Hardware Acceleration of Software Transactional Memory *

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

Department of Computer Science, University of Rochester
{ashriram,vmarathe,sandhya,scott,eisen,cheriot,scherer,spear}@cs.rochester.edu

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 software overheads of data copying, garbage collection, and validation, without introducing any global consensus algorithm in the cache coherence protocol (a commit is allowed to perform using only a few cycles of completely local operation). 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: they are vulnerable to deadlock, priority inversion, and arbitrary delays due to preemption. In addition, while coarse-grain lock-based algorithms are easy to understand, they limit concurrency. Fine-grain locking algorithms are thus often required, but these are difficult to design, debug, maintain, and understand.

Ad hoc *nonblocking* algorithms [15, 16, 24, 25] solve the semantic problems of locks by ensuring that forward progress is never precluded by the state of any thread or set of threads. They 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 [8], 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* [7], 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. Rajwar and Goodman's Transactional Lock Removal (TLR) [19, 20] 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. Martinez and Torrellas [13] 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. [21] describe a related mechanism (VTM) that uses hardware to *virtualize* transactions across both space and time. Moore et al. [18] 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. [5] 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.

While we see great merit in all these proposals, 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

* Presented at TRANSACT: the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, held in conjunction with PLDI, Ottawa, Ontario, Canada, June 2006.

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

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 may not run on legacy machines.

Moir [17] describes a design philosophy for a hybrid transactional memory system in which hardware makes a "best effort" attempt to complete transactions, falling back to software when necessary. The goal of this philosophy is to be able to leverage almost any reasonable hardware implementation. Kumar et al. [10] describe a specific hardware–software hybrid that builds on the software system of Herlihy et al. [6]. Unfortunately, this system still embeds significant policy in silicon. It assumes, for example, that conflicts are detected as early as possible (pessimistic concurrency control), disallowing either read-write or write-write sharing. Previous published papers [11, 22] reveal performance differences across applications of 2X – 10X *in each direction* for different approaches to contention management, metadata organization, and eagerness of conflict detection (i.e., write-write sharing). It is clear that no one knows the "right" way to do these things; it is likely that there is no one right way.

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, allowing flexible policy choice; 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 [12]. 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 pure hardware proposals, it requires no global consensus algorithm in the cache coherence protocol, no snapshotting of processor state, and message traffic comparable to that of a regular MESI coherence protocol. 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.) Finally, we employ a software *contention manager* [22, 23] 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, these data (and a small amount of nonspeculative metadata) are *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 the proposals of Moir and of Kumar et al., 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 the hybrid proposals, but also to TLR, LTM, VTM, and LogTM, 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 implementations could also be devised for directory-based machines (this is one topic of our ongoing work). TMESI could also be used with a variety of software systems other than RTM. We do not describe such extensions here.

Section 2 provides more detailed background and motivation for RTM, including an introduction to software TM in general, a characterization of its dominant costs, and an overview of how TMESI and RTM address them. Section 3 describes TMESI in detail, including its instructions, its states and transitions, and the mechanism used to detect conflicts and abort remote transactions. Section 4 then describes the RTM software that leverages this hardware support. Our choice of concrete policies reflects experimentation with several software TM systems, and incorporates several forms of dynamic adaptation to the offered workload. We conclude in Section 5 with a summary of contributions, a brief description of our simulation infrastructure (currently nearing completion), and a list of topics for future research.

2. RTM Overview

Software TM systems display a wide variety of policy and implementation choices. Our RSTM system [12] 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*.

2.1 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.[1] Only those objects explicitly identified as `Shared` are protected by the TM system. `Shared` objects cannot be accessed simultaneously in both transactional and non-transactional mode. 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* [18].) If a transaction aborts, its clones are discarded. RTM currently supports nested transactions only via subsumption in the parent.

Figure 1 contains an example of C++ RTM code to insert an element in a singly-linked sorted list of integers. The API is in-

[1] We do require that each object reside in its own set of cache lines.

```
void intset::insert(int val) {
  BEGIN_TRANSACTION;
    const node* previous = head->open_RO();
        // points to sentinel node
    const node* current = previous->next->open_RO();
        // points to first real node
    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
        previous->open_RW()->next = new Shared<node>(n);
    }
  END_TRANSACTION;
}
```

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

herited from our RSTM system [12], which runs on legacy hardware (space limitations preclude a full presentation here). The `rtm::Shared<T>` template class provides an opaque wrapper around transactional objects. Several crucial methods, including `operator new`, are provided by `rtm::Object<T>`, from which `T` must be derived. Within a transaction, bracketed by `BEGIN_TRANSACTION` and `END_TRANSACTION` macros, the `open_RO()` and `open_RW()` methods can be used to obtain `const T*` and `T*` pointers respectively. The `shared()` method performs the inverse operation, returning a pointer to the `Shared<T>` with which `this` is associated. Our code traverses the list from the head, opening objects in read-only mode, until it finds the proper place to insert the element. It then re-opens the object whose `next` pointer it needs to modify in read-write mode. To make such upgrades convenient, `Object<T>::open_RW` returns `shared()->open_RW()`.

