, its protocol states and transitions, and the mechanism used to detect conflicts and abort remote transactions. Section 5 provides additional detail on RTM software control, with an emphasis on how fast-path transactions in hardware interact with unbounded transactions in software. We conclude in Section 6 with a summary of contributions, a brief description of our simulation infrastructure (currently nearing completion), and a list of topics for future research.

2 The Transactional Memory Design Space

In the years following Herlihy and Moss's original TM paper, several groups proposed software implementations of transactional memory, among them Shavit and Touitou, who coined the term "Software Transactional Memory" (STM) [29]. Most of these implementations suffered from one or another fundamental limitation, or had overheads too large to be considered practical. In recent years, however, several groups have developed full-featured STM systems with overheads low enough to outperform coarse-grain locks for highly contended structures. Examples include the word-based systems of Harris & Fraser [3] and Saha et al. [26], and the object-based systems of Harris & Fraser, (OSTM [3]), Herlihy et al. (DSTM [10]), and Marathe et al. (ASTM [15]). The latter is among the best performing, matching the better of DSTM and OSTM on a wide variety of workloads. Unfortunately, when contention is low, locks still enjoy a significant performance advantage.

The first of the modern hardware TM systems is Rajwar and Goodman's Transactional Lock Removal (TLR) [24], which extends their earlier Speculative Lock Elision (SLE) [23]. TLR speculatively elides acquire and release operations in traditional lock-based code, allowing critical sections to execute in parallel so long as their write sets fit in cache and do not overlap. In the event of conflict, all processors but one roll back and acquire the lock conservatively. Timestamping is used to guarantee forward progress. Martínez

and Torrellas [16] describe a related mechanism for multithreaded processors that identifies, in advance, a "safe thread" guaranteed to win all conflicts.

Ananian et al. [1] argue that a TM implementation must support transactions of arbitrary size and duration. They describe two implementations, one of which (LTM) is bounded by the size of physical memory and the length of the scheduling quantum, the other of which (UTM) is bounded only by the size of virtual memory. Rajwar et al. [25] describe a related mechanism (VTM) that uses hardware to *virtualize* transactions across both space and time. Moore et al. [21] attempt to optimize the common case by making transactionally-modified overflow data visible to the coherence protocol immediately, while logging old values for roll-back on abort (LogTM). Hammond et al. [6] propose a particularly ambitious rethinking of the relationship between the processor and the memory, in which *everything* is a transaction (TCC). However, they require heavy-weight global consensus at the time of a commit and add significant hardware complexity to the processor core (i.e., duplicate register files and transactional buffers).

Taken together, the hardware and software TM systems of the last few years explore a very large design space. Some of the dimensions of this space are illustrated in Table 1:[1]

- Most hardware TM proposals (HyTM included) perform updates at the granularity of cache lines. Some STM proposals, notably the WSTM of Harris and Fraser [3] and the McRT system of Saha et al. [26], use word-sized blocks [3, 26] and log updates for roll-back on abort. Absent compile-time data-flow analysis, however, word-based STM systems require nontrivial overhead on every load or store. An attractive alternative in object-oriented languages is to identify blocks with language-level objects, accessed via pointers. Systems that adopt this approach include DSTM [10], OSTM [3], Transactional Monitors [32], STM Haskell [7], SXM [11], and ASTM [15]. Object-based STMs avoid the overhead of bookkeeping for every modified word, but may incur significant copying overhead if only a small fraction of an object is modified. For programs with objects of modest size, we believe the tradeoff tilts strongly in favor of object-based systems. For RTM, which avoids the overhead of copying in the common case, the advantage is compelling.

- A *lock-free* algorithm [13] guarantees that the system as a whole makes forward progress in a bounded number of steps. An *obstruction-free* algorithm [9] guarantees progress only if threads do not actively interfere with one another; it requires an external *contention manager* to avoid the possibility of livelock. A blocking algorithm has situations in which forward progress waits for some particular thread to complete its operation.

- When transactions conflict, aggressive contention management means the thread that notices the conflict actively aborts its "enemy". "Polite" means it waits or aborts itself. "Timestamp" means the older transaction wins. Recursive helping allows OSTM to be lock-free: when a transaction discovers a conflict, it performs whatever work is needed to push its enemy through to completion.

- The occurrence of an eager write (though not necessarily the value) becomes visible to other transactions immediately; lazy writes are hidden until commit time.

- Broadcast writes use hardware support to ensure that subsequent conflicting loads or stores will fail. In the absence of such hardware, an explicit list of readers for a given object allows a writer to abort the readers, one by one. Alternatively, readers may check for potential conflicts at certain appropriate times; depending on system details this may or may not be enough to prevent a doomed transaction from using mutually inconsistent data. Errors resulting from the use of such data may in some systems be caught by type checks or signal handlers.

[1] Characterizations in this table are based on a good faith reading of published papers, but may in some cases reflect a misinterpretation. We would be grateful for corrections from any reader with a more detailed understanding of the systems in question.

	DSTM [10]	OSTM [3]	SXM [11]	ASTM [15]	WSTM [3]	SH [7]	TMon [32]	McRT [26]	HyTM [20]	**RTM**	TLR [24]	TCC [6]	LTM [1]	UTM [1]	VTM [25]	LogTM [21]
Granularity of sharing																
words					×			×								
cache lines									×		×	×	×	×	×	×
objects	×	×	×	×		×	×	×		×						
Liveness																
lock-free		×										×				
obstruction-free	×		×	×	×				×	×			×		×	×
blocking			×			×	×	×	×		×		×	×	×	×
Contention management																
recursive helping		×													?	
aggressive	×		×	×	×				×	×			×			
timestamp	×		×	×	×				×	×	×			×		×
polite	×		×	×		×	×	×	×	×		×				
extensible (your choice)	×		×	×					×	×						
Conflict visibility																
eager	×		×	×	×			×	×	×	×		×	×	×	×
lazy		×		×		×	×			×		×				
conflict detection/tolerance																
broadcast										×	×	×	×		×	
multicast (visible readers)	×		×							×						×
incremental validation	×		×	×		×			×					×	×	
sandboxing		×			×	×										
potential inconsistency		×			×		×	×								
Overflow handling																
fall back to software			—		NA		—		×	×						
log old values, new, or both													N	B	N	O
serialize											×	×				
Nesting																
yes (closed)			×			×		×								
none (yet) or subsumption	×	×		×	×		×		×	×	×	×	×	×	×	×

Correction, Feb. 2006: WSTM uses lazy conflict visibility, not eager.

Table 1: Policies chosen by current TM systems. Software systems appear on the left; hardware on the right; hybrid in the middle. "SH" is STM Haskell. "TMon" is Transactional Monitors (high contention variant). Multiple entries for a given system indicate multiple, hybrid, or adaptive policies.

- When a TM system runs out of room in hardware tables, it can fall back to software, resort to serialization, or virtualize its tables in main memory. In the latter case the tables may contain values to be installed on commit (what Moore et al. call *lazy version management* [21]), values to be restored on abort (*eager version management*), or both. Software systems effectively log both; a transaction commits with a single compare-and-swap instruction, after which new values are immediately visible to other transactions. Like UTM, software and hybrid systems qualify as "eager" in Moore's taxonomy.

- Closed nesting of transactions [22] constitutes a potentially important performance optimization in which an inner transaction can abort and retry without aborting its parent.

