Adaptive Software Transactional Memory*

Virendra J. Marathe, William N. Scherer III, and Michael L. Scott

Technical Report #868
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226

{*vmarathe, scherer, scott*}@*cs.rochester.edu*

May, 2005

Abstract

Software Transactional Memory (STM) is a generic synchronization construct that enables automatic conversion of correct sequential objects into correct *nonblocking* concurrent objects. Recent STM systems, though significantly more practical than their predecessors, display inconsistent performance: differing design decisions cause different systems to perform best in different circumstances, often by dramatic margins. In this paper we consider four dimensions of the STM design space: (i) *when* concurrent objects are acquired by transactions for modification; (ii) *how* they are acquired; (iii) what they look like when not acquired; and (iv) the non-blocking semantics for transactions (lock-freedom vs. obstruction-freedom). In this 4-dimensional space we highlight the locations of two leading STM systems: the DSTM of Herlihy et al. and the OSTM of Fraser and Harris. Drawing motivation from the performance of a series of application benchmarks, we then present a new *Adaptive* STM (ASTM) system that adjusts to the offered workload, allowing it to match the performance of the best known existing system on every tested workload.

1 Introduction

Traditional lock-based concurrent systems are prone to several important problems including deadlock, priority inversion, convoying, and lack of fault tolerance. Coarse grain locks are not scalable, and algorithms based on fine grain locks are notoriously difficult to write. Ad-hoc nonblocking implementations of concurrent objects avoid the semantic problems of locks and can match or exceed the performance of fine grain locking, but are at least as difficult to write.

Ideally, one would like a mechanism that provides the convenience of coarse grain locks without their semantic or performance problems. Toward this end, Herlihy [6] was the first to suggest automatically converting sequential code to correct nonblocking concurrent code. Inspired in part by hardware proposals for multiword atomic primitives [9, 20], subsequent researchers developed more sophisticated constructions [1, 11, 15, 19, 21], but these suffer from various practical problems: Most impose time or space overheads too high for real-world systems. Some require unrealistic atomic primitives. Many support only static collections of objects and/or static transactions (in which the set of objects to be accessed is known in advance).

*This work was supported in part by NSF grant numbers EIA-0080124, CCR-9988361, and CCR-0204344, by DARPA/AFRL contract number F29601-00-K-0182, and by financial and equipment grants from Sun Microsystems Laboratories.

Recent software transactional memory (STM) systems have overcome most of these problems [2, 3, 5, 8]: They employ atomic primitives like compare-and-swap (CAS) and load-linked/store-conditional (LL/SC), which are widely supported by current multiprocessors. They support dynamic collections of objects and dynamic transactions. Their space overheads are modest. Finally, their performance overheads are low enough to outperform coarse-grain locks in important cases, and to provide an attractive alternative to even well-tuned fine-grain locks when convenience, fault tolerance, and clean semantics are considered. Those of us working in the field envision a day when STM mechanisms are embedded in compilers for languages like Java and C#, providing nonblocking implementations of `synchronized` methods "for free".

STM systems may be either word- or object-based, depending on the granularity at which data is modified. We focus on the latter, which appear particularly well suited to object-oriented languages such as Java and C#. The two leading object-based STMs are currently the DSTM of Herlihy et al. [8] and the OSTM of Fraser and Harris [2, 3]. In prior work [13], we undertook a preliminary comparison of these systems. Among other things, we found that DSTM outperforms OSTM by as much as an order of magnitude on write-dominated workloads, while OSTM outperforms DSTM by as much as a factor of two on read-dominated workloads. We attribute much of the difference to the choice of acquire semantics (discussed in Sections 2 and 3.1) and to indirect object access in DSTM, which imposes an extra cache miss on the typical read access.

In the current paper we discuss four principal design decisions for object-based STM, encompassing (i) *when* concurrent objects are acquired by transactions for modification (*eager* vs. *lazy* acquire); (ii) *how* they are acquired (per-object or per-transaction metadata); (iii) what they look like to readers when not currently acquired (level of indirection); and (iv) the type of nonblocking semantics (lock-free vs. obstruction-free). Our taxonomy enhances previous [3, 13] understandings of DSTM and OSTM performance by locating these systems within the four-dimensional design space. By clarifying the performance implications of design dimensions, the taxonomy has also allowed us to develop an *Adaptive* STM system (ASTM) that automatically adapts its behavior to the offered workload, allowing it to closely approximate the performance of the best existing system in every scenario we have tested.

As background, Section 2 outlines a common usage pattern and API for object-based STM. Section 3 presents our characterization of the STM design space and locates DSTM and OSTM within it. Section 4 describes three variants of ASTM, all of which revisit design decisions at run time based on the offered workload. The first two variants adapt along dimensions (ii) and (iii) above; the third adds simultaneous adaptation along dimension (i). Experimental results, presented in Section 5, show ASTM to be competitive with the better of OSTM and DSTM across all execution scenarios, making it highly attractive for general-purpose use.

2 STM Usage

STM transactions provide nonblocking concurrent access to one or more shared *transactional objects*. Transactions typically proceed through the following four phases with respect to any given object:

Open: makes the object available to the transaction. A transaction cannot access an unopened object, but objects may be concurrently opened by multiple transactions.

Acquire: asserts the transaction's *ownership* of the object. An object can be acquired by only one transaction at a time. Any other transaction that subsequently attempts to acquire the object must detect and resolve the conflict. The conflict resolution method determines whether the STM is lock-free [2, 3] or obstruction-free [5, 8]. Acquire is not typically a separate operation. DSTM [8] implements it as part of the open operation (*eager acquire*). OSTM [2, 3] implements it as part of the commit operation (*lazy acquire*).

Commit: attempts to atomically effect all changes to acquired objects made by the current transaction. Typically, this is done by using a compare-and-swap (CAS) or store-conditional instruction to update the status field in a *transaction descriptor* accessible to all competing transactions. (Without loss of generality we assume the use of CAS in the remainder of this paper.) This CAS serves as the linearization point [10] of the transaction.