2.2 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 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* [18]. They are also restricted to contention management algorithms simple enough (and static enough) to be implemented in hardware on a cache miss.

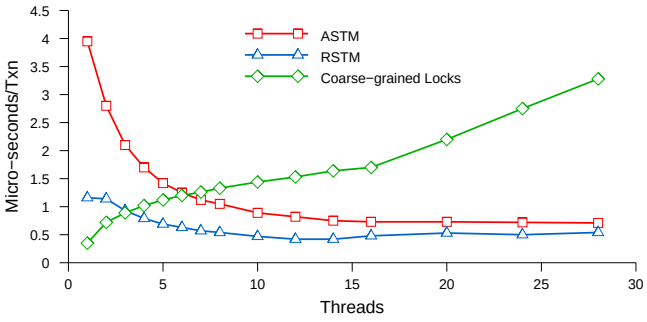

Figure 2. Performance scaling of RSTM, ASTM, and coarse-grain locking on a hash table microbenchmark.

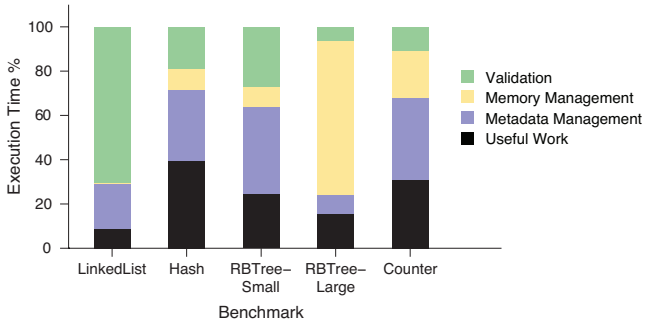

Figure 3. Cost breakdown for RSTM on a single processor, for five different microbenchmarks.

Work by Marathe et al. [11] 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 (we employ their adaptive heuristics). Likewise, work by Scherer et al. [22, 23] 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. In both these dimensions, RTM provides significantly greater flexibility than pure hardware TM proposals.

2.3 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 [11] to match the faster of Sun's DSTM [6] 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 256 buckets, and all values are taken from the range 0–255, leading to a steady-state average of 0.5 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 3× faster than RSTM. We need about 4 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 40 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 2.2. Suppose transaction R opens objects X and Y in read-only 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 [4] 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 with heavy contention; neither clearly wins in the common middle ground [23]. RSTM supports both options; the results in Figures 2 and 3 were collected using incremental validation.

2.4 Hardware Support

RTM uses hardware support (the TMESI protocol) 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. 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 of any given cache line 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, and without precluding read-write and write-write speculation.

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 (currently up to 4) adjacent words atomically

Table 1. ISA Extensions for RTM.

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 (all within the same cache line).

A transaction could, in principle, use hardware support for certain objects and not for others. For the sake of simplicity, our initial implementation of RTM 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.)

3. 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.

3.1 ISA Extensions

RTM requires eight new hardware instructions, listed in Table 1.

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 3.3). 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 3.2.

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 don't have room for all their object headers in the cache (more on this in Section 3.3). 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 [11].

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 [6, 11]. 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, speculatively read lines (the transactional and abort-on-invalidate lines) are untagged and revert to their corresponding MESI states. If the CAS succeeds, speculatively written lines become visible to the coherence protocol and begin responding to coherence messages. If the CAS fails, speculatively written lines are invalidated, and control transfers to the location registered by *SetHandler*. 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 allows a compare-and-swap across multiple contiguous locations (within a single cache line). As in Itanium's `cmp8xchg16` instruction [9], if the first two words at location A match their "old" values, all words are swapped with the "new" values (loaded into contiguous registers). Success is detected by comparing old and new values in the registers. *Wide-CAS* is intended for fast update of object headers.

3.2 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. The validation portion of a *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 "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).

3.2.1 State transitions

Figure 4 contains a state 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. No writeback or flush is required since the line is not dirty. Even during a transaction, silent eviction and re-read is not a problem because software ensures that no writer can commit unless it first aborts the reader. 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. 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 (writing processors) continue to transition to *TMI*; only one of the writers will eventually commit, resulting in only one of the caches reverting to *M* state. Lines originally in *M* or *TMM* state require a writeback on the first *TStore* to ensure that memory has the latest non-speculative value.