Perhaps the most important observation from Table 1 is that there is very little consensus on the right way to implement transactions. Hardware proposals display less variation than software proposals, but this stems in large part not from a clear understanding of tradeoffs, but rather from a tendency to embed more straightforward policies in hardware. Work by Marathe et al. [15] suggests that TM systems should choose between eager and lazy conflict detection based on the characteristics of the application, in order to obtain the best performance. Likewise, work by Scherer et al. [27, 28] and Guerraoui et al. [4, 5] suggests that the preferred contention management policy is also application-dependent, and may alter program run time by as much as an order of magnitude. Not only is there no obvious set of policies to embed in hardware today, it is unclear there will ever be an obvious choice, or that such a choice, if it does emerge, will be amenable to hardware implementation. Given this state of affairs, it seems desirable to remain as flexible as possible, and to leave policy to software when we can.

3 RTM Overview

As noted in Section 2, software TM systems display a wide variety of policy and implementation choices. Our RSTM system [8] draws on experience with several of these in an attempt to eliminate as much software overhead as possible, and to identify and characterize what remains. RTM is, in essence, a derivative of RSTM that uses hardware support to reduce those remaining costs. A transaction that makes full use of the hardware support is called a *hardware transaction*. A transaction that has abandoned that support (due to overflow or policy decisions made by the contention manager) is called a *software transaction*.

3.1 Programming Model

Like most (though not all) STM systems, RTM is *object-based*: updates are made, and conflicts arbitrated, at the granularity of language-level objects.[2] Only those objects explicitly identified as `Shared` are protected by the TM system. Other data (local variables, debugging and logging information, etc.) can be accessed within transactions, but will not be rolled back on abort.

Before a `Shared` object can be used within a transaction, it must be *opened* for read-only or read-write access. RTM enforces this rule using C++ templates and inheritance, but a functionally equivalent interface could be defined through convention in C. The `open_RO` method returns a pointer to the current version of an object, and performs bookkeeping operations that allow the TM system to detect conflicts with future writers. The `open_RW` method, when executed by a software transaction, creates a new copy, or *clone* of the object, and returns a pointer to that clone, allowing other transactions to continue to use the old copy. As in software TM systems, a transaction commits with a single compare-and-swap (CAS) instruction, after which any clones it has created are immediately visible to other transactions (like UTM and LogTM, software and hybrid TM systems employ what Moore et al. refer to as *eager version management* [21]). If a transaction aborts, its clones are discarded.

Figure 1 contains an example of C++ RTM code to insert an element in a singly-linked sorted list of integers. The code traverses the list from the head, opening each node for read-only access, until it finds the right place for the insert. It then re-opens the predecessor for read-write access, creates a new node, and links the node into the list. The `shared` method performs the inverse of `open`, returning an opaque reference suitable for use as a `next` pointer. Similarly, the `Shared<node>` constructor turns a nontransactional `node` into one that can be shared with other transactions.

[2] We do require that each object reside in it own cache line.

```
using namespace rtm;
void intset::insert(int val) {
  BEGIN_TRANSACTION;
    const node* previous = head->open_RO();
    const node* current = previous;

    while (current != NULL) {
        if (current->val >= val) break;
        previous = current;
        current = current->next->open_RO();
    }
    if (!current || current->val > val) {
        node *n = new node(val, current->shared());
            // uses Object<T>::operator new
        if (head == NULL)
            head = new Shared<node>(n);
        else
            previous->open_RW()->next = new Shared<node>(n);
            // Object<T>::open_RW returns this->shared()->open_RW()
    }
  END_TRANSACTION;
}
```

Figure 1: Insertion in a sorted linked list using RTM.

3.2 Software Implementation

The two principal metadata structures in RTM are the *transaction descriptor* and the *object header*. The descriptor contains an indication of whether the transaction is *active*, *committed*, or *aborted*. The header contains a pointer to the descriptor of the most recent transaction to modify the object, together with pointers to old and new clones of the data. If the most recent writer committed in software, the new clone is valid; otherwise the old clone is valid.

Before it can commit, a transaction T must *acquire* the headers of any objects it wishes to modify, by making them point at its descriptor. By using a CAS instruction to change the status word in the descriptor from *active* to *committed*, a transaction can then, in effect, make all its updates valid in one atomic step. Prior to doing so, it must also verify that all the object clones it has been reading are still valid.

Acquisition is the hook that allows RTM to detect conflicts between transactions. If a writer R discovers that a header it wishes to acquire is already "owned" by some other, still active, writer S, R consults a software *contention manager* to determine whether to abort S and steal the object, wait a bit in the hope that S will finish, or abort R and retry later. Similarly, if any object opened by R (for read or write) has subsequently been modified by an already-committed transaction, then R must abort.

RTM can perform acquisition as early as *open* time, or as late as just before commit. The former is know as *eager* acquire, the latter as *lazy* acquire. Most hardware TM systems, by contrast, perform the equivalent of acquisition by requesting exclusive ownership of a cache line. Since this happens as soon as the transaction attempts to modify the line, these systems are inherently restricted to *eager conflict management* [21]. They are also restricted to contention management algorithms simple enough (and static enough) to be implemented in hardware on a cache miss.

As noted at the end of Section 2 there are strong arguments for an adaptive approach to both conflict visibility and contention management. In both these dimensions, RTM provides significantly greater flexibility than pure hardware TM proposals.

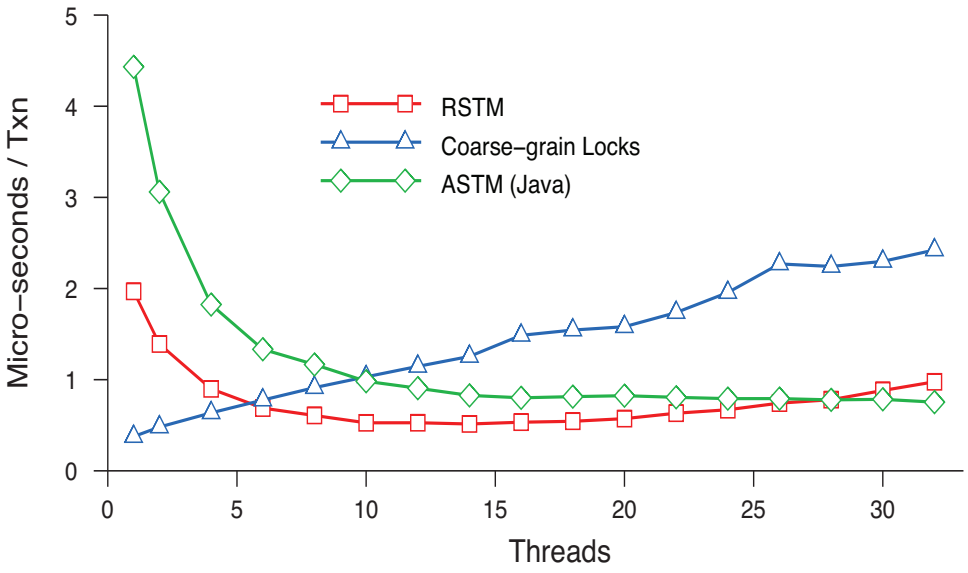


Figure 2: Performance scaling of RSTM, ASTM, and coarse-grain locking on a hash table microbenchmark.

3.3 Dominant Costs

Figure 2 compares the performance of RSTM (the all-software system from which RTM is derived) to that of coarse-grain locking on a hash-table microbenchmark as we vary the number of threads from 1 to 32 on a 16-processor 1.2GHz SunFire 6800. Also shown is the performance (in Java) of ASTM, previously reported [15] to match the faster of Sun's DSTM [10] and the Cambridge OSTM [3] across a variety of benchmarks. Each thread in the microbenchmark repeatedly inserts, removes, or searches for (one third probability of each) a random element in the table. There are 64 buckets, and all values are taken from the range 0–255, leading to a steady-state average of two elements per bucket.

