Nonblocking Transactions Without Indirection Using Alert-on-Update *

Michael F. Spear, Arrvindh Shriraman, Luke Dalessandro,
Sandhya Dwarkadas, and Michael L. Scott

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

ABSTRACT

Nonblocking implementations of software transactional memory (STM) typically impose an extra level of indirection when accessing an object; some researchers have claimed that the cost of this indirection outweighs the semantic advantages of nonblocking progress guarantees. We consider this claim in the context of a simple hardware assist, alert-on-update *(AOU), which allows a thread to request immediate notification if specified line(s) are replaced or invalidated in its cache.*

We show that even a single AOU line allows us to construct a simple, nonblocking STM system without extra indirection. At the same time, we observe that per-load validation operations, required for intra-object consistency in both the new system and in lock-based (blocking) STM, at least partially negate the resulting performance gain. Moreover, inter-object consistency checks, also required in both kinds of systems, remain the dominant cost for transactions that access many objects. We therefore present a second nonblocking STM system that uses multiple AOU lines (one per accessed object) to eliminate validation overhead entirely, resulting in a nonblocking, zero-indirection STM system that outperforms competing systems by as much as a factor of 2.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming—Parallel programming C.1.2 [Processor Architectures]: Multiprocessors

General Terms: Performance, Design, Languages

Keywords: Software transactional memory, Obstruction freedom, Event-based systems

1. INTRODUCTION

Although multicore processors are rapidly gaining acceptance in the commodity computer market, few existing programs are capable of exploiting thread-level parallelism. Lock-based code remains difficult to write correctly, and much evidence suggests that it is unreasonable to require all programmers to specify fine-grained locks explicitly. A growing consensus holds that transactions, long the foundation of database concurrency control, are the most promising near-term means to simplify the construction of multithreaded programs. To first approximation, the programmer specifies the blocks of code that must appear to execute atomically, and the underlying system assumes responsibility for running these in parallel with one another whenever possible.

Pure software implementations of transactional memory (STM) can be divided into two main camps: those that use locks under the hood (hidden from the user) and those that are nonblocking (typically obstruction-free [10, 11]). Several groups have found lock-based implementations to be faster in practice [2, 4, 9, 18], but nonblocking implementations have other advantages: they are immune to priority inversion in event-based code, and to performance anomalies caused by inopportune preemption or page faults. They also tend to avoid *convoying*, in which threads repeatedly contend for the same resources because waiting at locks has encouraged them to "fall into step" with one another. Finally, nonblocking implementations ensure consistency even if a thread can die in the middle of a transaction—a potentially compelling advantage if data are shared among threads with independent failure modes.

Indirection v. Locking. Lock-based STM systems typically modify data "in place". A transaction that aborts because of conflict with a peer may need to perform a certain amount of "cleanup" (e.g. application of an *undo log*) before the peer can continue. Similarly, a transaction may sometimes be unabortable (e.g., while applying a *redo log* to transfer updates from a private buffer to the master copy of the data). Lock-based STM systems include LibLTx [4], McRT [1, 17, 18], the Microsoft Bartok system [9], and TL2 [2].

While it is possible to build a nonblocking STM system with in-place updates [7, 14], such systems are generally quite complex. A more common approach is to introduce an extra level of indirection (Figure 1): all modifications to objects are performed on private copies; when a transaction commits, it uses a single `compare-and-swap` (`CAS`) on the transaction descriptor (to which all objects point) to logically redirect all relevant pointers to the corresponding private copies, which then become public and immutable. Indirection-based nonblocking STM systems include DSTM [11], OSTM [5], SXM [6], ASTM [12], and RSTM [22].

The indirection of nonblocking systems ensures that committing and aborting are both lightweight operations, and that objects read during a transaction are immutable. Unfortunately, the extra indirection also increases both capacity and coherence misses in the cache, by increasing the number of lines required to represent an

*This work was supported in part by NSF grants CCR-0204344, CNS-0411127, CNS-0615139, and CNS-0509270; an IBM Faculty Partnership Award; equipment support from Sun Microsystems Laboratories; and financial support from Intel and Microsoft.

SPAA'07, June 9–11, 2007, San Diego, California, USA.

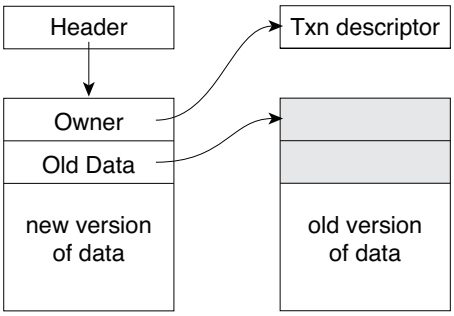

Figure 1: Metadata of RSTM, a representative nonblocking STM system. All references go through the indirection pointer (header), which is easily changed to install a new version.

object and the number of lines that are modified when changes to an object are committed.

The impact of these misses can be dramatic. On our local Sun-Fire machine, even a one-thread, coarse-lock implementation of the benchmarks presented in Section 4 slows down by as much as 90% (over 20% on average) when we introduce an extra level of indirection. This result is consistent with the work of Ennals [4] and of Dice et al. [3], who find cache misses to be a major factor in the speed of STM. While coherence misses (satisfied from a shared on-chip cache) will be cheaper on a chip multiprocessor than they have been on traditional SMPs, they are unlikely to decrease in cost over time, and large-scale machines will continue to be built from multiple processor chips, with very high cache miss penalties.

We use lightweight hardware support to eliminate indirection while retaining nonblocking semantics. In the process, we lose the guarantee of immutability for objects read by transactions. As in lock-based STM, we compensate by arranging for transactions to validate an object (double-check its version number) every time they read one of its fields. As discussed in previous work [22], validation is essential for correctness: absent compiler-enforced sandboxing of every operation that cannot be rolled back on abort, failure to validate can lead to arbitrary erroneous behavior in doomed transactions, breaking the programming model.

Cache-Thrashing v. Quadratic Validation. Whether a system uses locks or indirection, it is also necessary to verify mutual consistency across objects over time. This validation may be implemented by attaching an explicit *reader list* to each object, or by performing *incremental validation* of private lists of objects.

Systems that employ reader lists (also known as *visible readers*) require transactions to modify per-object metadata before the first read of an object, usually with an expensive atomic operation such as `CAS`. Some systems also require transactions to modify the metadata again at the end of the transaction. In return, the reading transaction is guaranteed that no other transaction will change the object without first notifying the reader. When a new object is accessed, the reader must only ensure that it hasn't been notified, via a constant-time comparison to a single location. The total cost to read n objects in this manner is $n \times (C_{\textit{list-add}} + C_{\textit{list-remove}} + C_{\textit{check-self}})$. This asymptotic $O(n)$ cost carries a high constant ($C_{\textit{list-add}}$) due to cache misses: with a random access pattern and any concurrency, the list is likely to reside exclusively in another thread's cache when the current thread adds itself to the list.