Among hardware TM systems, only TCC and RTM support read-write and write-write sharing; all the other schemes mentioned in Sections 1 and 2 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 [11]. 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.

3.2.2 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

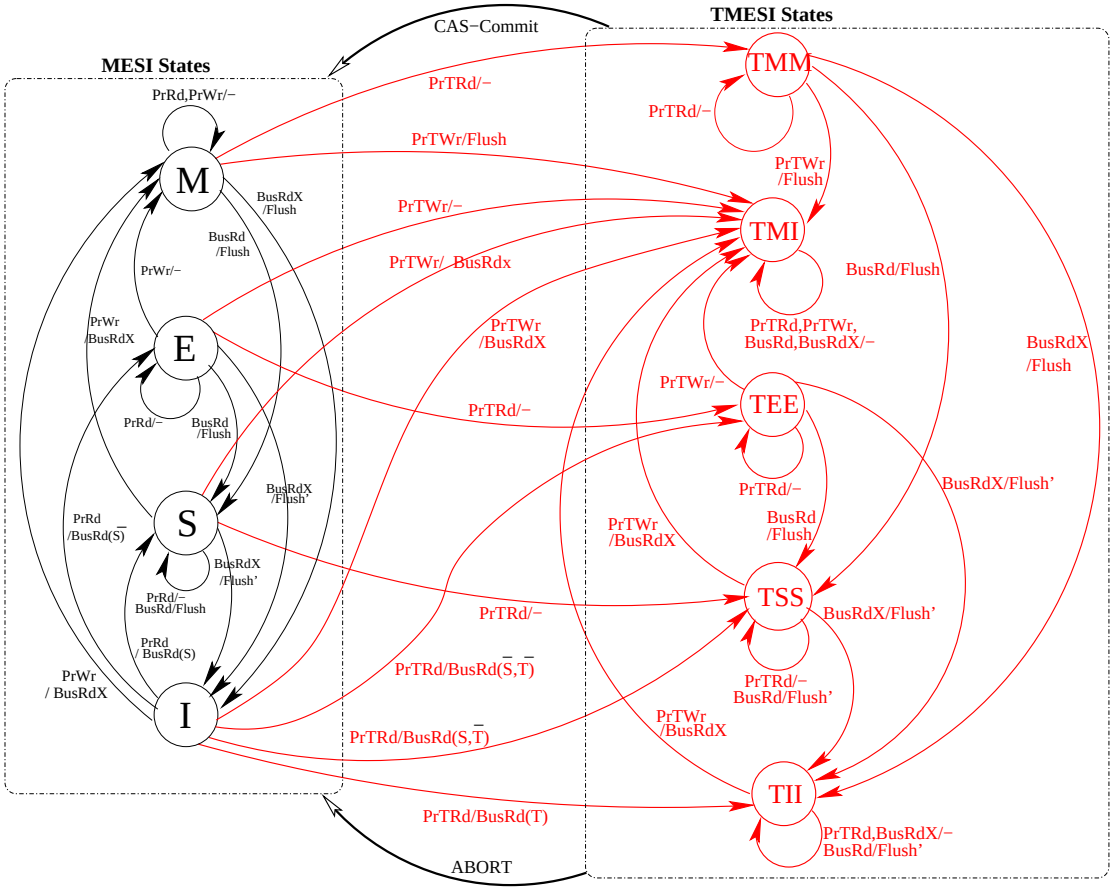

Figure 4. TMESI Protocol. Dashed boxes enclose the MESI and TMESI subsets of the state space. All TMESI lines revert to MESI states in the wake of a *CAS-Commit* or *Abort*. Specifically, the 2nd and 3rd letters of a TMESI state name indicate the MESI state to which to revert on commit or abort, respectively. 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".

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.

*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.

3.2.3 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 2.

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 2. Tag array encoding. Interpretations of the bits (right) give rise to 15 valid encodings of the 12 TMESI states.

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.

3.3 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 2.3, 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.

If the procesor is in user mode, delivery of the abort takes the form of a spontaneous subroutine call, thereby avoiding kernel-user crossing overhead. 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. If the processor is in kernel mode, delivery takes the form of an interrupt vectored in the usual way. If the processor is executing at interrupt level when an abort occurs, delivery is deferred until the return from the interrupt. Transactions may not be used from within interrupt handlers. Both kernel and user programs are allowed to execute hardware transactions, however, 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. 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.