Unsurprisingly, coarse-grain locking does not scale. Increased contention and occasional preemption cause the average time per transaction to climb with the number of threads. On a single processor, however, locking is an order of magnitude faster than ASTM, and more than 4× faster than RSTM. We need about 6 active threads in this program before software TM appears attractive from a performance point of view.

Instrumenting code for the single-processor case, we can apportion costs as shown in Figure 3, for five different microbenchmarks. Four—the hash table of Figure 2, the sorted list whose insert operation appeared in Figure 1, and two red-black trees—are implementations of the same abstract set. The fifth represents the extreme case of a trivial critical section—in this case one that increments a single integer counter.

In all five microbenchmarks TM overhead dwarfs real execution time. Because they have significant potential parallelism, however, both HashTable and RBTree outperform coarse-grain locks given sufficient numbers of threads. Parallelism is nonexistent in Counter and limited in LinkedList: a transaction that updates a node of the list aborts any active transactions farther down the list.

Memory management in Figure 3 includes the cost of allocating, initializing, and (eventually) garbage collecting clones. The total size of objects written by all microbenchmarks other than RBTree-Large (which uses 4 KByte nodes instead of the 28 byte nodes of RBTree-Small) is very small. As demonstrated by RBTree-Large, transactions that access a very large object (especially if they update only a tiny portion of it) will suffer enormous copying overhead.

In transactions that access many small objects, *validation* is the dominant cost. It reflects a subtlety of conflict detection not mentioned in Section 3.2. Suppose transaction R opens objects X and Y in read-only

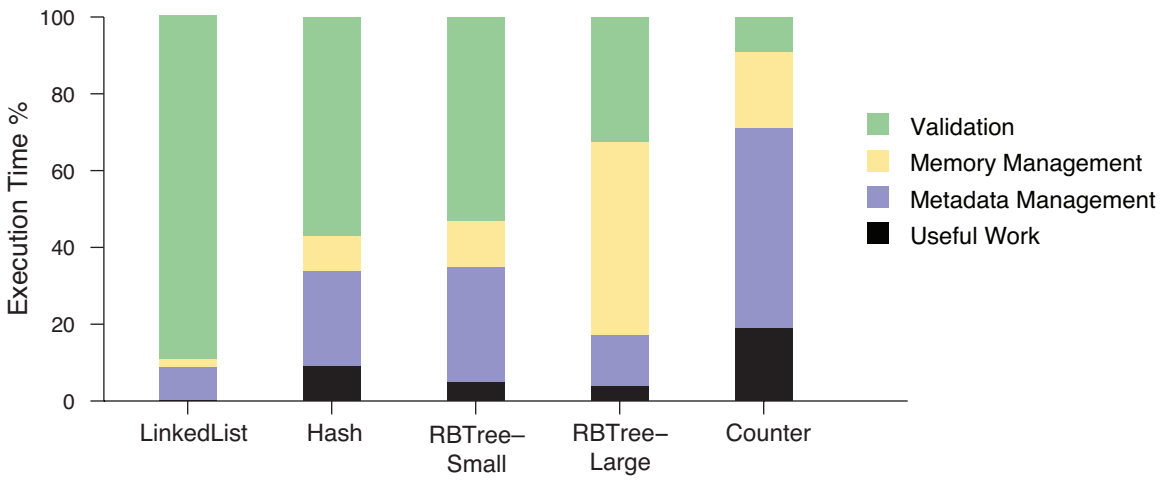


Figure 3: Cost breakdown for RSTM on a single processor, for five different microbenchmarks.

mode. In between, suppose transaction S acquires both objects, updates them, and commits. Though R is doomed to abort (the version of X has changed), it may temporarily access the old version of X and the new version of Y. It is not difficult to construct scenarios in which this *mutual inconsistency* may lead to arbitrary program errors, induced, for example, by stores or branches employing garbage pointers. (Hardware TM systems are not vulnerable to this sort of inconsistency, because they roll transactions back to the initial processor and memory snapshot the moment conflicting data becomes visible to the cache coherence protocol.)

Without a synchronous hardware abort mechanism, RSTM (like DSTM and ASTM) requires R to double-check the validity of all previously opened objects whenever opening something new. For a transaction that accesses a total of n objects, this *incremental validation* imposes $O(n^2)$ total overhead.

As an alternative to incremental validation, Herlihy's SXM [11] and more recent versions of DSTM allow readers to add themselves to a *visible reader* list in the object header at acquire time. Writers must abort all readers on the list before acquiring the object. Readers ensure consistency by checking the status word in their transaction descriptor on every *open* operation. Unfortunately, the constant overhead of reader list manipulation is fairly high. In practice, incremental validation is cheaper for small transactions (as in Counter); visible readers are cheaper for large transactions (as in LinkedList); neither clearly wins in the common middle ground [28]. RSTM currently employs incremental validation in all cases.

3.4 Hardware Support

RTM uses hardware support to address the memory management and validation overhead of software TM. In so doing it eliminates the top two components of the overhead bars shown in Figure 3.

1. Our TMESI protocol allows transactional data, buffered in the local cache, to be hidden from the normal coherence protocol. This buffering allows RTM, in the common case, to avoid allocating and initializing a new copy of the object in software. Like most hardware TM proposals, RTM keeps only the new version of speculatively modified data in the local cache. The old version is written through to memory if necessary at the time of the first transactional store. The new version becomes visible to the coherence protocol when and if the transaction commits. Unlike most hardware proposals (but like TCC), RTM allows data to be speculatively read or even written when it is also being written

by another concurrent transaction. TCC ensures, in hardware, that only one of the transactions will commit. RTM relies on software for this purpose.

2. TMESI also allows selected metadata, buffered in the local cache, to be tagged in such a way that invalidation will cause an immediate abort of the current transaction. This mechanism allows the RTM software to guarantee that a transaction never works with inconsistent data, without incurring the cost of incremental validation or visible readers (as in software TM), without requiring global consensus for hardware commit (as in TCC), and without precluding read-write and write-write speculation.

To facilitate atomic updates to multiword metadata (which would otherwise need to be dynamically allocated, and accessed through a one-word pointer), RTM also provides a wide compare-and-swap, which atomically inspects and updates several adjacent locations in memory.

A transaction could, in principle, use hardware support for certain objects and not for others. For the sake of simplicity, RTM currently takes an all-or-nothing approach: a transaction initially attempts to leverage TMESI support for write buffering and conflict detection of all of its accessed objects. If it aborts for any reason, it retries as a software transaction. Aborts may be caused by conflict with other transactions (detected through invalidation of tagged metadata), by the loss of buffered state to overflow or insufficient associativity, or by executing the *Abort* instruction. (The kernel executes *Abort* on every context switch.)

In the terminology of Table 1, RTM is object-based and obstruction-free. It can use an arbitrary out-of-band contention manager; by default we use the "Polka" policy of Scherer et al. [28]. Like ASTM, RTM adapts dynamically between lazy and eager acquire. At acquire time it broadcast-invalidates hardware readers by writing to metadata they have *ALoad*ed. It invalidates software readers one-by-one; these must have previously registered themselves in the object header.

4 TMESI Hardware Details

In this section, we discuss the details of hardware acceleration for common-case transactions, which have bounded time and space requirements. In order, we consider ISA extensions, the TMESI protocol itself, and support for conflict detection and immediate aborts. We conclude with an example.

4.1 ISA Extensions

RTM requires eight new hardware instructions, listed in Table 2.

The *SetHandler* instruction indicates the address to which control should branch in the event of an immediate abort (to be discussed at greater length in Section 4.3). This instruction could be executed at the beginning of every transaction, or, with OS kernel support, on every heavyweight context switch.