Release: cleans up metadata associated with a transaction after it commits or aborts. DSTM introduced the concept of *early release* [8], which allows a transaction to relinquish an accessed object prior to committing. Early release reduces the window of contention in which multiple transactions are transiently dependent on a particular object, but it requires the programmer to exploit application-specific knowledge to ensure that transactions are consistent and linearizable.

In our STM experiments (Section 5) we employ a uniform API for beginning, validating, committing, and aborting transactions; and for accessing objects in read-only or read-write modes:

`ref = STM_Open(obj, mode):` Make `obj` accessible for reading or writing (`mode` = *read* or *write*). To ensure invisibility of updates until the transaction commits, implementations typically copy any object opened in write mode, and return a reference to the copy.

`status = STM_Commit():` Attempt to linearize and commit. Return an indication of success or failure.

`STM_Abort(trans):` Abort transaction `trans`. A transaction may abort itself to preserve object semantics. It may abort itself or a competitor to resolve a conflict.

`STM_Release(obj):` Voluntarily release an object prior to committing. This is an unsafe optimization. Normal, on-time release is hidden in `STM_Commit` and `STM_Abort`.

We assume that acquisition of objects open for writing and validation of objects open for reading are hidden in either STM_Open or STM_Commit. Both acquisition and validation may, internally, abort the current transaction or one with which a conflict has occurred.

3 STM Design Space

In this section we discuss four key dimensions in the STM design space. We also present a brief comparison of DSTM and OSTM in light of them.

3.1 Eager vs. Lazy Acquire

Transactions acquire objects when first accessed under eager acquire semantics, but at commit time under lazy acquire semantics. Eager acquire allows a transaction to detect contention earlier; this enables a doomed transaction to abort instead of performing useless work. Eager acquire also makes it easier to ensure consistency: If every conflict is resolved when first detected, then an unaborted transaction can be sure that its objects are mutually consistent as of the most recent open operation. With lazy acquire, by contrast, the obvious approach requires a transaction to maintain a private list of opened objects, and to re-verify the entire list when opening another. This approach imposes total overhead quadratic in the number of opened objects. An alternative is to "sandbox" transactional code to catch and recover from memory and arithmetic faults, and to impose a limit on execution time as heuristic protection against infinite loops. In some cases it may be safe to execute with inconsistent data, in which case no verification overhead is required, but this requires application-specific knowledge.

On the other side of the tradeoff, lazy acquire can greatly shrink the window wherein transactions see each other as contenders and may be unnecessarily aborted or delayed. In particular, with lazy acquire transaction A will never be aborted by transaction B if B itself aborts before A attempts to commit.

3.2 Metadata Structure

A transactional object typically wraps an underlying data object; i.e. it contains a pointer through which the data object can be referenced. Two different means of acquiring such an object appear in current object-based STM systems: (i) an acquirer makes the transactional object point to the transaction descriptor; and (ii) the acquirer makes the transactional object point to an *indirection object* that contains pointers to the transaction descriptor and to old and new versions of the data object. Option (i) allows conflicting transactions to see all of their competitors' metadata, but makes object release expensive: A released object cannot be left in an acquired state (where it points to a descriptor), because subsequent accesses would have to search the descriptor's write list to find a current version. Additionally, maintaining descriptors for previous committed and aborted transactions would constitute potential space overhead quadratic in the total number of objects. Instead, a *cleanup* operation must update transactional objects to point directly to the current data object, at the cost of an extra `CAS` per object.

Cleanup can be omitted entirely with per-object metadata, at the cost of 2× overhead in space. Alternatively, a lightweight cleanup phase might "zero out" pointers to obsolete data objects, making them available for garbage collection. An attractive intermediate option is to omit the cleanup phase, but allow readers to clean up obsolete data objects incrementally.

3.3 Indirection in Object Referencing

As noted in the previous subsection, cleanup is essential with per-transaction metadata. It is optional with per-object metadata. If performed, it provides readers with direct access to the data objects of currently unacquired transactional objects. If not performed, it will tend to induce an extra cache miss for every object opened in read-only mode. The consequent slower reads have a significant impact on overall throughput for read-dominated workloads [13].

3.4 Lock-Freedom vs. Obstruction-Freedom

Lock-free concurrent objects guarantee that at least one thread makes progress in a bounded number of time steps. Obstruction-freedom [7] admits the possibility of livelock, but tends to lead to substantially simpler code: Lock-free STM systems typically arrange to acquire objects in some global total order. They also perform *recursive helping*, in which transaction A performs transaction B's operations on its behalf when A detects a conflict with B. Lock-freedom is most naturally implemented with lazy acquire and sorting; without them a transaction might discover the need to open an object that precedes some other already-opened object in the global total order.

Arbitration among competing transactions in an obstruction-free system is handled "out of band" by a separate *contention manager* [8]. With the right manager, one can obtain high probabilistic—or even strict—guarantees of progress [4, 17, 18].

3.5 Placing DSTM and OSTM

DSTM is obstruction-free, whereas OSTM is lock-free. Transactional objects in DSTM point to an indirection *locator* object, which in turn contains pointers to the most recent acquirer's transaction descriptor and to old and new versions of the data. The acquirer's status determines which version of an object is valid: the

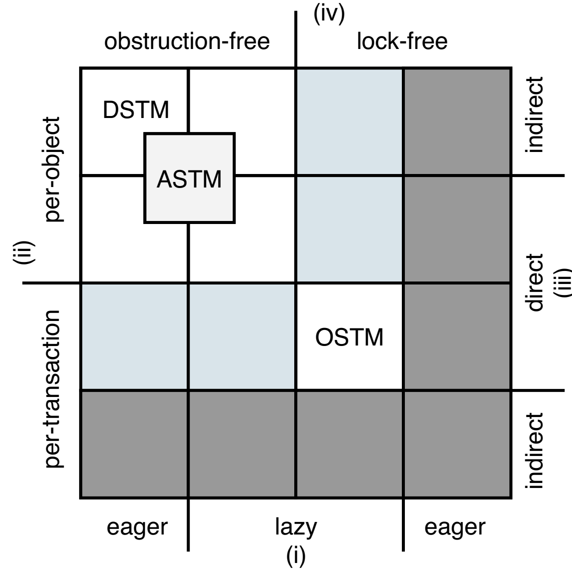

Figure 1: STM design space. The dark grey areas seem ill-advised: eager acquire makes it difficult to implement lock freedom; per-transaction metadata increases the cost of indirect access by readers. The light grey areas seem to have little to recommend them: sorting for lock freedom is naturally performed on the write list of per-transaction metadata, but per-object metadata makes it easier to find the current version of a recently written object for which direct access has yet to be restored.