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.[2]

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. Memory always has an up-to-date non-speculative copy of data, which it returns; lines in TMI state do not respond to read or write requests from the bus, thereby allowing readers from both hardware and software transactions to work with the stable non-speculative copy. 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.

3.4 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.

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 E1, E2, and E3.

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

[2] 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.

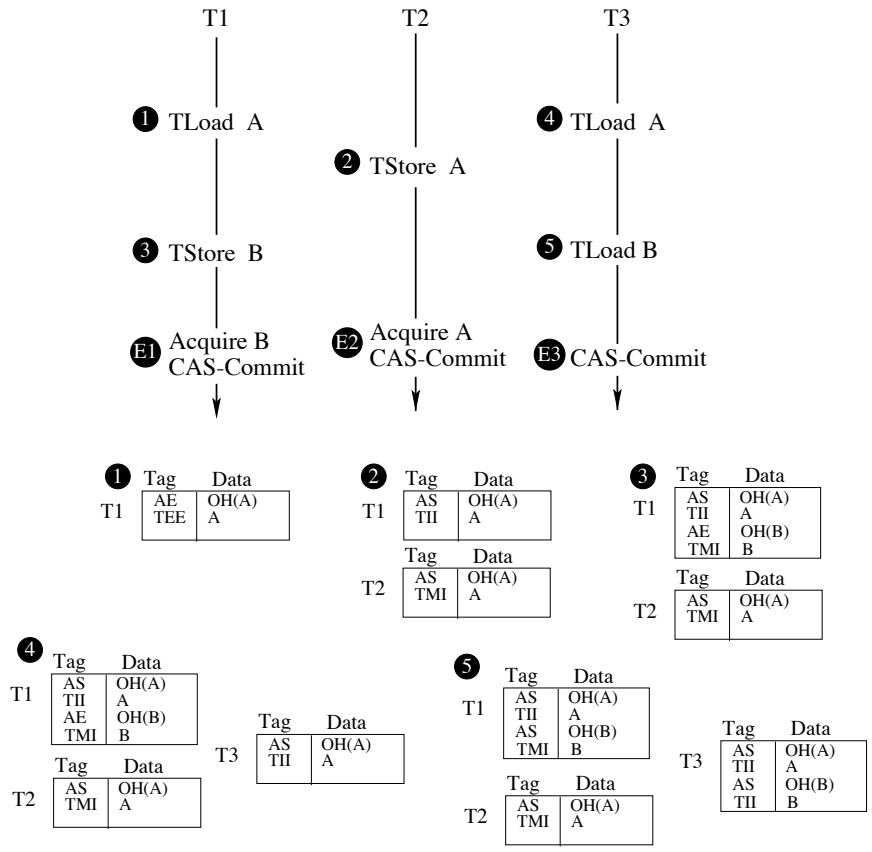

Figure 5. Execution of Transactions. Top: interleaving of accesses in three transactions, with lazy acquire. Bottom: Cache tag arrays at various event points. (OH(x) is used to indicate the header of object x.)

Sections 1 or 2 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.

4. 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 due to 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.

4.1 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 3.3, 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 least significant 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. To first approximation, RTM object headers combine DSTM-style *TMObject* and *Locator* fields [6].[3]

Serial numbers allow RTM to avoid dynamic memory management for transaction descriptors by reusing them. 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

[3] RSTM avoids the need for WCAS by moving much of an object's metadata into the data object instance, rather than the header. In particular, it arranges for the newer data object to point to the older [12]. We keep all metadata in the header in RTM to minimize the need for `ALoaded` cache lines.