The *TLoad* and *TStore* instructions are *transactional* loads and stores. All accesses to transactional data are transformed (via compiler support) to use these instructions. They move the target line to one of five *transactional states* in the local cache. Transactional states are special in two ways: (1) they are not invalidated by read-exclusive requests from other processors; (2) if the line has been the subject of a *TStore*, then they do not supply data in response to read or read-exclusive requests. More detail on state transitions appears in Section 4.2.

The *ALoad* instruction supports immediate aborts of remote transactions. When it *acquires* a to-be-written object, RTM performs a nontransactional write to the object's header. Any reader transaction whose correctness depends on the consistency of that object will previously have performed an *ALoad* on the header (at the time of the *open*). The read-exclusive message caused by the nontransactional write then serves as a broadcast notice that immediately aborts all such readers. A similar convention for transaction descriptors allows hardware transactions to immediately abort software transactions even if those software transactions

Instruction	Description
SetHandler (H)	Indicate address of user-level abort handler
TLoad (A, R)	Transactional Load from A into R
TStore (R, A)	Transactional Store from R into A
ALoad (A, R)	Load A into R; tag "abort on invalidate"
ARelease (A)	Untag *ALoad*ed line
CAS-Commit (A, O, N)	End Transaction
Abort	Invoked by transaction to abort itself
Wide-CAS (A, O, N, K)	Update K adjacent words atomically

Table 2: ISA Extensions for RTM.

don't have room for all their object headers in the cache (more on this in Section 4.3). In contrast to most hardware TM proposals, which eagerly abort readers whenever another transaction performs a conflicting transactional store, TMESI allows RTM to delay acquires when speculative read-write or write-write sharing is desirable [15].

The *ARelease* instruction erases the abort-on-invalidate tag of the specified cache line. It can be used for *early release*, a software optimization that dramatically improves the performance of certain transactions, notably those that search large portions of a data structure prior to making a local update [10, 15]. It is also used by software transactions to release an object header after copying the object's data.

The *CAS-Commit* instruction performs the usual function of compare-and-swap. In addition, if the CAS succeeds, transactional lines revert to their corresponding MESI states and begin to respond as usual to coherence messages. If the CAS fails, then speculatively read lines revert to their corresponding states, but speculatively written lines are invalidated, and control transfers to the location registered by *SetHandler*. In either case, abort-on-invalidate lines are all untagged. The motivation behind *CAS-Commit* is simple: software TM systems invariably use a CAS to commit the current transaction; we overload this instruction to make buffered transactional state once again visible to the coherence protocol.

The *Abort* instruction clears the transactional state in the cache in the same manner as a failed *CAS-Commit*. Its principal use is to implement condition synchronization by allowing a transaction to abort itself when it discovers that its precondition does not hold. Such a transaction will typically then jump to its abort handler. *Abort* is also executed by the scheduler on every context switch.

The *Wide-CAS* instruction specifies three addresses and a length. If the specified number of memory words at location A match those at location O, they are replaced with the ones at location N. *Wide-CAS* is intended for fast update of object headers. It works only within one cache line.

4.2 TMESI Protocol

A central goal of our design has been to maximize software flexibility while minimizing hardware complexity. Like most hardware TM proposals (but unlike TCC or Herlihy & Moss's original proposal), we use the processor's cache to buffer a single copy of each transactional line, and rely on shared lower levels of the memory hierarchy to hold the old values of lines that have been modified but not yet committed. Like TCC—but unlike most other hardware systems—we permit mutually inconsistent versions of a line to reside in different caches. Where TCC requires an expensive global arbiter to resolve these inconsistencies at commit time, we rely on software to resolve them at acquire time. *CAS-Commit* is a purely local operation (unlike TCC, which broadcasts all written lines) that exposes modified lines to subsequent coherence traffic.

Our protocol requires no bus messages other than those already required for MESI. We add two new processor messages, PrTRd and PrTWr, to reflect *TLoad* and *TStore* instructions, respectively, but these are

visible only to the local cache. We also add a wired-or "threatened" bus signal (T) analogous to the existing "shared" signal (S). The T signal serves to warn a reader transaction of the existence of a potentially conflicting writer. Because the writer's commit will be a local operation, the reader will have no way to know when or if it actually occurs. It must therefore make a conservative assumption when it reaches the end of its own transaction (Until then the line is protected by the software TM protocol).

4.2.1 State transitions

Figure 4 contains a transition diagram for the TMESI protocol. The four states on the left comprise the traditional MESI protocol. The five states on the right, together with the bridging transitions, comprise the TMESI additions. Cache lines move from a MESI state to a TMESI state on a transactional read or write. Once a cache line enters a TMESI state, it stays in the transactional part of the state space until the current transaction commits or aborts, at which time it reverts to the appropriate MESI state, indicated by the second (commit) or third (abort) letters of the transactional state name.

The *TSS*, *TEE*, and *TMM* states behave much like their MESI counterparts. In particular, lines in these states continue to supply data in response to bus messages. The two key differences are (1) on a PrTWr we transition to *TMI*; (2) on a BusRdX (bus read exclusive) we transition to *TII*. These two states have special behavior that serves to support speculative read-write and write-write sharing. Specifically, *TMI* indicates that a speculative write has occurred on the local processor; *TII* indicates that a speculative write has occurred on a remote processor, but not on the local processor.

A *TII* line must be dropped on either commit or abort, because a remote processor has made speculative changes which, if committed, would render the local copy stale. A *TMI* line is the complementary side of the scenario. On abort it must be dropped, because its value was incorrectly speculated. On commit it will be the only valid copy; hence the reversion to *M*. Software must ensure that conflicting writers never both commit, and that if a conflicting reader and writer both commit, the reader does so first from the point of view of program semantics.[3] Lines in *TMI* state assert the T signal on the bus in response to BusRd messages. The reading processor then transitions to *TII* rather than *TSS* or *TEE*. Processors executing a *TStore* instruction ignore the T signal, on the assumption that only one of the writers will commit.

Among hardware TM systems, only TCC and RTM support read-write and write-write sharing; all the other schemes mentioned in Sections 1 and 3 use eager conflict detection. By allowing a reader transaction to commit before a conflicting writer acquires the contended object, RTM permits significant concurrency between readers and long-running writers. Write-write sharing is more problematic, since only one transaction can usually commit, but may be desirable in conjunction with early release [15]. Note that nothing about the TMESI protocol *requires* read-write or write-write sharing; if the software protocol detects and resolves conflicts eagerly, the *TII* and *TMI* states will simply go unused.

4.2.2 Abort-on-invalidate

In addition to the states shown in Figure 4, the TMESI protocol provides *AM*, *AE*, and *AS* states. The A bit is set in response to an *ALoad* instruction, and cleared in response to an *ARelease*, *CAS-Commit*, or *Abort* instruction (each of these requires an additional processor–cache message not shown in Figure 4). Invalidation or eviction of an *Ax* line aborts the current transaction. If the processor is executing at interrupt level when an abort occurs, delivery is deferred until the return from interrupt. If the processor is in kernel mode when the abort is delivered, delivery takes the form of an exception. If the processor is in user mode,

[3] There are many ways to misuse TMESI hardware. Some of these (e.g., a nontransactional load or store to a line currently cached in a transactional state) produce a hardware exception. Others (e.g., conflicting writers that both commit) silently leave the system in an inconsistent state. RTM correctness is fundamentally dependent on correct software. We do not enumerate the failure states here.