new version is valid only when the acquirer has `COMMITTED`. An unacquired transactional object in OSTM points directly to the appropriate version of the data. As we shall see, this difference in object referencing results in poor performance for DSTM in read-dominated workloads.

OSTM is lock-free; hence, it uses lazy acquire. DSTM uses eager acquire. For write access, DSTM acquires objects by swapping in a new locator; OSTM acquires objects by making them point directly to the transaction descriptor. Neither DSTM nor OSTM acquires objects accessed in read-only mode. Rather, both systems maintain a private *read list*, which they revalidate on every `open` operation. OSTM must also validate its write list on every `open`. These bookkeeping, sorting, validation, and cleanup overheads cause OSTM to perform poorly in write-dominated workloads. DSTM has no cleanup phase, but it arranges for readers, at `open` time, to “zero out” fields in locators that point to obsolete data objects, thus making them available to the garbage collector.

The four-dimensional STM design space is depicted graphically in Figure 1. Within this space, DSTM can be summarized as the point ⟨eager acquire, per-object metadata, indirect object referencing, obstruction-free⟩; OSTM maps to ⟨lazy acquire, per-transaction metadata, direct object referencing, lock-free⟩. In the following section we describe our ASTM system. It adapts across dimensions (i) and (iii) of the design space: ⟨eager OR lazy acquire, per-object metadata, direct OR indirect object referencing, obstruction-free⟩.

4 Adaptive Software Transactional Memory

As noted in Section 3, and as demonstrated in preliminary experiments [13], the location of an STM system in the design space plays a key role in its algorithmic complexity and performance. The fact that different systems perform best in different circumstances raises the question: Is it possible to adapt among multiple

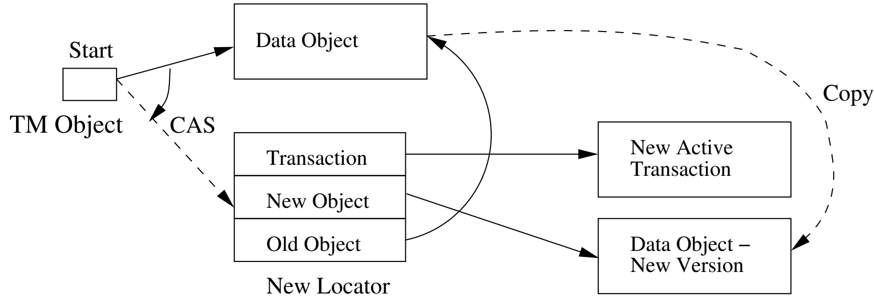

Figure 2: Acquiring a previously unacquired object.

design points to obtain a "best-of-both-worlds" STM system that performs well in all scenarios? We address this question with our ASTM design.

4.1 Basic ASTM Design

In our experiments to date we have yet to identify a workload in which lock freedom provides a performance advantage over obstruction freedom. Given the design flexibility and algorithmic simplifications enabled by obstruction freedom, together with the success of practical contention managers in avoiding livelock problems [18], we have adopted obstruction freedom for ASTM.

Figure 2 depicts a *Transactional Memory Object (TMObject*, borrowed from DSTM terminology*)* in ASTM. By default, TMObjects point directly to data objects. A reader transaction does not acquire a TMObject. Instead, it maintains a private read list and guarantees transaction consistency by re-validating the objects on that list at each new open operation. Reads by a transaction are thus *invisible* to other transactions in the system. A writer transaction, on the other hand, acquires a TMObject by installing an indirection object. Borrowing again from DSTM, we refer to this indirection object as a *locator*. Figure 2 illustrates the object acquisition process. ASTM locators, like those of DSTM, contain three pointers: one for the acquirer's transaction descriptor, and one each for old and new versions of the data object. Before acquiring an object a writer instantiates a new locator and a new version of the data object, which it copies from the most recently committed version, found in the current TMObject. The writer then acquires the target TMObject by using an atomic CAS to swing it to the new locator.

A transaction may be in any of three states: ACTIVE, ABORTED, or COMMITTED. With contention, a writer's acquire attempt may fail, or it may find an ACTIVE transaction's locator in the TMObject. The contention manager is invoked to determine which transaction should proceed. If the contention manager indicates that transaction A should abort its competitor B, A attempts to do so by CASing B's descriptor's state from ACTIVE to ABORTED. Otherwise the current transaction retries the acquire (possibly after waiting for some time to let its competitor complete execution). Once all operations for a transaction are complete, it is committed upon successfully changing the descriptor state from ACTIVE to COMMITTED. Immediately prior to committing, a transaction must perform one final validation of objects on its private read list.

A TMObject that points directly to a data object is said to be in the *unacquired* state; a TMObject that points to a locator is in the *acquired* state. If TMObjects were to remain in the acquired state after a writer completes, then subsequent reads would suffer indirection overhead, as described in Section 3.3. To spread the overhead of cleanup over time, and to avoid it entirely when the subsequent access is a write, we leave the work to readers. A reader that finds an object in the acquired state with a transaction that is either COMMITTED or ABORTED first converts the object to the unacquired state. Figure 3 illustrates this conversion. A reader detects contention when an acquired TMObject's locator refers to an ACTIVE transaction.