An STM system that does not use reader lists (also known as *invisible readers*) can avoid the cache misses associated with metadata updates. Every time a new object is encountered, however, the active transaction T must verify that no other transaction has modified any object previously read by T. Thus to read n objects, the transaction must perform $\sum_{i=1}^{n} i = O(n^2)$ validation operations. In addition, information about the n objects must be stored locally to the transaction. For small transactions, this $O(n)$ bookkeeping and $O(n^2)$ validation overhead is still smaller than the $O(n)$ overhead of a reader list [13, 22].

Contributions and Roadmap. In recent work we have proposed a simple hardware mechanism, *alert-on-update* (AOU), that allows a thread to request immediate notification if certain lines are evicted from its cache. We have described [21, 23] how this mechanism can be used to detect transaction conflicts without requiring either visible reader lists (with their attendant cache misses) or incremental validation (with its $O(n^2)$ aggregate cost). In the current paper we show that AOU can also be used to enable in-place update without abandoning nonblocking semantics. In addition, using full-system simulation of a single-chip multiprocessor, we quantify the performance impact of using AOU for conflict detection.

We describe the AOU mechanism in Section 2. In Section 3, we describe how it can be used to implement a new zero-indirection STM. This system requires only that a single line be tagged alert-on-update in the cache. (AOU-based conflict detection requires one line for every object accessed in a transaction, but we can easily fall back to detection based on reader lists or incremental validation in the event of cache overflow.)

Our performance results appear in Section 4. We first evaluate the impact of indirection by comparing our new system to coarse-grain locking, to a similar zero-indirection system based on locks, and to our previous RSTM system, which imposes one level of indirection [22]. We then consider more extensive use of AOU, comparing the new system and RSTM with and without AOU-based conflict detection, and with and without our previously-described *commit counter* heuristic [22], which reduces the asymptotic complexity of validation when transactions rarely overlap. We discuss future work and conclude in Section 5.

On a suite of commonly used microbenchmarks, AOU-based conflict detection dramatically improves the performance of both RSTM and the new zero-indirection system. With any given choice of conflict detection, however, AOU-based zero-indirection STM provides performance very similar to that of both RSTM and the lock-based STM system. This suggests that the overhead of (constant-cost) per-load validation is roughly equal to the cost of extra indirection in our experiments. Clearly, the relative cost of per-load validation will depend in general on the average number of transactional accesses per object; full evaluation of this tradeoff awaits a broader set of benchmarks. Several other factors, however, suggest that our simulated system constitutes something of a "worst case" for systems with per-access validation, and that the balance may tilt in favor of zero-indirection STM in future systems.

2. ALERT-ON-UPDATE

Validation imposes significant overhead on STM implementations. While validation of a single location is simple, the number of distinct locations, and the frequency at which they must be validated, adds up to one of the principal components of overall system work. Consistency violations occur when memory is changed by another thread, but are typically not detected until a validation point, where the modified location is polled. Consequently there is a delay between when a transaction becomes inconsistent and when it becomes aware of its inconsistency. Atomicity and isolation depend on never performing an erroneous, externally visible operation during the delay. The key to AOU-based STM is the observa-

Registers	
`%aou_handlerPC`:	address of handler to be called on a user-space alert
`%aou_oldPC`:	PC immediately prior to call to `%aou_handlerPC`
`%aou_alertType`:	remote_write, lost_alert, or capacity/ conflict eviction
`%alert_enable`:	set if alerts are to be delivered; unset when they are masked
interrupt vector table	one extra entry to hold address of handler for kernel-mode alerts
Instructions	
`set_handler %r`	move `%r` into `%aou_handlerPC`
`clear_handler`	clear `%aou_handlerPC` and flash-clear alert bits for all cache lines
`aload %r`	set alert bit for cache line containing the address in `%r`; set overflow condition code to indicate whether the bit was already set
`arelease %r`	unset alert bit for line containing the address in `%r`
`arelease_all`	flash-clear alert bits on all cache lines
`enable_alerts`	set the alert-enable bit
Cache	
one extra bit per line (the alert bit), orthogonal to the usual state bits	

Table 1: Alert-on-update hardware requirements.

tion that for a transaction whose working set fits in the processor's cache, a consistency violation will always be preceded by a cache eviction, triggered by a remote transaction to ensure coherence.

Since hardware cache coherence protocols already manage evictions and respond to the very same events that indicate consistency violations, AOU has a simple implementation [21, 23]. No changes are required to the coherence protocol or to the processor/memory interface; AOU simply exposes cache eviction events to user code as spontaneous subroutine calls. To use AOU, a thread registers an *alert handler* address with the hardware, and then loads lines of interest using a special `aload` instruction. When the cache controller evicts the line for any reason (typically for coherence, but potentially due to capacity or conflict), it notifies the local processor, which then causes an immediate call to the current thread's alert handler. For the sake of generality, we propose a special register, set by the hardware, that allows the handler to distinguish between coherence, capacity, and conflict evictions, and to tell when events that occur in close temporal proximity may have been rolled into a single alert. Our STM system, however, does not need this information; it suffices simply to know that an alert has occurred.

Full Implementation. Alert-on-update does not require a particular coherence protocol, although it relies on cache coherence events. For our evaluation, we use a modified MESI protocol. Table 1 summarizes hardware requirements: a special register to hold the address of the user-mode alert handler, a register describing the current alert; an interrupt vector table entry (for alerts received while running in kernel mode); instructions to mark and unmark cache lines; instructions to set and unset the user-mode handler; and an instruction to re-enable alerts when the handler has finished execution. The `aload` instruction returns a value indicating whether the line was already marked.

Lightweight Implementation. The full implementation associates a bit with each line in the cache, providing the programmer with the ability to receive notification from many different sources at once (although all notifications are still handled by the same handler). Since using AOU for only a single line at a time can enable interesting algorithms, we consider a simpler hardware implementation in which bits are not associated with individual lines. Instead, we augment the cache controller with a register storing a single location that is marked AOU (one could easily imagine implementations with some small fixed number of such registers; one suffices in our particular STM implementation). The controller signals the processor if it observes (via a coherence event) a remote write to the stored line.

In a snooping coherence protocol, this lightweight implementation obviates the need for the `%aou_alertType` register; on a capacity or conflict eviction, the cache controller can continue to watch for remote writes on the `aload`ed line, even though the line is not cached locally. Additionally, the `arelease` and `clear_handler` instructions could be merged, if desired. The simpler implementation requires only three registers, three instructions, and a single entry in the interrupt vector table.

3. BUILDING STM WITH AOU

In this section we present a three-step evolution of TM systems, enabled by increasing reliance on AOU. The first is a zero-indirection lock-based STM system. The API and some aspects of the code are inherited from our C++ STM library, RSTM [25]. The second system uses one `aload`ed line per thread to guarantee nonblocking progress; the third uses one `aload`ed line per object to avoid incremental validation without requiring visible readers.