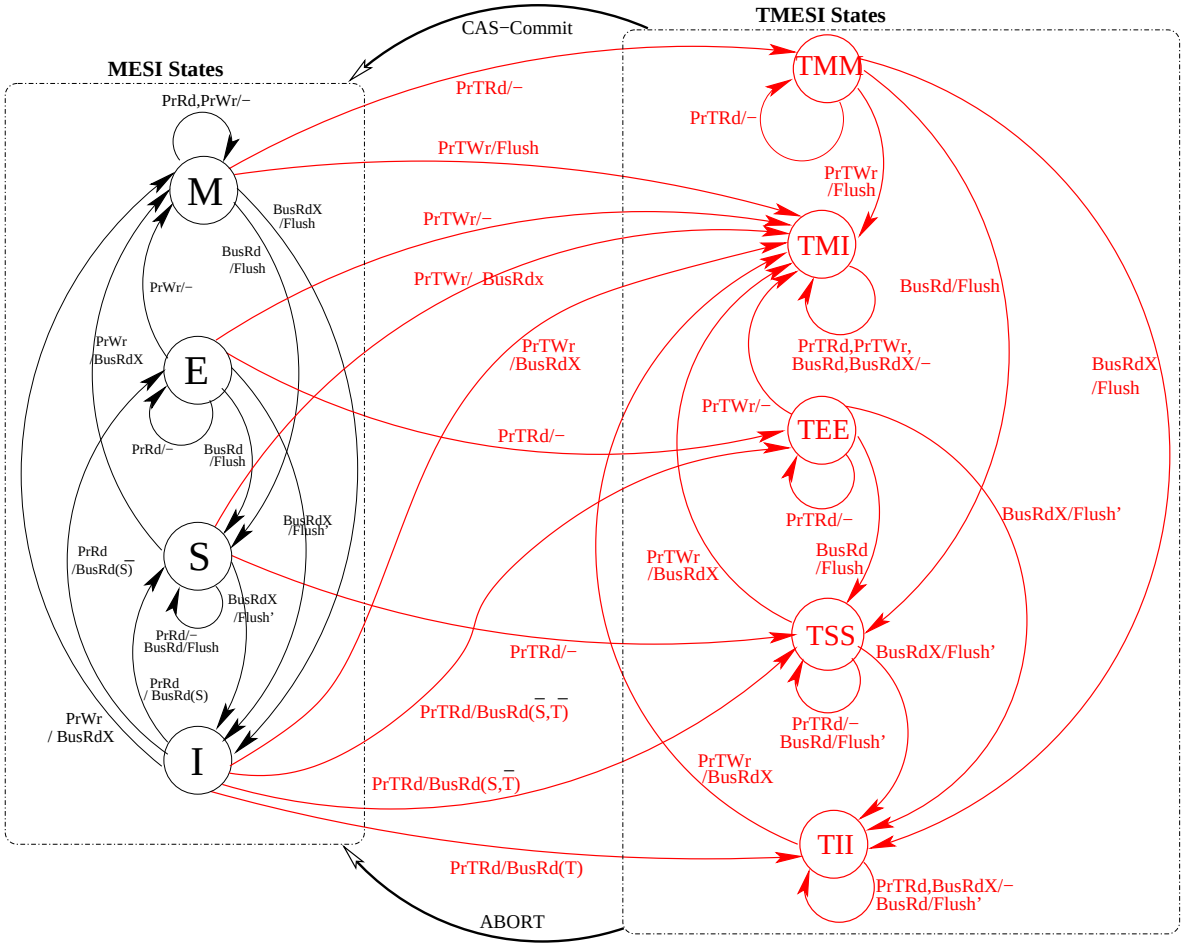
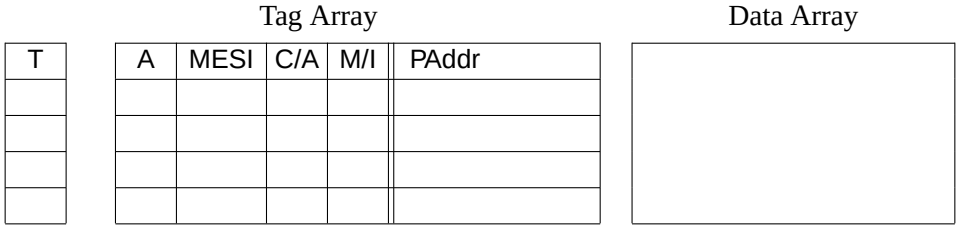


Figure 4: TMESI Protocol. Dashed boxes enclose the MESI and TMESI subsets of the state space. All TMESI lines revert to MESI states in the wake of a *CAS-Commit* or *Abort*. Specifically, the 2nd and 3rd letters of a TMESI state name indicate the MESI state to which to revert on commit or abort, respectively. Notation on transitions is conventional: the part before the slash is the triggering message; after is the ancillary action. "Flush" indicates that the cache supplies the requested data; "Flush′" indicates it does so iff the base protocol prefers cache–cache transfers over memory–cache. When specified, S and T indicate signals on the "shared" and "threatened" bus lines; an overbar means "not signaled".

delivery takes the form of a spontaneous subroutine call. The current program counter is pushed on the user stack, and control transfers to the address specified by the most recent *SetHandler* instruction. If either the stack pointer or the handler address is invalid, an exception occurs.

*ALoad*s serve three related roles in RTM. First, every transaction *ALoad*s its own transaction descriptor (the word it will eventually attempt to *CAS-Commit*). If any other transaction aborts it (by CAS-ing its descriptor to *aborted*), the first transaction is guaranteed to notice immediately. Second, every hardware transaction *ALoad*s the headers of objects it reads, so it will abort if a writer acquires them. Third, a software transaction *ALoad*s the header of any object it is copying (*ARelease*ing it immediately afterward), to ensure the integrity of the copy. Note that a software transaction never requires more than two *ALoad*ed words at once, and we can guarantee that these are never evicted from the cache.



T	A	MESI	C/A	M/I	State
0	0	0 0	—	—	I
0	0	1 1	0	0	I
0	0	0 1	—	—	S
0	0	1 0	—	—	E
0	0	1 1	1	—	M
0	0	1 1	0	1	M
1	0	0 0	—	—	TII
1	0	0 1	—	—	TSS
1	0	1 0	—	—	TEE
1	0	1 1	—	0	TMI
1	0	1 1	—	1	TMM
0	1	0 1	—	—	AS
0	1	1 0	—	—	AE
0	1	1 1	1	—	AM
0	1	1 1	0	1	AM

T	Line is (1) / is not (0) transactional
A	Line is (1) / is not (0) abort-on-invalidate
MESI	2 bits: I (00), S (01), E (10), or M (11)
C/A	Most recent txn committed (1) or aborted (0)
M/I	Line is/was in TMM (1) or TMI (0)

Table 3: Tag Array encoding. Tags are organized schematically as shown at top, with the T bit used to selectively enable broadcast on the C/A line. Interpretations of the bits (below right) give rise to 15 valid encodings of the 12 TMESI states.

4.2.3 State tag encoding

All told, a TMESI cache line can be in any of 12 different states: the four MESI states (*I*, *S*, *E*, *M*), the five transactional states (*TII*, *TSS*, *TEE*, *TMM*, *TMI*), and the three abort-on-invalidate states (*AS*, *AE*, *AM*). For the sake of fast commits and aborts, we encode these in five bits, as shown in Table 3.

At commit time, if the CAS in *CAS-Commit* succeeds, we first broadcast a 1 on the C/A bit line, and use the T bits to conditionally enable only the tags of transactional lines. Following this we flash-clear the A and T bits. For *TSS*, *TMM*, *TII*, and *TEE* the flash clear alone would suffice, but *TMI* lines must revert to *M* on commit and *I* on abort. We use the C/A bit to distinguish between these: a line is interpreted as being in state *M* if its MESI bits are 11 and either C/A or M/I is set. On Aborts we broadcast 0 on the C/A bit line.