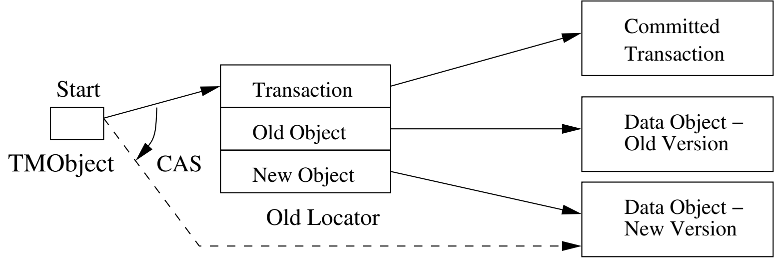

Figure 3: Cleanup while opening an acquired object in read mode.

In workloads dominated by reads, ASTM objects will tend to stay in the unacquired state, improving performance through lack of indirection. In workloads dominated by writes, objects will tend to stay in the acquired state, avoiding the overhead of cleanup. Additionally, ASTM uses eager acquire by default for objects opened in writable mode. ASTM thus seems positioned to perform well for a variety of workloads. Our experimental results (Section 5) confirm this expectation.

4.2 Workload Effect: Readers vs. Writers

As in DSTM, an ASTM transaction that opens N objects in write mode requires $N+1$ `CAS`es: N to acquire the objects, and one to commit. However, subsequent readers may perform up to N `CAS`es to return the objects to an unacquired state. For write-dominated workloads, these cleanup `CAS`es are rare.

With a roughly uniform mix of reads and writes, transactions that include reads are likely to perform several cleanup `CAS`es. In this case, TMObjects could repeatedly flip between acquired and unacquired states. Readers would become slower, due both to the extra level of indirection in acquired TMObjects and to the extra `CAS`es needed to revert objects from the acquired to unacquired states. These `CAS`es could also interfere with writers, slowing them down as well. As the results in Section 5 will show, however, overall system throughput for ASTM is always competitive with the better existing STM. We attribute this primarily to the division of labor between reads and writes, and to the use of eager acquire: Compared to DSTM, we avoid an extra level of indirection for unacquired objects; compared to OSTM, the benefits of eager acquire significantly outweigh the performance lost from flipping transactional object states.

In the best case, an individual transaction may find all its objects unacquired and incur no indirection overhead. In the worst case, it may find all its objects acquired. In addition to indirection overhead, it would then incur the costs of cleanup for objects opened in read mode plus contention-induced slowdown for writes. In Section 5, we assess benchmarks with transactions that fall in several points on the read-write spectrum. We intend to continue exploring reader-writer tradeoffs as future research. In particular, it seems worth considering implementations in which readers do not always perform cleanup operations, but use runtime profiling or randomization to move objects to unacquired state only when the percentage of reads is sufficiently high. For cases where read throughput is more important than write throughput, it may be worth considering implementations in which writers sometimes proactively restore objects to the unacquired state as a part of their release code.

4.3 Lazy ASTM

As described so far, ASTM uses eager acquire for objects opened in write mode. Suppose that transaction A detects a conflict with B when attempting to open some object of mutual interest. If A is doomed to fail, detecting the conflict early may allow it to abort, and avoid unnecessary work. Similarly, if B is doomed to fail, A has the opportunity to kill it right away. As noted in Section 3, however, the tradeoff can go the other

way. We cannot in general be sure which transaction(s) might eventually succeed. In an obstruction-free system like ASTM, the contention manager simply makes an informed guess. If A waits for—or aborts itself in favor of—B, and B fails to commit, we have incurred a significant cost. Similarly, if A aborts B but then itself fails to commit, the work of B was wasted. Either way, this cost could have been avoided if A had delayed its acquire. Even if A rightly kills B, it is possible that B would have detected an inconsistency and aborted itself soon after, suggesting that we may have wasted the overhead of contention management.

To evaluate these tradeoffs, we have developed a lazy variant of ASTM that acquires no objects until commit time. Like the default Eager ASTM, Lazy ASTM uses contention management; both are obstruction-free. Lazy acquire reduces the window in which transactions may see each other as competitors.

As in Eager ASTM, each TMObject in Lazy ASTM can be in the acquired or unacquired state. Readers perform cleanup as necessary. Writers, on the other hand, do not eagerly acquire target TMObjects. Instead, a transaction maintains a private *write list* in its descriptor, in addition to the read list. This write list is revalidated (as is the read list) when opening each new transactional object. Essentially, a transaction remains invisible to other transactions until commit time. As a result, several transactions may open the same object in write mode at the same time. At commit time, a transaction attempts to acquire the TMObjects in its write list by swapping in pointers (using `CAS`) to newly created locators, each corresponding to the respective TMObject. The `CAS` will fail if any other transaction has modified the object in-between. But where OSTM sorts objects and uses recursive helping to guarantee lock-freedom, ASTM traverses the write list unsorted and relies on the contention manager to arbitrate conflicts.

As in Eager ASTM, writers perform no cleanup to convert TMObjects from acquired to unacquired state; i.e. a target TMObject's locator is not replaced by the most recently valid data version. Consequently, entries on the write list must associate two different kinds of values with TMObjects: data pointers for objects initially encountered in unacquired state and locator pointers for objects initially encountered in acquired state. (Entries also contain, for each object, a pointer to the newly allocated locator that will be used at acquire time.) At revalidation or acquire time, a writer may find that the locator pointer in a TMObject has been replaced with a data pointer, yet the original version of the data is still valid. Permitting the writer to continue in such cases is critical for good throughput in workloads with a substantial percentage of reads.

4.4 Adapting to Acquire Semantics

Transactions that are doomed to fail tend to detect contention late in their lifetime with lazy acquire, after doing a lot of wasted work. On the other hand, with eager acquire, doomed transactions can interfere with other transactions, delaying or aborting them. Unfortunately there is no way in general to distinguish between these scenarios. In fact, our experiments so far seem to suggest that the tradeoffs largely "cancel out": Though eager acquire is usually a little faster, neither dramatically outperforms the other in most cases. This suggests to us that eager acquire is usually preferable, in order to avoid the bookkeeping complexity of maintaining write lists.