3.1 A Zero-Indirection, Locking STM

We have sought in our work to maintain all of the beneficial features of RSTM while eliminating indirection. In particular, our new Lock-and-Redo framework supports both eager and lazy acquisition, provides early release, uses the RSTM API, supports user-provided allocators, and is code-compatible for contention management [19] and validation heuristics [22]. We maintain metadata and check for conflicts at the granularity of language-level objects, rather than individual words, which minimizes the number of words that must be `aload`ed in the AOU-based system of Section 3.2 and increases opportunities for compiler optimizations (not considered here) to elide redundant bookkeeping operations.

Our framework can be viewed as a modification of RSTM employing three transformations:

(1) Lock-and-Redo. As in RSTM, transactions perform all speculative writes on private object clones. Each clone serves, in effect, as a "redo log", to be applied to the master copy in the wake of a successful commit.[1] A thread must perform this copy-back (or verify that some other thread has done so) before it can start a new transaction. In the absence of hardware support, log application is performed under the protection of per-object locks. If a transaction attempts to read a locked object, it must wait. If a transaction attempts to read an unlocked object whose redo log has not yet been applied, the reader locks the object and applies the log on behalf of the writer. Deadlock is not possible in the absence of thread failure, since no transaction is permitted to hold more than one lock at any time.

(2) Per-Access Validation. Because objects are updated in place, a transaction is not guaranteed that the (copy of an) object it reads is immutable. To ensure that all reads come from the same consistent version of the object, we attach a version number to each object. Applying a redo log increments the version number, and reading transactions ensure the consistency of the version number

[1] Other implementations of the log are of course possible. Cloning will be expensive if only a small part of an object is modified, but it has the advantage of simplicity, particularly for transactions that read their own writes.

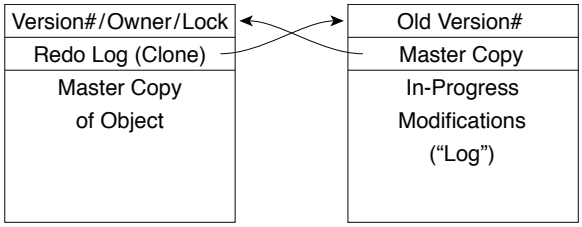

Figure 2: Metadata for a locking, zero-indirection, object-based STM. Object pointers all refer to the box on the left.

before using any field read from an object. Ideally, this mechanism would be implemented by the compiler; we currently embed it in accessor methods.

(3) Indirection Reduction. Given the above modifications, the RSTM indirection header is unnecessary. Pointers to shared objects need not pass through an extra object to mediate conflicts and indicate acquisition. Instead we pack lock status, version number, and acquisition status into a single word in the object itself.

Metadata Manipulations. Figure 2 depicts the per-object metadata for our system. Every object contains two header words, which we modify only using atomic two-word `compare-and-swap` (`CAS`) instructions.[2] In the common case, an object is not owned: its redo log pointer is null and its version number is odd. In this case, the object contents can be read directly, so long as the version number has not changed since the first time the object was accessed by the current transaction.

As in RSTM, writers must *acquire* every written object. They can do so eagerly (at the time of the first access) or lazily (just before commit). An object in the acquired state contains a pointer to the owner in the first header field, and a pointer to the redo log in the second. The first word of the redo log contains the old version number of the object. The second word contains a back-pointer to the public version of the object. If the first word of an object holds the special value 2, then the object is locked and can only be accessed by the lock holder, who is presumed to be actively copying its log back to the master copy.

When a transaction aborts, any object it has acquired is rolled back by zeroing the redo log pointer and resetting the version number. The lock is not required. Aborts are thus nonblocking: any transaction can clean up an object owned by an aborted transaction in a single instruction.

Any reader who encounters an unlocked object with a non-null redo log pointer and a committed owner may attempt to lock the object and apply the redo log; the committed transaction attempts to do the same for each of its updated objects. After completing a log application, a thread zeros the redo log pointer and sets the version number word to the old version number + 2, effectively releasing the lock. Though we do not explore the possibility in our experiments, it may be desirable to use scheduler hooks to discourage the preemption of threads performing copy-back.

Validation. Since acquisition and locking both modify the same header field, it is straightforward for a transaction to ensure that an object has not changed; it need only compare the first word of the header to a private copy of that field that was read the first time the object was accessed. In the common case, this field has not changed, and for small transactions all necessary information is stored in the cache. If the object is locked, then a change has been successfully committed, and the reader must abort. Similarly, if the object is owned by an active or aborted transaction, the reader must assume that it lost a contention management decision and must abort. Thus upon any field access a single comparison is sufficient for validation. When a new object is first read, all previously read objects are validated using this same simple comparison. As in RSTM with invisible readers, the aggregate cost of validation for an N-object transaction is $O(n^2)$.

Costs. The lock-and-redo framework has two drawbacks relative to RSTM. First, without hardware support the framework sacrifices obstruction freedom. Second, it sacrifices the guarantee that committed objects are immutable. Without this guarantee, transactions must perform lightweight validation on every access to a field of an object (in addition to the all-previous-objects validation described in the preceding paragraph). Though the cost of each intra-object validation is low, the total cost may be high if fields of the same object are read many times. Static analysis may eliminate some redundant per-access validations; the naive implementation used in our experiments induces an overhead of 10% in single-threaded code when compared to RSTM.

Comparison to Other STMs. Our TM design most closely resembles TL2 [2]; both perform all speculative writes out of place, perform incremental validation, and use per-access validation to ensure that user code never observes inconsistent values. The chief differences are that our system does not rely on a global clock mechanism (instead treating a global clock as an orthogonal validation heuristic [22]), and that our system permits both eager and lazy acquire. Other similar systems include Microsoft's Bartok TM [9] and Intel's McRT [18]. These systems perform updates in-place, and do not permit lazy acquire. McRT additionally omits incremental validation, relying on the programmer or compiler to invoke validation explicitly before any potentially dangerous operation.

3.2 Restoring Nonblocking Guarantees

The sole purpose of the per-object lock in our framework is to ensure that as soon as one thread has completed the application of a redo log, no other thread continues attempting to apply that log. Since application of a redo log is idempotent, and since the TM ensures that the lock is only used to apply redo logs, we can restore nonblocking guarantees by making the lock revocable. Using `AOU` we can construct a special-purpose revocable lock much more simply than is possible in the general case on conventional hardware [8]. We do not expect the use of a single `alert-on-update` to significantly increase performance; the code is equivalent in complexity to lock-based code, and with OS support the lock-based code should not demonstrate pathological preemption effects. The contribution of this section is rather to demonstrate that AOU makes it easy to build nonblocking STM without introducing indirection.