4.3 Conflict Detection & Immediate Aborts

Hardware TM systems typically checkpoint processor state at the beginning of a transaction. As soon as a conflict is noticed, the hardware restarts the losing transaction. Most hardware systems make conflicts visible as soon as possible; TCC delays detection until commit time. Software systems, by contrast, require that transactions *validate* their status explicitly, and restart themselves if they have lost a conflict.

The overhead of validation, as we saw in Section 3.3, is one of the dominant costs of software TM. RTM avoids this overhead by *ALoad*ing object headers in hardware transactions. When a writer modifies

the header, all conflicting readers are aborted by a single (broadcast) BusRdX. In contrast to most hardware TM systems, this broadcast happens only at acquire time, *not* at the first transactional store, allowing flexible policy.

Unfortunately, nothing guarantees that a software transaction will have all of its object headers in *ALoad*ed lines. Moreover software validation at the next *open* operation cannot ensure consistency: because hardware transactions modify data in place, objects are not immutable, and inconsistency can arise among words of the same object read at different times. The RTM software therefore makes every software transaction a visible reader, and arranges for it to *ALoad* its own transaction descriptor. Writers (whether hardware or software) abort such readers at acquire time, one by one, by writing to their descriptors. In a similar vein, a software writer *ALoad*s the header of any object it needs to clone, to make sure it will receive an immediate abort if a hardware transaction modifies the object in place during the cloning operation.[4]

Because RTM detects conflicts based on access to object headers only, correctness for hardware transactions does not require that *TII*, *TSS*, *TEE*, or *TMM* lines remain in the cache. These can be freely evicted and reloaded on demand. When choosing lines for eviction, the cache preferentially retains *TMI* and *Ax* lines. If it must evict one of these, it aborts the current transaction, which will then retry in software. Other hardware schemes buffer both transactional reads and writes, exerting much higher pressure on the cache.

The abort delivery mechanism, described in Section 4.2.2, allows both the kernel and user programs to execute hardware transactions, so long as those transactions complete before control transfers to the other. The operating system is expected to abort any currently running user-level hardware transaction when transferring from an interrupt handler into the top half of the kernel. Interrupts handled entirely in the bottom half (TLB refill, register window overflow) can safely coexist with user-level transactions. Interrupt handlers themselves cannot make use of transactions. User transactions that take longer than a quantum to run will inevitably execute in software. With simple statistics gathering, RTM can detect when this happens repeatedly, and skip the initial hardware attempt.

4.4 Example

Figure 5 illustrates the interactions among three simple concurrent transactions. Only the transactional instructions are shown. Numbers indicate the order in which instructions occur. At the beginning of each transaction, RTM software executes a *SetHandler* instruction, initializes a transaction descriptor (in software), and *ALoad*s that descriptor. Though the *open* calls are not shown explicitly, RTM software also executes an *ALoad* on each object header at the time of the *open* and before the initial *TLoad* or *TStore*.

Let us assume that initially objects A and B are invalid in all caches. At ❶ transaction T1 performs a *TLoad* of object A. RTM software will have *ALoad*ed A's header into T1's cache in state *AE* (since it is the only cached copy) at the time of the *open*. The referenced line of A is then loaded in *TEE*. When the store happens in T2 at ❷, the line in *TEE* in T1 sees a BusRdX message and drops to *TII*. The line remains valid, however, and T1 can continue to use it until T2 acquires A (thereby aborting T1) or T1 itself commits. Regardless of T1's outcome, The *TII* line must drop to *I* to reflect the possibility that a transaction threatening that line can subsequently commit.

At ❸ T1 performs a *TStore* to object B. RTM loads B's header in state *AE* at the time of the *open*, and B itself is loaded in *TMI*, since the write is speculative. If T1 commits, the line will revert to *M*, making the *TStore*'s change permanent. If T1 aborts, the line will revert to *I*, since the speculative value will at that point be invalid.

[4] An immediate abort is not strictly necessary if the cloning operation is simply a bit-wise copy; for this it suffices to double-check validity after finishing the copy. In object-oriented languages, however, the user can provide a class-specific `clone` method that will work correctly only if the object remains internally consistent.

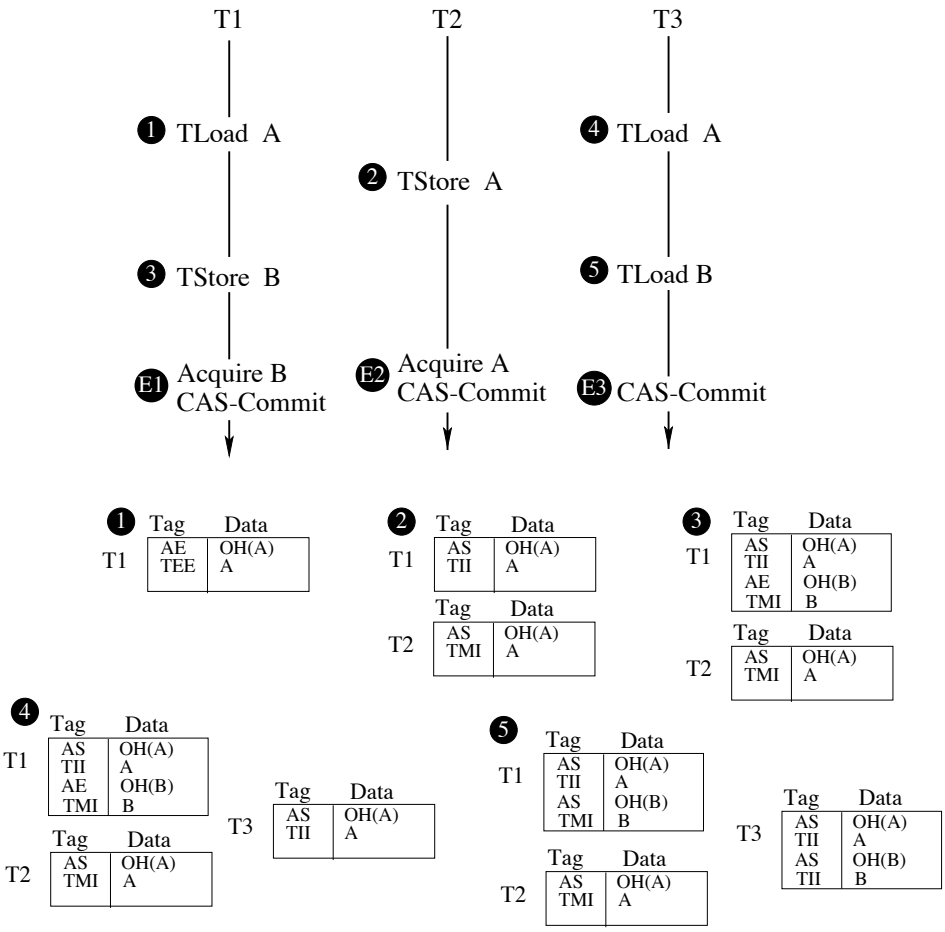


Figure 5: Execution of Transactions. Top: interleaving of accesses in three transactions, with lazy acquire. Bottom: Cache tag arrays at various event points. (OH(x) is used to indicate the header of object x.)

At ❹ transaction T3 performs a *TLoad* on object A. Since T2 holds the line in *TMI*, it asserts the T signal in response to T3's BusRd message. This causes T3 to load the line in *TII*, giving it access only until it commits or aborts (at which point it loses the protection of software conflict detection). Prior to the *TLoad*, RTM software will have *ALoad*ed A's header into T3's cache during the *open*, causing T2 to assert the S signal and to drop its own copy of the header to *AS*. If T2 acquires A while T3 is active, its BusRdX on A's header will cause an invalidation in T3's cache and thus an immediate abort of T3.