In one specific case, however, our experimental results (Section 5) reveal a significant advantage for lazy acquire: transactions that access a large number of objects in read-only mode, that perform early release on many of these (thereby reducing the window of contention with other potentially conflicting transactions), and that access only a few objects in write mode. A common example is a reader transaction that uses early release incrementally while following a linked path through a data structure. Such a transaction may suffer delays if it encounters objects eagerly acquired by writers. Lazy acquire significantly reduces contention windows for this workload.

The next logical step in our quest for adaptivity in ASTM is to dynamically select an object acquisition strategy. From our observations on the synergy between lazy acquire and early release, we invoke a history-

based heuristic for a transaction to adapt to the better acquire strategy (eager or lazy) based on observations made by past transactions. The final version of our system ("Full ASTM") defaults to eager acquire, since this provides the best performance in most cases (details in Section 5). We maintain rolling averages for percentages of writes and of early releases that follow at least one write across transactions executed by a thread. So long as the percentage of writes is below a threshold w and the percentage of early releases exceeds another threshold r, subsequent transactions use lazy acquire; otherwise they use eager acquire.

Our heuristic assumes that transactions will tend to behave similarly to those recently completed. It also reflects the quadratic cost of incremental validation: Even a modest number of writes will result in overhead significant enough to overcome the benefits from reduced contention windows in lazy acquire / early release. In our experiments, we have set the early release threshold r at 50% and the write access threshold w at 25%. Limited experimentation with other values suggests that results are largely insensitive to the exact percentage used.

As in Eager ASTM, readers clean up transactional objects if necessary, and writers use indirection objects (DSTM-style locators). Full ASTM remains obstruction-free; it requires contention management.

5 Experimental Results

In this section we provide a detailed empirical analysis of the main aspects of the STM system design space addressed in this paper: acquire semantics, acquire methodology, object referencing style, and progress guarantees. We present experimental results for a variety of concurrent data structures to assess the extent to which ASTM successfully adapts to the offered workload. Specifically, we consider six different benchmarks: a hash table, two variants of a list-based sorted set of integers, an adjacency-list based random graph, a red-black tree, and an HTTP web cache simulation using the least-frequently used (LFU) algorithm. We use DSTM and OSTM as a baseline against which to compare our results.

Experiments were conducted on a 16-processor SunFire 6800, a cache-coherent multiprocessor with 1.2GHz UltraSPARC III processors. The testing environment was Sun's Java 1.5 beta 1 HotSpot JVM, augmented with a JSR166 update from Doug Lea [12]. We measured throughput over a period of 10 seconds for each benchmark, varying the number of worker threads from 1 to 48. Results were averaged over a set of six test runs. In all experiments, we use the Polka contention manager [18] for DSTM and for all three variants of ASTM (Eager, Lazy, Full); Scherer and Scott report this manager to be both fast and stable. We perform incremental revalidation in all systems for all benchmarks, rechecking the consistency of previously opened objects when opening anything new.

5.1 Coarse-Grain Locks and STM

The STM systems considered here have the potential for higher concurrency than lock-based mutual exclusion algorithms, because transactions that access disjoint sets of objects can proceed concurrently without interference. Figure 4 shows performance results for the IntSetHashtable benchmark. This benchmark employs a small hash table (16 buckets) containing up to 256 distinct values with external chaining. Each thread repeatedly performs a lookup, insert, or delete. In this experiment, 90% of the operations are lookups. Coarse-grain locks (implemented using either Java `synchronized` methods or a test-and-`test-and-set` (TATAS) spin lock) perform more than a decimal order of magnitude faster than any STM when there is a single worker thread. This reflects the base case overhead of STM. As the number of threads increases, however, the higher concurrency of STM allows all three systems to outperform both varieties of coarse-grain lock by a significant margin. For small numbers of threads the TATAS lock outperforms the scheduler-based lock, but its performance drops quickly with increasing contention. Beyond 16 threads (the number of processors on the machine), preemption in critical sections further reduces the performance

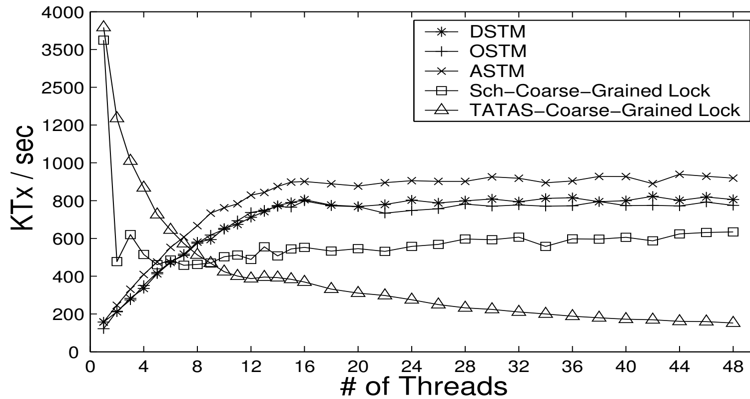

Figure 4: Relative performance of STM systems and coarse-grain locks (`synchronized` methods, test-and-`test-and-set`) on IntSetHashtable with 10% writes

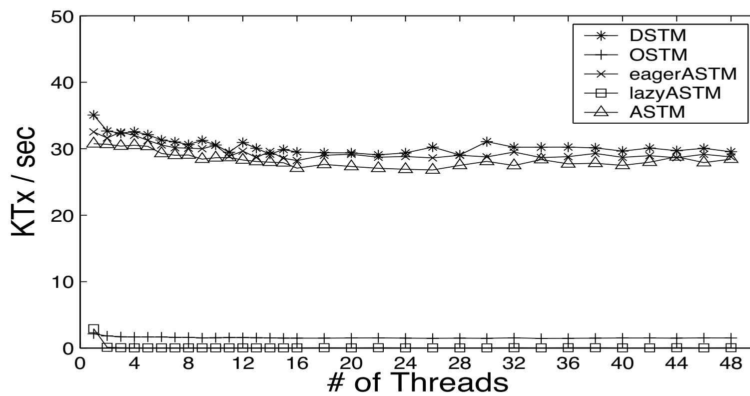

Figure 5: IntSet performance results

of TATAS, but not that of scheduler-based locks: threads that wait in a `synchronized` method yield the processor, and the lock holder soon gets to run again.