If all threads agree on the set of locations to be written (a condition guaranteed by the redo log), then the lock can be stolen as long as the previous lock holder is certain not to continue writing once it loses the lock. Figure 3 demonstrates how the redo lock can be stolen using `AOU`. The `AcquireRevocableLock()` operation can either lock an unlocked object or overwrite the lock of a locked object. In the latter case, the current lock holder will receive an immediate alert, ensuring that if the new lock holder completes, no other threads are writing the object. To avoid pathological behavior, `AcquireRevocableLock()` waits for a bounded period of

[2]The 2-word `CAS` is helpful but not necessary; the redo log pointer could take the place of the owner variant of the overloaded first word, in which case the owner field could move into the redo log. This would allow use of a single-word `CAS`, at the expense of an additional level of indirection to determine the object's owner.

```
bool success = true
try
    set_handler({throw Alert()})
    if (log = o->HasRedoLog)
        aload(o->lock);
        if (o->AcquireRevocableLock(log))
            o->ApplyRedoLog(log)
            o->ReleaseLockAndClearLog()
        arelease(o->lock)
    clear_handler
catch (Alert)
    success = false

AcquireRevocableLock(log):
    do
        v = versionNumber
    while (v == 2 && backoff() < THRESHOLD)
    return CASX(this, <v, log>, <2, log>)
```

Figure 3: A single `aload`ed line suffices to steal responsibility for applying a redo log to object *o*.

time before attempting to steal the lock. As an optimization, a committed transaction that finds one of its objects "locked" can simply skip that object, as if the log had been successfully applied. Since threads never hold more than one lock, and since that lock is used only to protect log application, a single `aload`ed line suffices to restore obstruction freedom.

3.3 Reducing Validation Costs

We now focus on a more pervasive use of `AOU` to dramatically improve performance. As in the previous subsection, we use `AOU` to implement revocable locks. However, we also use `AOU` to eliminate both quadratic-time inter-object validation and per-access intra-object validation in the common case.

In the common case, a transaction reads and writes only a small number of objects. If all transaction headers fit in cache, then once an object O is read, its header ought to remain in the cache until the transaction completes. Barring pathological cache overflows, invalidation of O's header implies that O has been acquired by another transaction and the current transaction should abort.

If the transaction registers an alert handler that immediately aborts, and then each object header is initially loaded using `alert-on-update`, all validation can be elided: An alert is certain to precede any modification to relevant metadata, and all forms of validation fail only if some relevant metadata is modified. In effect, `AOU` transforms the cache into a self-validating read set.

Overflow. Our proposal is somewhat idealized, since the capacity of a cache is limited. If we have abundant but finite lines that can be tagged `alert-on-update`, then we may tag up to K objects (where K is based on the cache size) and then fall back to explicit incremental and per-access validation for the remaining $R - K$ objects in the read set. In our implementation, transactions estimate K and decrease it when they are alerted due to overflow (detected through the `%aou_alertType` register). The code path for the first K objects still involves a conditional check to ensure that AOU is in use, but this is a simple in-cache comparison. The branch that follows the comparison is easily predicted, and the test itself should execute in parallel with "real work" on an out-of-order processor.

Immediate Aborts. With abundant lines tagged for `alert-on-update`, there are no explicit validation points at which a transaction may abort. Instead, as soon as an alert is issued the transaction immediately jumps to its abort handler. Certain operations (such as memory management) must be guarded so that they are not interrupted by an alert; since these operations do not access transactional data, there is no risk of consistency violations if alerts are deferred during these blocks. We currently implement deferred aborts through a hardware instruction, although a software-based implementation is straightforward.

Comparison to HASTM. RTM was originally introduced (without performance results) in a 2005 technical report and a paper at TRANSACT '06 [21]. Researchers in the McRT group at Intel subsequently published a variant of AOU that uses synchronous polling instead of asynchronous events to detect cache line evictions. Their HASTM system [17] uses eager acquire and in-place updates, and again relies on the compiler [24] or programmer to invoke validation before any potentially dangerous operation. Because it polls for evictions, it cannot guarantee immediate aborts, and relies on blocking semantics.

4. EVALUATION

In this section we present experimental results that measure the impact of `alert-on-update` on TM performance. We consider both throughput and the cache miss rate, which we use as a measure of the benefit of removing indirection. All results were obtained through full-system simulation.

4.1 Simulator Framework

We simulate a 16-way chip multiprocessor (CMP) using the GEMS/Simics infrastructure [15], a full system functional simulator that faithfully models the SPARC architecture. The `alert-on-update` hardware specified in Section 2 is accessed through the Simics "magic instruction" interface; the AOU bit is implemented using the SLICC [15] framework. Simulation parameters are listed in Table 2.

16-way CMP, Private L1, Shared L2	
Processor Cores	1.2GHz in-order, single issue, ideal IPC=1
L1 Cache	64KB 4-way split, 64-byte blocks, 1 cycle latency, VB:32 entries
L2 Cache	8MB, 8-way unified, 64-byte blocks, 4 banks, 20 cycle latency
Memory	2GB, 100 cycle latency
Interconnect	4-ary totally ordered hierarchical tree, 1 cycle link latency, 64-byte links

Table 2: Simulation Parameters

We use the GEMS network model for bus and switch contention. Simics allows us to run an unmodified Solaris 9 kernel on our target system; its "user-mode-change" and "exception-handler" interface provides a mechanism to detect user-kernel mode crossings. For TLB misses, register-window overflow, and other kernel activities required by an active user context, we defer alerts until control transfers back from the kernel.

4.2 Benchmarks

We consider the following five benchmarks, designed to stress different aspects of software TM. In all benchmarks, we execute a fixed number of transactions in single-thread mode to advance the data structure to a steady state. We then execute a fixed number of transactions concurrently in multiple threads to evaluate scalability and throughput. During the timed trial, we also monitor L1 cache misses (read and write), as we expect them to decrease in systems without indirection.

HashTable. Transactions use a hash table with 256 buckets and overflow chains to lookup, insert, or delete a value in the range $0 \ldots 255$ with equal probability. At steady state, the table is 50% full. HashTable transactions are very small and rarely conflict.

RBTree. Transactions attempt to insert, remove, or delete values in the range $0 \ldots 4095$ with equal probability. At steady state there are about 2048 entries, with about half of the values stored in leaves. RBTree transactions have tens of objects in their read sets, but conflict infrequently.

LFUCache. LFUCache uses a large (2048) array-based index and a smaller (255 entry) priority queue to track the most frequently accessed pages in a simulated web cache. When re-heapifying the queue, transactions always swap a value-one node with a value-one child; this induces hysteresis and gives each page a chance to accumulate cache hits. Pages to be accessed are randomly chosen using a Zipf distribution: $p(i) \propto \Sigma_{0<j\leq i} j^{-2}$. Conflicts are highly likely, but transactions are very small.