Event ❺ is similar to ❹, and B is also loaded in *TII*.

We now consider the ordering of events Ⓔ1, Ⓔ2, and Ⓔ3.

1. **E1 happens before E2 and E3**: When T1 acquires B's header, it invalidates the line in T3's cache. This causes T3 to abort. T2, however, can commit. When it retries, T3 will see the new value of A from T1's commit.

2. **E2 happens before E1 and E3**: When T2 acquires A's header, it aborts both T1 and T3.

3. **E3 happens before E1 and E2**: Since T3 is only a reader of objects, and has not been invalidated by writer acquires, it commits. T2 can similarly commit, if E1 happens before E2, since T1 is a reader of A. Thus, the ordering **E3, E1, E2** will allow all three transactions to commit. TCC would also

admit this scenario, but none of the other hardware schemes mentioned in Sections 1 or 3 would do so, because of eager conflict detection. RTM enforces consistency with a single BusRdX per object header. In contrast, TCC must broadcast all speculatively modified lines at commit time.

5 RTM Software

In the previous section we presented the TMESI hardware, which enables flexible policy making in software. With a few exceptions related to the interaction of hardware and software transactions, policy is set entirely in software, with hardware serving simply to speed the common case. Transactions that overflow hardware the size or associativity of the cache are executed entirely in software, while ensuring interoperability with concurrent hardware transactions. Software transactions are essentially *unbounded* in space and time.

In the subsections below we first describe the metadata that allows hardware and software transactions to share a common set of objects, thereby combining fast execution in the common case with unbounded space in the general case. We then describe mechanisms used to ensure consistency when handling immediate aborts. Finally, we present context-switching support for transactions with unbounded time.

5.1 Transactions Unbounded in Space

The principal metadata employed by RTM are illustrated in Figure 6. The object header has five main fields: a pointer to the most recent writer transaction, a serial number, pointers to one or two clones of the object, and a head pointer for a list of software transactions currently reading the object. (The need for explicitly visible software readers, explained in Section 4.3, is the principal policy restriction imposed by RTM. Without such visibility [and immediate aborts] we see no way to allow software transactions to interoperate with hardware transactions that may modify objects in place.)

The low bit of the transaction pointer in the object header is used to indicate whether the most recent writer was a hardware or software transaction. If the writer was a software transaction and it has committed, then the "new" object is current; otherwise the "old" object is current (recall that hardware transactions make updates in place). Writers acquire a header by updating it atomically with a *Wide-CAS* instruction.[5] To first approximation, RTM object headers combine DSTM-style *TMObject* and *Locator* fields [10].

Serial numbers allow RTM to avoid dynamic memory management by reusing transaction descriptors. When starting a new transaction, a thread increments the number in the descriptor. When acquiring an object, it sets the number in the header to match. If, at *open* time, a transaction finds mismatched numbers in the object header and the descriptor to which it points, it interprets it as if the header had pointed to a matching *committed* descriptor. On abort, a thread must erase the pointers in any headers it has acquired. As an adaptive performance optimization for read-intensive applications, a reader that finds a pointer to a *committed* descriptor replaces it with a sentinel value that saves subsequent readers the need to dereference the pointer.

For hardware transactions, the in-place update of objects and reuse of transaction descriptors eliminate the need for dynamic memory management. Software transactions, however, must still allocate and deallocate clones and entries for explicit reader lists. For these purposes RTM employs a lightweight, custom storage manager. In a software transaction, acquisition installs a new data object in the "New Object" field, erases the pointer to any data object O that was formerly in that field, and reclaims the space for O. Immediate aborts ensure that dangling references will never be used.

[5] RSTM emulates *Wide-CAS* using a test-and-set lock around a small critical section. Occasional preemption within this section accounts for the slight upward turn in the RSTM curve of Figure 2 for thread counts beyond 16 (the number of processors in the machine).

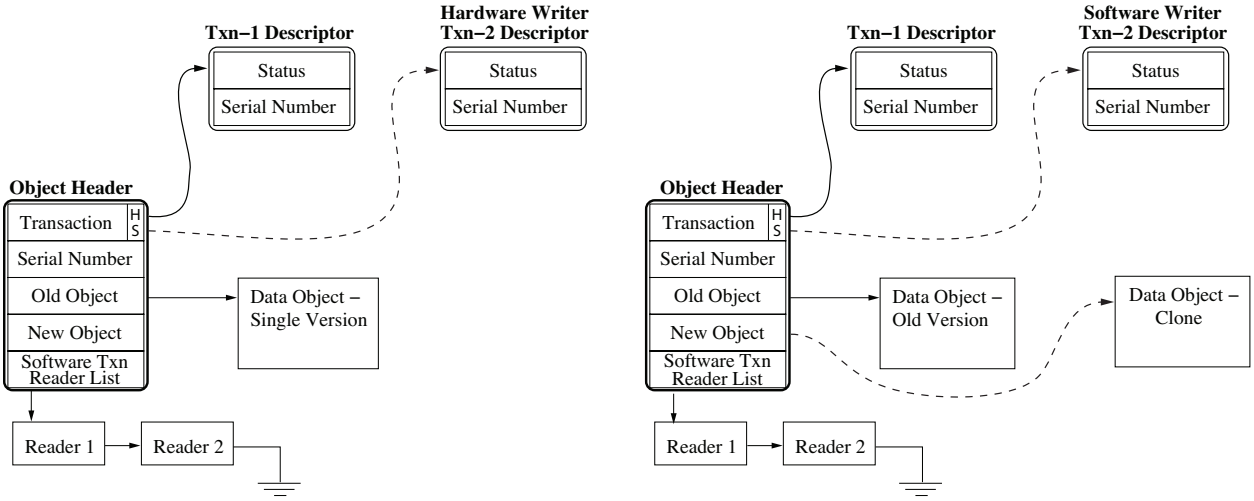


Figure 6: RTM metadata structure. On the left a hardware transaction is in the process of acquiring the object, overwriting the transaction pointer and serial number fields. On the right a software transaction will also overwrite the New Object field. If a software transaction acquires an object previous owned by a committed software transaction, it overwrites ⟨Old Object, New Object⟩ with ⟨New Object, Clone⟩. Several software transactions can work concurrently on their own object clones prior to *acquire* time, just as hardware transactions can work concurrently on copies buffered in their caches.

5.2 Deferred Aborts

While aborts must be synchronous to avoid any possible data inconsistency, there are times when they should *not* occur. Most obviously, they need to be postponed whenever a transaction is currently executing RTM system code (e.g., memory management) that needs to run to completion. Within the RTM library, code that should not be interrupted is bracketed with `BEGIN_NO_ABORT...END_NO_ABORT` macros. These function in a manner reminiscent of the preemption avoidance mechanism of SymUnix [2]: `BEGIN_NO_ABORT` increments a counter, inspected by the standard abort handler installed by RTM. If an abort occurs when the counter is positive, the handler sets a flag and returns. `END_NO_ABORT` decrements the counter. If it reaches zero and the flag is set, it clears the flag and reinvokes the handler.

Transactions are permitted to perform nontransactional operations for logging, profiling, debugging, or similar purposes. Occasionally these must be executed to completion (e.g. because they acquire and release an I/O library lock). For this purpose, RTM makes `BEGIN_NO_ABORT` and `END_NO_ABORT` available to user code. In database terminology, code within these macros, provided it generates no exception itself, is guaranteed atomicity but not isolation.