It is important to note in this comparison that the coding effort required for the STM-based hash table is comparable to that of using `synchronized` methods for mutual exclusion. A fine-grain lock-based solution might perform significantly better, but at much higher development cost. For more complex data structures (e.g., balanced search trees) this development cost is often prohibitive.

5.2 Write-Dominated Workloads

In DSTM and Eager ASTM, objects opened in write mode are immediately acquired, and the transaction is free to forget the original transactional objects. Validation, moreover, is trivial: If the transaction descriptor is still `ACTIVE`, all objects opened in write mode are consistent. Because they acquire objects lazily, OSTM and Lazy ASTM must maintain a write list in addition to the read list, and must revalidate objects on *both* lists incrementally. ASTM defaults to eager acquire mode, but when it switches to lazy acquire mode, it incurs the same write list maintenance and validation overhead as OSTM and Lazy ASTM.

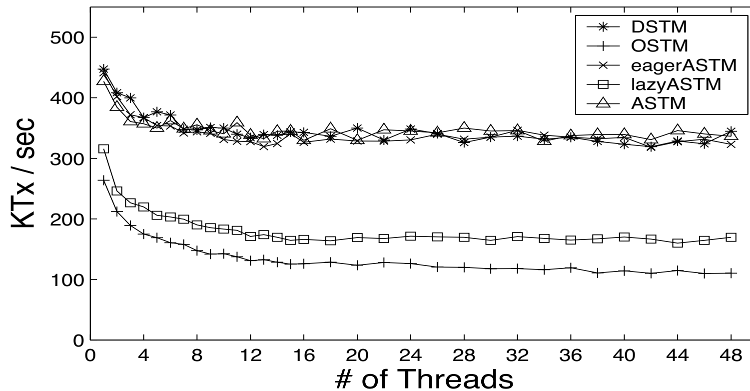

Figure 6: LFUCache performance results

In write-dominated workloads, OSTM and Lazy ASTM suffer significantly from the bookkeeping and incremental validation overhead for maintaining write lists for transactions. DSTM and Eager ASTM, however, do not incur these overheads. Our strategy of having ASTM acquire objects eagerly by default enables ASTM to remain in eager acquire mode for write-dominated transactions. To assess the impact of this design choice, Figure 5 shows STM performance results on the IntSet benchmark. IntSet maintains a sorted list of integers ranging between $0 \ldots 255$ (the range has been restricted in all benchmarks to deliberately increase contention). Concurrent threads continuously insert and delete nodes in the list. A transaction opens all visited nodes in write mode. DSTM, Eager ASTM and ASTM all perform comparably: an order of magnitude better than OSTM and Lazy ASTM. Experiments with modified versions of the code (not shown here) reveal that most of the overhead in OSTM and Lazy ASTM comes from extra bookkeeping; the remainder comes from incremental validation and sorting (in OSTM). Notice that Lazy ASTM livelocks (drops from low throughput to essentially none) when we move from one to two processors.

Figure 6 shows performance results for the LFUCache benchmark [17]. This program uses a priority queue heap to simulate cache replacement in an HTTP web proxy via the least-frequently used (LFU) algorithm [16]. The LFU replacement policy assumes that the most frequently accessed pages are most likely to be accessed next. A transaction typically reads a page, increments its frequency count, and rearranges the heap (if necessary).

To approximate the demand on a real web cache, we pick pages randomly from a Zipf distribution. As a result, a small set of pages is accessed most frequently; transactions are essentially write-dominated (for page i, the cumulative probability of selection is $p_c(i) \propto \sum_{0<j\leq i} 1/j^2$). DSTM, Eager ASTM and ASTM perform comparably. Lazy ASTM suffers from bookkeeping overhead; OSTM additionally incurs overhead from sorting and recursive helping.

5.3 Read-Dominated Workloads

As discussed in Section 3, the style in which unacquired objects are referenced has a considerable impact on read performance. DSTM uses indirection objects (locators), while OSTM arranges for unacquired transactional objects to point to data objects directly. All three ASTM variants adapt their object referencing style to the underlying workload. We illustrate this facet of ASTM's performance relative to OSTM and DSTM using two read-dominated benchmarks: IntSetRelease (another sorted list of integers), and RBTree (a concurrent red-black tree).