LinkedList-Release. Transactions use *early release* to minimize read-set size while performing inserts, lookups, and deletes with equal probability in a sorted, singly-linked list holding values in the range $0 \ldots 255$. Early release is an application-specific optimization that allows a transaction to indicate that writes to an object no longer jeopardize its own ability to commit. Transactions are long-running, but early release keeps the read set small and prevents conflicts.

RandomGraph. Transactions insert or delete vertices from an undirected graph represented with adjacency lists. Edges in the graph are chosen at random, with each new vertex initially having up to 4 randomly selected neighbors. RandomGraph transactions are large (hundreds of objects) and conflict with high likelihood.

4.3 Runtime Systems Evaluated

We compare a total of 8 systems. As baselines, we consider RSTM (`RSTM`), RSTM with our global commit counter heuristic (`RSTM+C`) [22], and a coarse-grained lock library (`CGL`) that enforces mutual exclusion between all transactions through a single test-and-test-and-set lock. `CGL` affords no parallelism, but introduces no overheads for single-threaded code. The commit counter is a global count of the number of transactions that have attempted to commit. When a transaction acquires an object, it sets a local flag indicating that it must increment the counter before attempting to commit. Now when opening a new object, a reader can skip incremental validation if the global commit counter has not changed since the last time the reader checked it.

We also evaluate the STM systems of Sections 3.2 and 3.3. The first, `AOU_1`, uses a single AOU line to eliminate indirection; the second, `AOU_N`, uses one AOU line per object to eliminate validation. For `AOU_1`, we also consider a variant that uses the global commit counter (`AOU_1+C`). Lastly, we consider a variant of RSTM (`RTM-Lite`[3]) that uses AOU to avoid validation overheads (like `AOU_N`) but without eliminating indirection. This library does not incur per-access validation, but has increased cache pressure. Finally, we include the lock-based TM of Section 3.1 (`LOCK`). Its performance is within 5% of `AOU_1` on average, which is unsurprising, since `AOU_1` differs from `LOCK` only when locks are stolen; we do not discuss `LOCK` further here.

To ensure a fair comparison, we use the same benchmark code, memory manager, and contention managers in all systems. For contention management we use the Polka manager [19]. All benchmarks were compiled using `gcc4.1.1` using `O3` optimization, and all TMs use eager acquire.

[3]The name RTM-Lite is derived from its relationship to our hardware-software hybrid TM, called RTM [21].

4.4 Indirection and Per-Access Validation

By eliminating indirection, our new TM reduces cache misses by up to 31%, with read misses dropping by up to 55%. Figure 6 depicts this effect. In HashTable, RBTree, and LinkedList-Release, a third of all transactions are read-only, and these benchmarks see the biggest decrease in cache misses. Unfortunately, the impact of copy-back in write-dominated workloads is substantial. In LFUCache, where often only one object is accessed (and it is written), the write misses due to copy-back increase by over 50%, and actually increase the total cache miss rate. The tradeoff in cache misses is about even for RandomGraph.

Furthermore (Figure 4), we find very little improvement in overall throughput for our indirection-free TMs, despite a decrease in overall cache misses. The write-back operation of lock-and-redo introduces overhead (including write misses) on the critical path of every committed writing transaction. For highly contended objects, a reader encountering a locked object must pull the lock out of modified state in the writer's cache in order to spin, and the writer then must re-acquire exclusive access to continue the copy-back, since the lock and data are colocated on the same cache line. This cache thrashing exacerbates copy-back overhead, and increases the time a lock is held in our `LOCK` system.

Lastly, we note that in separate experiments per-access validation introduced a 10% slowdown for single-threaded executions. Absent compiler support to eliminate redundant checks, indirection-free systems require checks of an object's header field on every access. This expense is particularly pronounced on our simulated processors, which execute in-order and cannot perform the checks in parallel with the application's critical path.

In the end, trading indirection for validation and (possibly) more write misses may or may not improve run time. In an attempt to verify our simulator, we tested our all-software systems on a 16-processor SunFire 6800. We observed speedups as high as 35% over `RSTM` for HashTable running under the `LOCK` runtime, and slowdowns as high as 30% for the same runtime and LFUCache.

4.5 Eliminating Read-Set Validation

By leveraging abundant AOU lines, both `RTM-Lite` and `AOU_N` are able to improve TM performance by 1.4–2× in HashTable, RB-Tree, LinkedList-Release, and LFUCache. Single-thread RandomGraph improves by a factor of 5. Not only do these systems outperform their unaccelerated counterparts (`RSTM` and `AOU_1`, respectively) at all thread levels, they also outperform our commit-counter heuristic in almost all cases.

In previous work we showed that the commit counter entails a tradeoff: in return for a constant-time indication of whether any transaction has committed, all transactions must serialize on a single global counter. For HashTable, where transactions tend not to conflict, this forced serialization is a bottleneck that slows performance. However, both `RTM-Lite` and `AOU_N` avoid serializing on a counter while still enabling validation calls to be skipped. As a result, we see HashTable improve by over 20%, whereas the commit counter actually degrades performance with respect to `RSTM`.

In LFUCache, where transactions conflict with high likelihood and consequently do not admit scalability, we still see that removing validation without adding an expensive `fetch-and-increment` enables an improvement of almost 40%. Furthermore, since AOU decreases the time required to com-