5.3 Transactions Unbounded in Time

To permit transactions of unbounded duration, RTM must ensure that software transactions survive a context switch, and that they be aware, on wakeup, of any significant events that transpired while they were asleep. Toward these ends, RTM requires that the scheduler be aware of the location of each thread's transaction descriptor, and that for software transactions this descriptor contain, in addition to the information shown in Figure 6, (1) and indication of whether the transaction is running in hardware or in software, and (2) for software transactions, the transaction pointer and serial number of any object currently being cloned.

The scheduler performs the following actions.

1. To avoid confusing the state of multiple transactions, the scheduler executes an *Abort* instruction on every context switch, thereby clearing both T and A states out of the cache. A software transaction can resume execution when rescheduled. A hardware transaction, on the other hand, is aborted. The scheduler modifies its state so that it will wake up in its abort handler when rescheduled.

2. As previously noted, interoperability between hardware and software transactions requires that a software transaction *ALoad* its transaction descriptor, so it will notice immediately if aborted by another transaction. When resuming a software transaction, the scheduler re-*ALoad*s the descriptor.

3. A software transaction may be aborted while it is asleep. At preemption time the scheduler notes whether the transaction's status is currently *active*. On wakeup it checks to see if this has been changed to *aborted*. If so, it modifies the thread's state so that it will wake up in its abort handler.

4. A software transaction must *ALoad* the header of any object it is cloning. On wakeup the scheduler checks to see whether that object (if any) is still valid (by comparing the current and saved serial numbers and transaction pointers). If not, it arranges for the thread to wake up in its handler. If so, it re-*ALoad*s the header.

These rules suffice to implement unbounded software transactions that interoperate correctly with (bounded) hardware transactions.

6 Conclusions and Future Work

We have described a transactional memory system, RTM, that uses hardware to accelerate transactions managed by a software protocol. RTM is 100% source-compatible with the RSTM software TM system, providing users with a gentle migration path from legacy machines. We believe this style of hardware/software hybrid constitutes the most promising path forward for transactional programming models.

In contrast to previous transactional hardware protocols, RTM

1. requires only one new bus signal, no new bus messages, and no hardware consensus protocol.

2. requires, for fast path operation, that only *speculatively written* lines be buffered in the cache.

3. falls back to software on overflow, or at the direction of the contention manager, thereby accommodating transactions of effectively unlimited size and duration.

4. allows software transactions to interoperate with ongoing hardware transactions.

5. supports immediate aborts of remote transactions, even if their transactional state has overflowed the cache.

6. permits read-write and write-write sharing, when desired by the software protocol.

7. permits "leaking" of information from inside aborted transactions, for logging, profiling, debugging, and similar purposes.

8. performs contention management entirely in software, enabling the use of adaptive and application-specific protocols.

We are currently nearing completion of an RTM implementation for the GEMS SIMICS/SPARC-based simulation infrastructure [17]. Since the cost of metadata management is linear in the number of objects *open*ed, it is reasonable to expect rather modest performance differences in comparison to full hardware TM, differences that we expect will be compensated for by flexible policy. In future work, we plan to explore a variety of topics, including other styles of RTM software (e.g., word-based); nested transactions; gradual fall-back to software, with ongoing use of whatever fits in cache; context tags for simultaneous transactions in separate hardware threads; and realistic real-world applications. We also hope to arrange a (simulated) head-to-head comparison with one or more hardware TM systems.

Acknowledgments

We are grateful to the Scalable Synchronization Group at Sun Microsystems Laboratories for providing access to the DSTM code, on which ASTM is based, and to the Multifacet group at the University of Wisconsin–Madison for providing access to the GEMS simulation infrastructure, currently being used to evaluate RTM.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proceedings of the Eleventh International Symposium on High Performance Computer Architecture*, pages 316–327, San Francisco, CA, February 2005.

[2] J. Edler, J. Lipkis, and E. Schonberg. Process Management for Highly Parallel UNIX Systems. In *Proceedings of the USENIX Workshop on Unix and Supercomputers*, Pittsburgh, PA, September 1988.

[3] K. Fraser and T. Harris. Concurrent Programming Without Locks. Submitted for publication, 2004.

[4] R. Guerraoui, M. Herlihy, M. Kapalka, and B. Pochon. Robust Contention Management in Software Transactional Memory. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, October 2005. In conjunction with OOPSLA'05.

[5] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Distributed Computing*, Las Vegas, Nevada, August 2005.

[6] L. Hammond, V. Wong, M. Chen, B. Hertzberg, B. Carlstrom, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the Thirty-First International Symposium on Computer Architecture*, München, Germany, June 2004.

[7] T. Harris, S. Marlow, S. P. Jones, and M. Herlihy. Composable Memory Transactions. In *Proceedings of the Tenth ACM Symposium on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

[8] C. Heriot, V. Marathe, M. F. Spear, A. Acharya, S. Dwarkadas, D. Eisenstat, W. N. Scherer III, M. L. Scott, and A. Shriraman. Low-Overhead Software Transactional Memory for C++. Technical Report, Department of Computer Science, University of Rochester, January 2006. In preparation.

[9] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of the Twenty-Third International Conference on Distributed Computing Systems*, Providence, RI, May, 2003.

[10] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software Transactional Memory for Dynamic-sized Data Structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 92–101, Boston, MA, July 2003.

[11] M. Herlihy. SXM: C# Software Transactional Memory. Unpublished manuscript, Brown University, May 2005.

[12] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[13] M. Herlihy. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

[14] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the Twentieth International Symposium on Computer Architecture*, pages 289–300, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, December 1992.

[15] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. In *Proceedings of the Nineteenth International Symposium on Distributed Computing*, Cracow, Poland, September 2005.

[16] J. F. Martínez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, San Jose, CA, October 2002.

[17] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. In *ACM SIGARCH Computer Architecture News*, September 2005.

[18] M. M. Michael. Scalable Lock-Free Dynamic Memory Allocation. In *Proceedings of the SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 35–46, Washington, DC, June 2004.

[19] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth ACM Symposium on Principles of Distributed Computing*, pages 267–275, Philadelphia, PA, May 1996.

[20] M. Moir and others. Hybrid Transactional Memory. Unpublished manuscript, Sun Microsystems Laboratories, Burlington, MA, July 2005.

[21] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the Twelfth International Symposium on High Performance Computer Architecture*, Austin, TX, February 2006.

[22] E. Moss and T. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, October 2005. In conjunction with OOPSLA'05.

[23] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the Thirty-Fourth International Symposium on Microarchitecture*, Austin, TX, December 2001.

[24] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, San Jose, CA, October 2002.

[25] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proceedings of the Thirty-Second International Symposium on Computer Architecture*, Madison, WI, June 2005.

[26] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. A High Performance Software Transactional Memory System for a Multi-Core Runtime. Submitted for publication, October 2005.

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

[28] W. N. Scherer III and M. L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the Twenty-Fourth ACM Symposium on Principles of Distributed Computing*, Las Vegas, NV, July 2005.

[29] N. Shavit and D. Touitou. Software Transactional Memory. *Distributed Computing*, 10(2):99–116, February 1997. Originally presented at the *Fourteenth ACM Symposium on Principles of Distributed Computing*, August 1995.

[30] H. Sundell and P. Tsigas. NOBLE: A Non-Blocking Inter-Process Communication Library. In *Proceedings of the Sixth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington, DC, March 2002. Also TR 2002-02, Chalmers University of Technology and Göteborg University, Göteborg, Sweden.

[31] R. K. Treiber. Systems Programming: Coping with Parallelism. RJ 5118, IBM Almaden Research Center, April 1986.

[32] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional Monitors for Concurrent Objects. In *Proceedings of the Eighteenth European Conference on Object-Oriented Programming*, pages 519–542, June 2004.