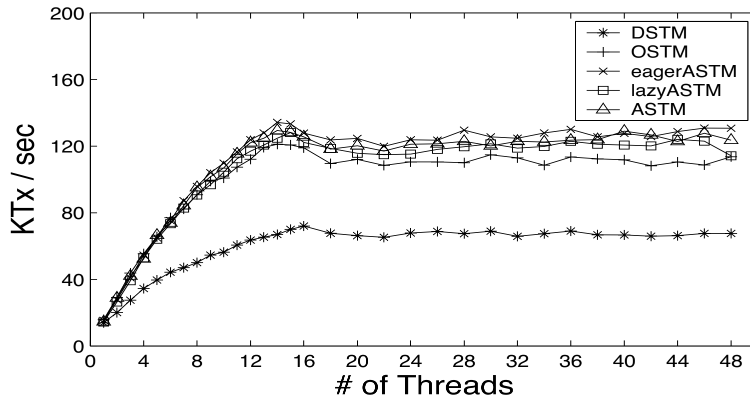

Figure 7: IntSetRelease performance results

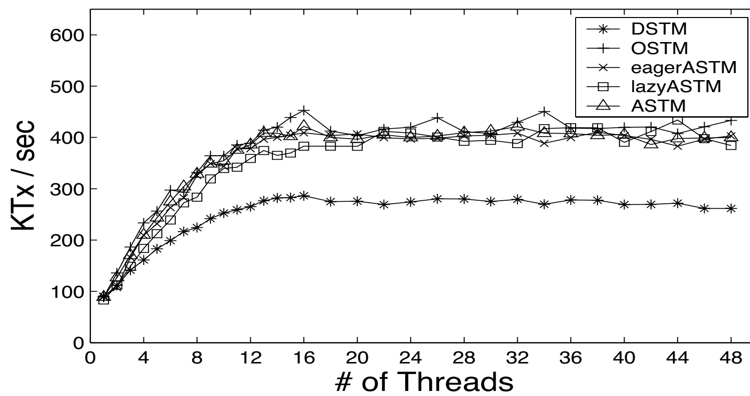

Figure 8: RBTree performance results

IntSetRelease is a highly concurrent IntSet variant in which transactions open objects in read mode and release them, early, while moving down the list. Once found, the nodes to be updated are upgraded to write mode. DSTM performs the worst due to indirection overhead. ASTM and Eager ASTM perform the best, with Lazy ASTM and OSTM incurring minor overheads from bookkeeping (both) and sorting (OSTM). Even though most objects are typically released early, writes always occur at the very end of a transaction. For this reason, ASTM tends to remain in eager acquire mode.

RBTree is a concurrent red-black (balanced) search tree. As in the linked-list benchmarks, threads repeatedly perform randomized insertions and deletions. A transaction typically opens objects in read mode as it searches down the tree for the right place to perform an insertion or deletion. Subsequently, it upgrades to write mode only those nodes that are needed to rebalance the tree. Performance results appear in Figure 8. All the ASTMs and OSTM perform comparably well. Since no early releases are involved in transactions, ASTM remains in eager acquire mode. Since most nodes are never upgraded to write mode, DSTM yields the lowest throughput.

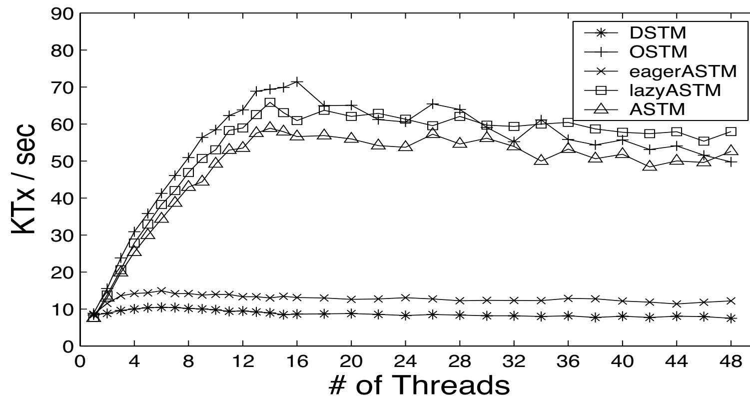

Figure 9: RandomGraphList performance results

5.4 Conflict Window

Although lazy object acquisition leads to bookkeeping and validation overheads, it helps reduce the window of contention between transactions. As per our discussion in Section 4.3, early release can lead to a situation in which lazy acquire might be expected to significantly outperform eager acquire. We see this situation in the RandomGraphList benchmark.

RandomGraphList represents a random undirected graph as a sorted linked list in which each node points to a separate sorted neighbor list. To increase contention, we restrict the graph size to 256 nodes, numbered 0...255. Transactions randomly insert or delete a node. For insertions, transactions additionally select a small randomized neighbor set for the target node. Transactions traverse the adjacency list (using read and early release as in IntSetRelease) to find the target node. They then read and update that node's neighbor list, together with the neighbor list of each of its new neighbors.

Transactions are very long in RandomGraphList. With eager acquire, the window of conflict is very large. Figure 9 shows STM throughputs for RandomGraphList. Lazy acquire is a clear winner, with throughput more than a factor of five higher than that of eager acquire. With eager acquire, reader transactions encounter unnecessary contention with writer transactions (for objects that would be released early anyway), leading to virtually serialized access to the main node list. DSTM bears the additional indirection overhead and hence performs the worst. Here, synergy between lazy acquire and early release significantly reduces the number of read-write conflicts. Eager ASTM performs a little better than DSTM, but significantly worse than Lazy ASTM. Lazy ASTM lags a little behind OSTM up to 16 threads; since data access patterns are random (due to random neighbor sets), transactions in Lazy ASTM apparently livelock on each other for a while before someone makes progress. OSTM, being lock-free, does not suffer from this delay. After 16 threads though, the overhead of recursive helping in OSTM causes reduction in its throughput. ASTM follows Lazy ASTM's performance curve, and is more or less competitive with both OSTM and Lazy ASTM. Because ASTM defaults to eager acquire mode, the first few transactions use eager acquire and are consequently very slow. Thereafter, ASTM switches to lazy acquire mode and starts catching up with Lazy ASTM and OSTM.

5.5 Lock-Freedom v. Obstruction-Freedom

Lock-freedom gives strong progress guarantees that obstruction-freedom does not. On the other hand, obstruction-freedom renders greater simplicity and flexibility. For example, obstruction-freedom gives the

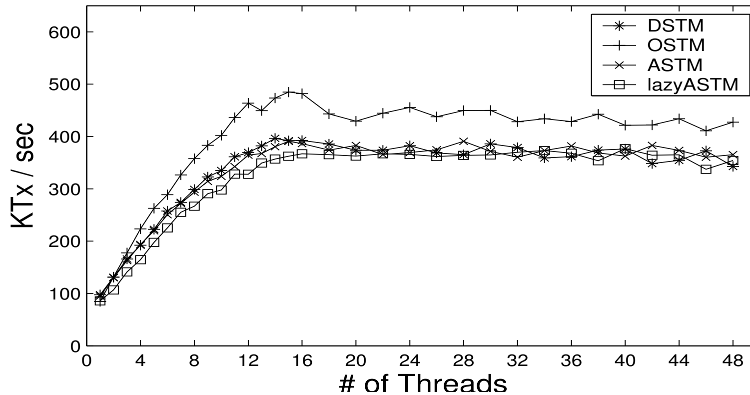

Figure 10: RandomGraphTable performance results

flexibility of choosing acquire semantics in ASTM. OSTM, by virtue of being lock-free, requires lazy acquire and thus the attendant overheads. In this section we present an example benchmark, the RandomGraphTable, that favors lock-freedom.