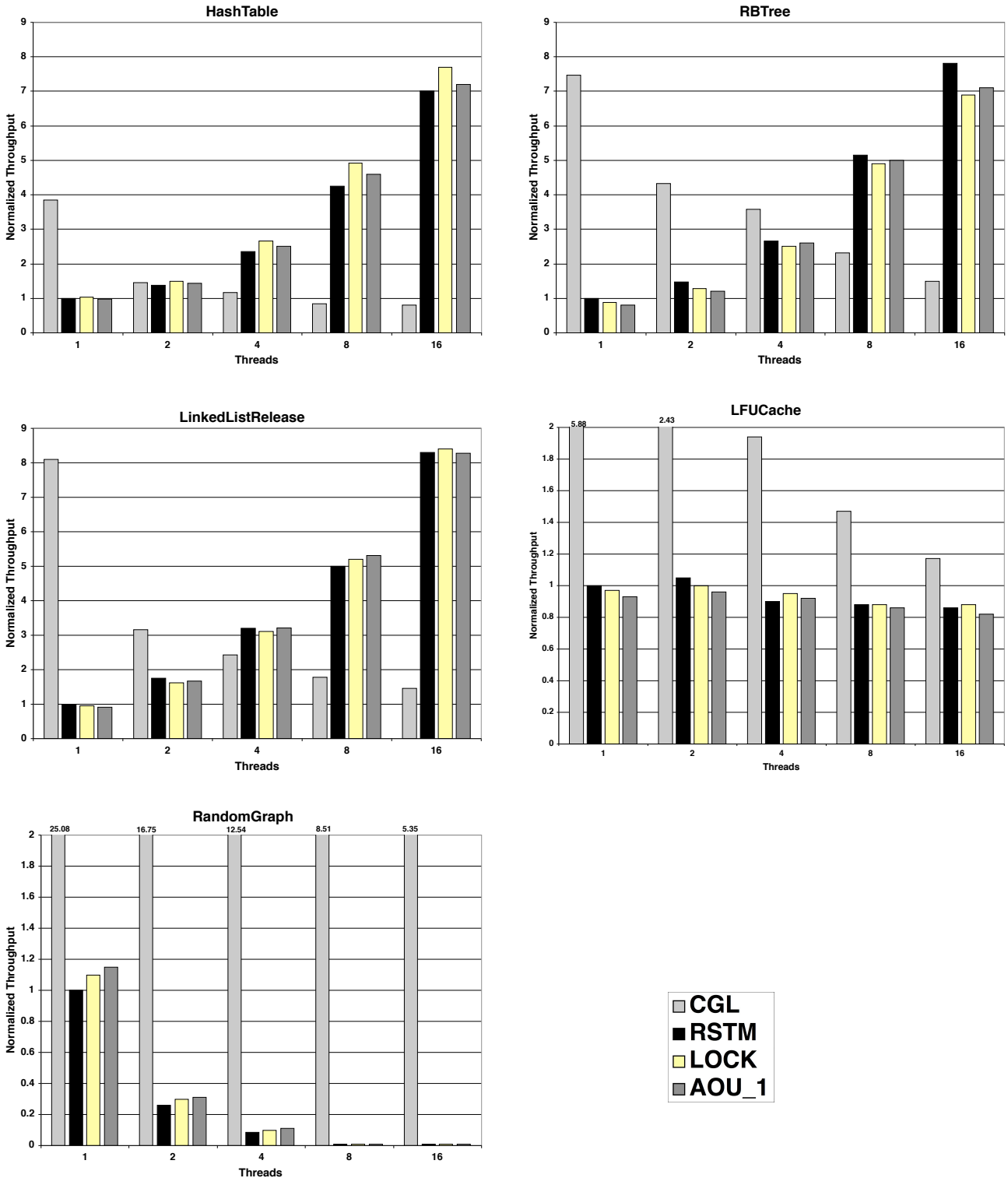

Figure 4: Throughput of lock-based and nonblocking indirection-free TMs. Savings due to reduced indirection are frequently offset by additional validation overheads. Results are normalized to `RSTM`, 1 thread.

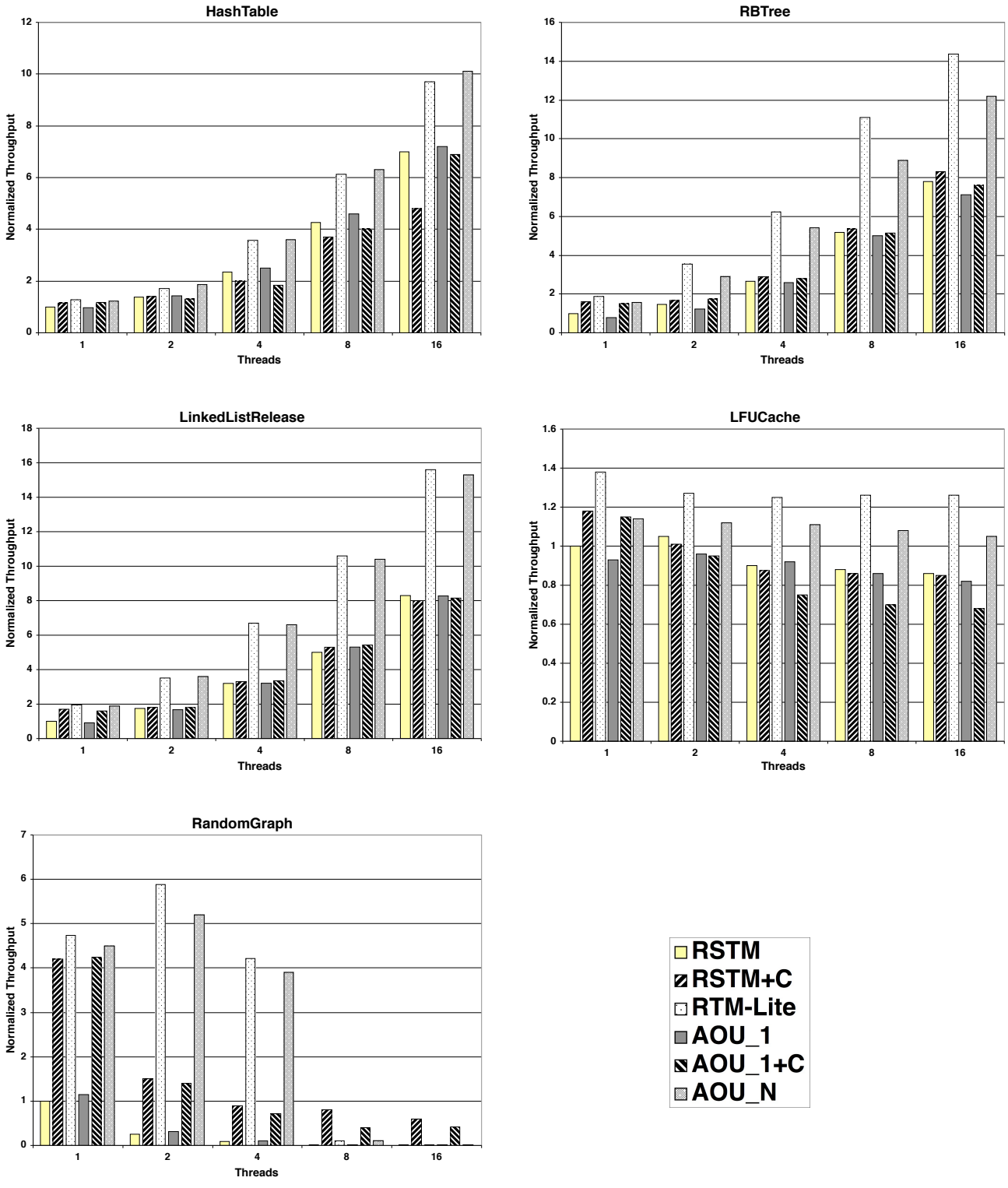

Figure 5: Using `alert-on-update` to eliminate validation improves performance by as much as a factor of 2 (a factor of 5 in RandomGraph), and outperforms the global commit counter heuristic. Results are normalized to `RSTM`, 1 thread.