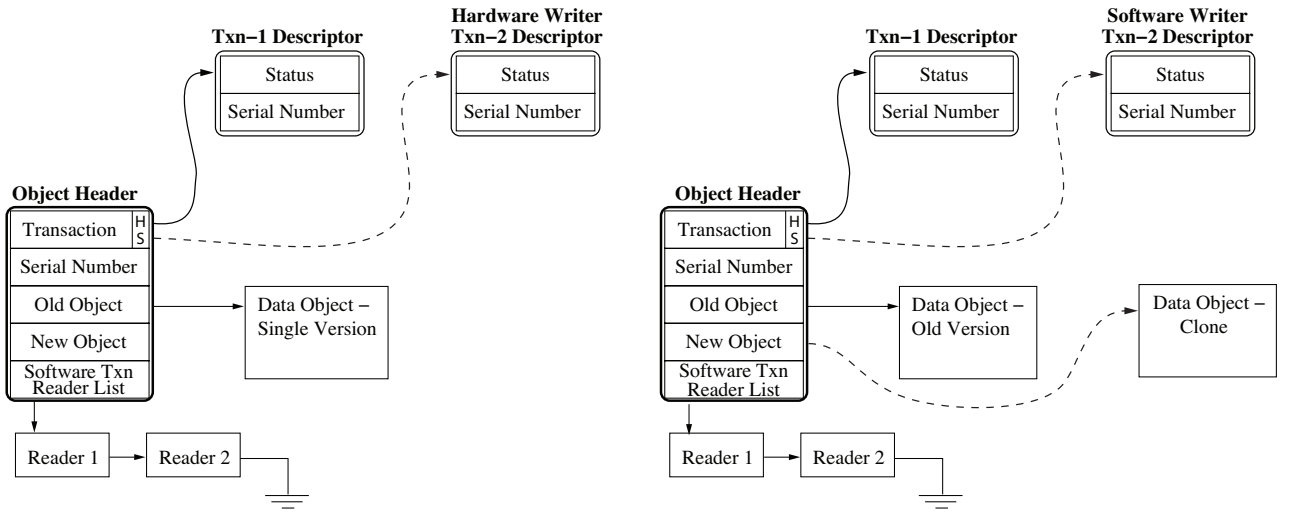

Figure 6. RTM metadata structure. On the left a hardware transaction is in the process of acquiring the object, overwriting the transaction pointer and serial number fields. On the right a software transaction will also overwrite the New Object field. If a software transaction acquires an object previously 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.

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 within the TM runtime. 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 preclude the use of dangling references.

4.2 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 may 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.

4.3 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 this descriptor contain, in addition to the information shown in Figure 6, (1) an 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.

5. Conclusions and Future Work

We have described a transactional memory system, RTM, that uses hardware to accelerate transactions managed by a software proto-

col. 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 and no hardware consensus protocol or extra traffic at commit time.
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 using the GEMS SIMICS/SPARC-based simulation infrastructure [14]. In future work, we plan to explore a variety of topics, including other styles of RTM software (e.g., word-based); hardware (e.g., directory-based protocols); 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.

References

[1] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded Transactional Memory. In *Proc. of the 11th Intl. Symp. on High Performance Computer Architecture*, San Francisco, CA, Feb. 2005.

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

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

[4] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management in SXM. In *Proc. of the 19th Intl. Symp. on Distributed Computing*, Cracow, Poland, Sept. 2005.

[5] 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 *Proc. of the 31st Intl. Symp. on Computer Architecture*, München, Germany, June 2004.

[6] 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*, Boston, MA, July 2003.

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

[8] M. Herlihy and J. E. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Intl. Symp. on Computer Architecture*, San Diego, CA, May 1993. Expanded version available as CRL 92/07, DEC Cambridge Research Laboratory, Dec. 1992.

[9] Intel Corporation. Intel Itanium Architecture Software Developers Manual. Revision 2.2, Jan. 2006.

[10] 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.

[11] 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.

[12] 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 Languages, Compilers, and Hardware Support for Transactional Computing*, Ottawa, ON, Canada, July 2006. Held in conjunction with PLDI 2006. Expanded version available as TR 893, Dept. of Computer Science, Univ. of Rochester, Mar. 2006.

[13] J. F. Martinez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly Parallel Applications. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2002.

[14] 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. Multifacets General Execution-driven Multiprocessor Simulator (GEMS) Toolset. In *ACM SIGARCH Computer Architecture News*, Sept. 2005.

[15] M. M. Michael. Scalable Lock-Free Dynamic Memory Allocation. In *Proc. of the SIGPLAN 2004 Conf. on Programming Language Design and Implementation*, Washington, DC, June 2004.

[16] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proc. of the 15th ACM Symp. on Principles of Distributed Computing*, Philadelphia, PA, May 1996.

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

[18] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based Transactional Memory. In *Proc. of the 12th Intl. Symp. on High Performance Computer Architecture*, Austin, TX, Feb. 2006.

[19] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proc. of the 34th Intl. Symp. on Microarchitecture*, Austin, TX, Dec. 2001.

[20] R. Rajwar and J. R. Goodman. Transactional Lock-Free Execution of Lock-Based Programs. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 2002.

[21] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing Transactional Memory. In *Proc. of the 32nd Intl. Symp. on Computer Architecture*, Madison, WI, June 2005.

[22] 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. Johns, NL, Canada, July 2004.

[23] 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.

[24] H. Sundell and P. Tsigas. NOBLE: A Non-Blocking Inter-Process Communication Library. In *Proc. of the 6th Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Washington, DC, Mar. 2002. Also TR 2002-02, Chalmers Univ. of Technology and Göteborg Univ., Göteborg, Sweden.

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