RandomGraphTable is a variant of RandomGraphList in which the main node list is represented as an array. Each node's neighbor list is still a sorted linked list. A transaction uses a node's ID to index directly into the table. Node lookup time is thus reduced substantially compared to RandomGraphList. Figure 10 shows STM throughputs on RandomGraphTable. All STMs show similar throughput at one thread. With increasing concurrency, however, the obstruction-free STMs (DSTM, ASTM, Eager ASTM, and Lazy ASTM) apparently tend toward partial livelock. OSTM does not suffer this delay. The obstruction-free STMs perform comparably, but noticeably worse than OSTM.

6 Concluding Remarks

The possibility of automatically converting sequential code to concurrent code has made STM an attractive topic of research, and recent work [2, 3, 5, 8] has brought it to the verge of practical utility. Much will now depend on the constants: Is the overhead relative to lock-based code small enough to be outweighed by the (clear and compelling) semantic and software engineering benefits?

In this paper we have presented a detailed analysis of the design space of modern STMs, identifying four key dimensions of STM system design. Various choices in this space lead to varied performance tradeoffs. Consequently no single STM performs the best (or comparably) in all possible scenarios. Our analysis has led us to create a new STM system that adapts to the offered workload, yielding throughput comparable to the best existing system in all scenarios we have tested. The object acquisition methodology, adaptive object referencing style, and obstruction-free nature of ASTM have been carefully selected to give maximum throughput in all cases. We have also demonstrated the feasibility of adapting acquire semantics via simple history-based heuristics.

Although our analysis of the STM design space is fairly exhaustive, another design dimension remains unexplored: tradeoffs in *visible* vs. *invisible* reads [17, 18]. Preliminary results from our exploration of read strategies (not shown here) suggest that adapting between the two may improve performance. We also forsee future work in greater exploration of execution scenarios that favor eager or lazy acquire semantics. Finally, we will strive to find ever better adaptation heuristics.

Acknowledgments

We are grateful to Sun's Scalable Synchronization Research Group for donating the SunFire machine and for providing us with a copy of their DSTM code.

References

[1] G. Barnes. A Method for Implementing Lock-Free Shared Data Structures. In *Proc. of the 5th Ann. ACM Symp. on Parallel Algorithms and Architectures*, pages 261–270, 1993.

[2] K. Fraser. Practical Lock-Freedom. Technical Report UCAM-CL-TR-579, Cambridge University Computer Laboratory, Feb. 2004.

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

[4] R. Guerraoui, M. P. Herlihy, and B. Pochon. Toward a Theory of Transactional Contention Managers. In *Proc. of 24th Ann. ACM Symp. on Principles of Distributed Computing*, July 2005.

[5] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proc. of 18th Ann. ACM Conf. on Object-Oriented Prog., Sys., Langs., and Apps.*, Oct. 2003.

[6] M. P. Herlihy. A Methodology for Implementing Highly Concurrent Data Structures. In *Proc. of the 2nd ACM Symp. on Principles & Practice of Parallel Prog.*, Mar. 1990.

[7] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction Free Synchronization: Double-Ended Queues as an Example. In *Proc. of 23rd Intl. Conf. on Distributed Computing Sys.*, pages 522–529, May 2003.

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

[9] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proc. of the 20th Ann. Intl. Symp. on Computer Architecture*, pages 289–300, May 1993.

[10] M. P. Herlihy and J. M. Wing. Linearizability: a Correctness Condition for Concurrent Objects. *ACM Trans. on Prog. Langs. and Sys.*, 12(3):463–492, 1990.

[11] A. Israeli and L. Rappoport. Disjoint-Access-Parallel Implementations of Strong Shared Memory Primitives. In *Proc. of the 13th Ann. ACM Symp. on Principles of Distributed Computing*, pages 151–160, 1994.

[12] D. Lea. Concurrency JSR-166 Interest Site. http://gee.cs.oswego.edu/dl/concurrency-interest/.

[13] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Design Tradeoffs in Modern Software Transactional Memory Systems. In *Proc. of 14th Workshop on Langs., Compilers, and Run-time Support for Scalable Sys.*, Oct. 2004.

[14] V. J. Marathe, W. N. Scherer III, and M. L. Scott. Adaptive Software Transactional Memory. Technical Report TR 868, Dept. of Computer Science, University of Rochester, May 2005.

[15] M. Moir. Transparent Support for Wait-Free Transactions. In *Proc. of the 11th Intl. Workshop on Distributed Algorithms*, pages 305–319. Springer-Verlag, 1997.

[16] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proc. of the 1990 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Sys.*, pages 134–142, 1990.

[17] W. N. Scherer III and M. L. Scott. Contention Management in Dynamic Software Transactional Memory. In *Proc. of Workshop on Concurrency and Synchronization in Java Progs.*, pages 70–79, July 2004.

[18] W. N. Scherer III and M. L. Scott. Advanced Contention Management in Dynamic Software Transactional Memory. In *Proc. of 24th Ann. ACM Symp. on Principles of Distributed Computing*, July 2005.

[19] N. Shavit and D. Touitou. Software Transactional Memory. In *Proc. of 14th Ann. ACM Symp. on Principles of Distributed Computing*, pages 204–213, Aug. 1995.

[20] J. M. Stone, H. S. Stone, P. Heidelberger, and J. Turek. Multiple Reservations and the Oklahoma Update. *IEEE Parallel and Distributed Technology*, 1(4):58–71, Nov. 1993.

[21] J. Turek, D. Shasha, and S. Prakash. Locking without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking. In *Proc. of the 11th ACM Symp. on Principles of Database Sys.*, pages 212–222, 1992.