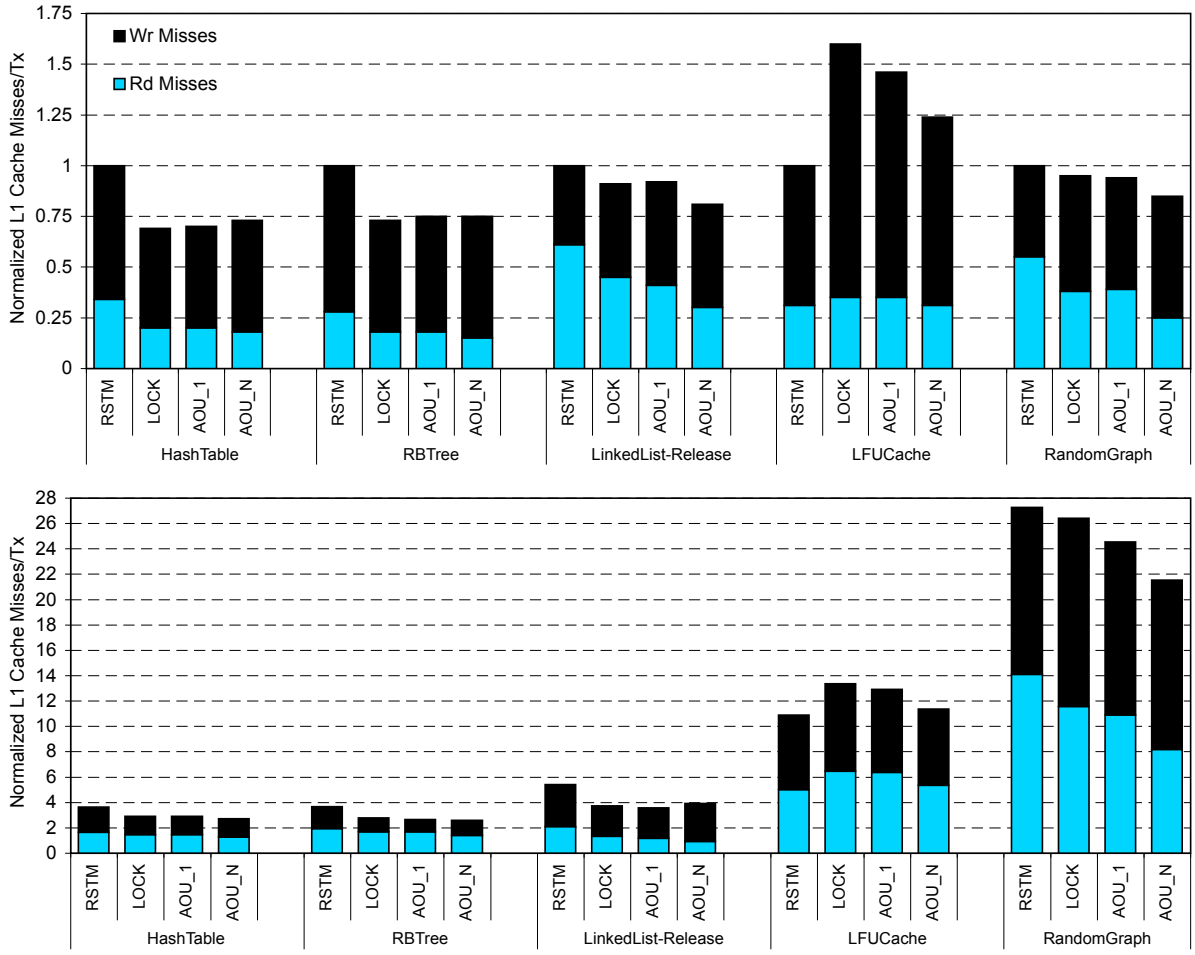

Figure 6: Top:L1 cache misses per transaction at 1 thread. Bottom:L1 cache misses at 16 threads. Results are normalized to `RSTM`, 1 thread.

mit a transaction, performance degrades less at higher thread levels. With faster transactions, the window of conflict is smaller.

In RBTree and LinkedList, the counter is not a bottleneck, but it is imprecise. When a writing transaction increments the counter and commits, all active transactions are forced to validate, even if they do not conflict with the writer. Thus for longer transactions and moderate concurrency (T threads), a transaction is likely to validate $\frac{T-1}{2} = O(T)$ times, even if there are no conflicts. Since AOU precisely tracks conflicts, it is not victim to false-positive events, and thus it improves performance by a much larger amount. For RBTree at 16 threads, AOU increases throughput by 70%, whereas the commit counter improves throughput by less than 10%. LinkedList-Release sees a 2× speedup with AOU, and only a 10% speedup with the counter. The imprecision and false positives induced by the counter mask the concurrency of these benchmarks.

As in other benchmarks, RandomGraph single-thread performance is slightly higher with AOU than with the counter, since AOU does not require an expensive `CAS` operation. However, the counter enables reorderings that approximate *mixed invalidation* [20, 22], which dramatically improves throughput in RandomGraph. Briefly, the counter defers detection of conflicts between a reader and a subsequent writer of an object until the writer commits. If the reader commits first, the conflict is ignored. This behavior is not present when using `AOU`, since the writer's acquisition will immediately alert the reader. Since the window of contention is long in RandomGraph, and since the counter shrinks this window considerably, the commit counter delivers substantially better throughput than AOU.

Analysis of cache misses identifies an interesting trend: When read set validation is avoided, cache misses decrease. This is a direct result of the reduced bookkeeping afforded by AOU. Since the transaction relies on the cache for notification of conflicts, it is not necessary to maintain a large list of all objects read in order to enable validation. Additionally, there is no costly validation step that pulls metadata into the cache, possibly at the expense of objects read transactionally. By reducing bookkeeping, AOU reduces cache pressure and avoids capacity evictions, decreasing the overall miss rate. Since the commit counter is imprecise, it has a similar, but less pronounced effect.

4.6 Latency and Overhead

Figure 7 quantifies the overheads incurred by our various TM systems in single-thread execution. Among the principle overheads, only validation and bookkeeping vary significantly across systems; other overheads are either negligible (due to the lack of conflicts in single-threaded code) or constant.

Our latency measurements reflect some instrumentation artifacts. As in HASTM [24], since object metadata is located within

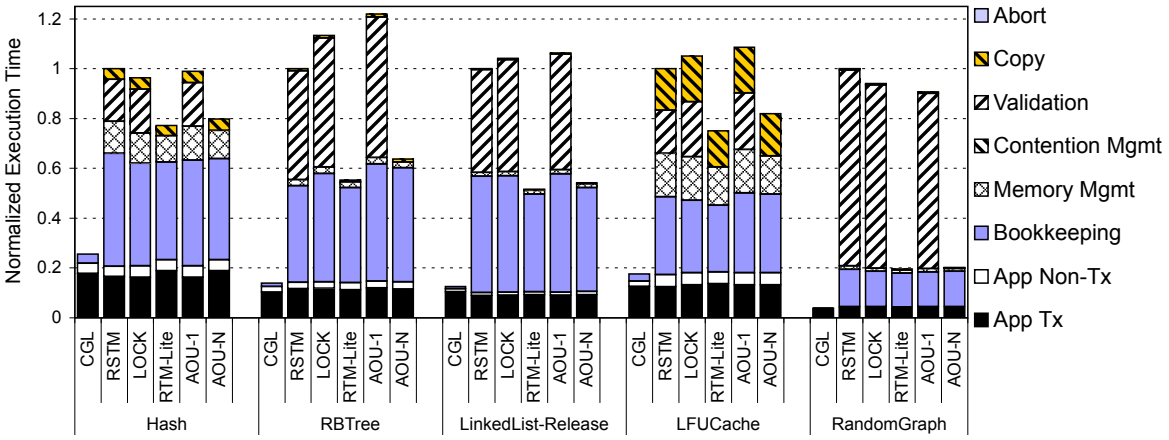

Figure 7: Latency and Overhead for redo_lock at 1 thread.

data objects, the cost of pulling an object into the cache is represented as bookkeeping rather than real work (App Tx). In `RSTM`, this results in one level of indirection being assigned to App Tx and the other to metadata manipulation (Bookkeeping), as desired. However, for redo_lock this artificially inflates bookkeeping. Secondly, since per-access validation is only a three-instruction sequence (cache hit, compare, branch), we treat that overhead as App Tx rather than as validation, in order to limit the instrumentation cost. This incorrectly adds all per-access validation in the redo_lock-based systems to App Tx overhead.

The combination of these effects paints a surprising picture. Indirection and per-access validation overheads are roughly equal, resulting in a slight slowdown in redo_lock for most benchmarks despite the removal of indirection. Furthermore, in the absence of validation we see that metadata bookkeeping is the dominant overhead. In our systems, this overhead is the cost of flexibility and obstruction freedom: we must bookkeep eager and lazy writes separately, resulting in higher constant overhead per transaction, and we must execute multiple branches when reading any object, in order to choose between visible and invisible reads (in `RSTM`) and eager or lazy acquire. We also collect extensive statistics to drive contention management and adaptive policies (not employed here) that choose between eager and lazy acquire, and between visible and invisible readers. To support obstruction freedom and flexible contention management, our systems must obey a protocol for stealing ownership, stealing locks, and aborting competitors that places tens of instructions on the critical path. For large transactions, this per-object cost is an obstacle to good performance.

Sensitivity to Cache Size. Our benchmarks present a best-case scenario for `AOU_N` and `RTM-Lite`. Even RandomGraph fits entirely in the L1 cache, and thus despite hundreds of transactional objects, `AOU` can still be leveraged to avoid all incremental validation overhead. Under more taxing conditions (such as cache associativity constraints or read sets dramatically larger than the number of cache lines and victim-buffer entries), the relative benefit of `AOU` will decrease. Assuming no commit counter, for $R >> C$ objects, where C is the cache size (in lines), the expected validation overhead is $O((R-C)^2)$. Compared to the validation overhead of `RSTM` or redo_lock ($O(R^2)$), the cost will be less in practice, though still quadratic. For such workloads, combining `AOU` and the commit counter would appear to be an attractive option.

5. CONCLUSIONS

We have shown that a simple hardware mechanism, `alert-on-update` (AOU), can be used to (1) eliminate indirection in nonblocking STM, and (2) avoid both intra- and inter-object validation. Using full-system simulation, we have quantified the performance impact of these optimizations.

Somewhat to our surprise, we find that indirection elimination alone (whether via locks or AOU) is largely performance neutral, since it introduces the need to validate the consistency of objects on second and subsequent loads. This may be an overly pessimistic finding: compiler elimination of provably unnecessary validations or out-of-order execution of those that are necessary may tip the balance toward zero-indirection systems. On the other hand, application of redo logs can induce significant numbers of cache misses in the event of high contention (as in our LFUCache benchmark), and the cost of these misses may go up in future systems. Using undo logs instead of redo logs (as suggested by Moore et al. [16]) would avoid the extra miss overhead in successful transactions, but it would also preclude the use of lazy conflict detection, which Dice et al. argue is essential to good performance in lock-based TM [2]. On balance, we find that indirection elimination is a good idea, but that its benefits may be overstated if one does not take into account the need for per-access validation.

In an unequivocally positive result, we find that using AOU to eliminate validation can dramatically improve performance, by factors of 1.4–2 in most of our microbenchmarks, and by as much as 5× in the "torture-test" case of RandomGraph. Our results indicate that AOU is a major win for STM implementation, and suggest the possibility of achieving nonblocking semantics in software-managed TM at no penalty relative to blocking alternatives.

Within the TM runtime, we are exploring alternative implementations of the redo log, to avoid the cost of cloning in transactions that modify small portions of large objects. We are also developing a system that uses AOU to protect in-place object updates, with stealable undo logs to preserve nonblocking semantics. We intend to address the overhead of per-access validation via compiler optimizations that cache the results of validated reads and that eliminate validation operations that are provably redundant, or that have no feasible code path to a potentially dangerous ("un-roll-back-able") operation. Farther afield, we are exploring several nontransactional uses of AOU, together with other hardware assists for STM.

Acknowledgments

Our thanks to Virtutech for providing free academic access to the Simics platform, and to the Multifacet group at the University of Wisconsin–Madison for providing us with source for their GEMS simulation system. Virendra Marathe helped to devise the alert-on-update mechanism; Hemayet Hossain helped to develop our simulation infrastructure.We would also like to thank the anonymous reviewers for many helpful suggestions.

6. REFERENCES

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

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

[3] D. Dice and N. Shavit. What Really Makes Transactions Fast? In *ACM SIGPLAN Workshop on Transactional Computing*, Ottawa, ON, Canada, June 2006.

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

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

[6] 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.

[7] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *OOPSLA 2003 Conf. Proc.*, Anaheim, CA, Oct. 2003.

[8] T. Harris and K. Fraser. Revocable Locks for Non-Blocking Programming. In *Proc. of the 10th ACM Symp. on Principles and Practice of Parallel Programming*, Chicago, IL, June 2005.

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

[10] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *Proc. of the 23rd Intl. Conf. on Distributed Computing Systems*, Providence, RI, May, 2003.

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

[12] 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.

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

[14] V. Marathe and M. Moir. An Efficient Nonblocking Copyback Mechanism in Hybrid Transactional Memory (poster paper). In *Proc. of the 12th ACM Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

[15] 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*, Sept. 2005.

[16] 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.

[17] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson. Architectural Support for Software Transactional Memory. In *Proc. of the 39th Intl. Symp. on Microarchitecture*, Dec. 2006. Orlando, FL.

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

[19] 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.

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

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

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

[23] M. F. Spear, A. Shriraman, H. Hossain, S. Dwarkadas, and M. L. Scott. Alert-on-Update: A Communication Aid for Shared Memory Multiprocessors (poster paper). In *Proc. of the 12th ACM Symp. on Principles and Practice of Parallel Programming*, San Jose, CA, Mar. 2007.

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

[25] The Rochester Software Transactional Memory Runtime. 2006. www.cs.rochester.edu/research/synchronization/rstm/.