Design Tradeoffs in Modern Software Transactional Memory Systems*

Virendra J. Marathe, William N. Scherer III, and Michael L. Scott
Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{vmarathe, scherer, scott}@cs.rochester.edu

ABSTRACT

Software Transactional Memory (STM) is a generic nonblocking synchronization construct that enables automatic conversion of correct sequential objects into correct concurrent objects. Because it is nonblocking, STM avoids traditional performance and correctness problems due to thread failure, preemption, page faults, and priority inversion.

In this paper we compare and analyze two recent object-based STM systems, the DSTM of Herlihy et al. and the FSTM of Fraser, both of which support dynamic transactions, in which the set of objects to be modified is not known in advance. We highlight aspects of these systems that lead to performance tradeoffs for various concurrent data structures. More specifically, we consider object ownership acquisition semantics, concurrent object referencing style, the overhead of ordering and bookkeeping, contention management versus helping semantics, and transaction validation. We demonstrate for each system simple benchmarks on which it outperforms the other by a significant margin. This in turn provides us with a preliminary characterization of the applications for which each system is best suited.

1. INTRODUCTION

A *concurrent* object is a data object shared by multiple threads of control within a concurrent system. Classic lock-based implementations of concurrent objects suffer from several important drawbacks, including deadlocks, priority inversion, convoying, and lack of fault tolerance. Due to these drawbacks, the last two decades have seen an increasing interest in *nonblocking* synchronization algorithms, in which the temporary or permanent failure of a thread can never prevent the system from making forward progress.

*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.

Software Transactional Memory (STM) [16] is a particularly attractive approach to the construction of nonblocking objects, allowing highly concurrent (fine grain) implementations to be created in a purely mechanical way from correct sequential code. Use of an STM system significantly simplifies the task of implementing concurrent objects. 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". There are several challenges involved in achieving that vision, however, including simple (no-contention) overhead, and arbitration among competing transactions when contention is high.

Transactional Memory was originally proposed by Herlihy and Moss as a novel architectural support mechanism for nonblocking synchronization [10]. A similar mechanism was proposed concurrently by Stone et al. [17]. A *transaction* is defined as a finite sequence of instructions (satisfying the linearizability [11] and atomicity properties) that is used to access and modify concurrent objects. Herlihy and Moss [10] proposed the implementation of transactional memory by simple extensions to multiprocessor cache coherence protocols. Their transactional memory provides an instruction set for accessing shared memory locations by transactions. Several groups subsequently proposed software-only mechanisms with similar semantics [1, 2, 3, 12, 16, 18]. Shavit and Touitou [16] coined the term "software transactional memory".

Early STM systems were essentially academic curiosities, with overheads too high to be considered for real-world systems. More recent systems have brought the overhead down to a level where it may be considered an acceptable price for automation, fault tolerance, and clean semantics. The execution overhead of STM comes from the bookkeeping required for transactions. In our experiments, the `IntSet` benchmark (to be discussed later) runs a decimal order of magnitude slower than a version with a global lock, in the absence of contention. At the same time, it scales much better with contention. Herlihy et al. [9] and Fraser [5] show that their STMs are highly scalable, and outperform coarse-grain locks for complex data structures like search trees.

This paper is a comparative analysis of two promising recent object-based realizations of STM: the DSTM [9] of Herlihy et al. and the FSTM [5] of Fraser. Our comparison highlights the key design aspects that differ significantly in these systems, and are mainly responsible for their differing per-

Copyright held by the author/owner

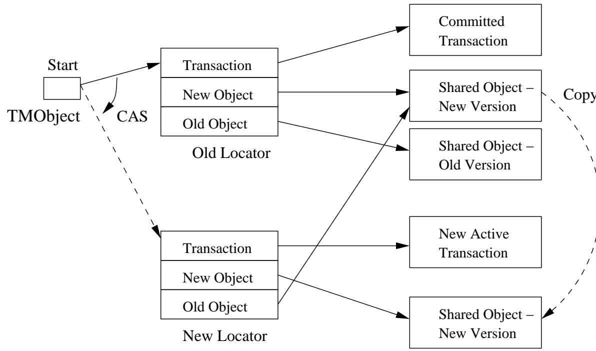

Figure 1: Opening a TMObject recently modified by a committed transaction

formance on various benchmarks. In Sections 2 and 3 we briefly describe the design of DSTM and FSTM respectively. Our comparative evaluation appears in Section 4. We focus in particular on object ownership acquisition semantics, concurrent object referencing style and its effects, the extra overhead of ordering and bookkeeping, contention management versus helping semantics, and transaction validation. The comparison uses experimental results from Java-based implementations to quantify our findings. In Section 5 we present related work. We conclude in Section 6.

2. DSTM

Dynamic Software Transactional Memory (DSTM) [9] supports *obstruction-free* [8] transactions on dynamically selected sets of objects. Obstruction freedom simplifies the implementation by guaranteeing progress only in the absence of contention. Arbitration among competing transactions is handled "out of band" by a separate *contention management* system [15]. To further reduce contention, DSTM also introduces the concepts of *early release*, in which a transaction can drop a previously "opened" object from its atomicity set, and *invisible reads*, which allow a transaction to access an object in *read* mode in such a way that other transactions that access the same object do not detect a conflict.

Figure 1 depicts the architecture of a *Transactional Memory Object (TMObject)* in DSTM. The TMObject acts as a wrapper around a concurrent data object. It contains a pointer to a *locator* object. The locator stores a pointer to a descriptor of the most recent transaction that tried to modify the TMObject, together with pointers to old and new versions of the data object. A transaction descriptor pointed to by a locator may be in any of the three states: `ACTIVE`, `ABORTED`, or `COMMITTED`. If the transaction is `COMMITTED`, the new version referred to by the locator is the most recent valid version of the concurrent object; otherwise the old version is the valid version.

A transaction must access a data object via its wrapper TMObject using the *open* operation. A transaction may open a TMObject in *read* mode or *write* mode. If the object is opened in write mode, the transaction first *acquires* that TMObject. To do so, the transaction creates a new locator object that points to the transaction descriptor, the most recent valid version of the data object, and a newly created copy of the most recent valid version. The transaction then performs an atomic *Compare and Swap (CAS)* on the pointer in the TMObject, to swing it to the new locator. A failure of this CAS implies that some other transaction has opened (acquired) the TMObject in-between, in which case the current transaction must retry the acquire. Figure 1 depicts the open operation of a TMObject after a recent commit.

In the presence of contention, a transaction that wishes to acquire a given object may find that the most recent previous transaction to do so (pointed to by the locator of the TMObject) is still `ACTIVE`. The current transaction can then choose to wait, to abort, or to force the competitor to abort. To make this decision it queries the contention manager. Scherer and Scott have shown that the choice of contention management policy can have a major effect on performance [15].

For read-only access to TMObjects, a full-fledged acquire operation would cause unnecessary contention between transactions. To avoid this contention, each transaction maintains a private list (the *read-list*) of objects it has opened in read-only mode. Because these objects may be modified by other transactions, the current transaction must recheck their validity before attempting to commit. If the use of stale data during a not-yet-committed transaction may lead to incorrect behavior (addressing errors, infinite loops, divide-by-zero, etc.), then a transaction may need to revalidate all open read-only objects whenever it opens another object. The current version of the DSTM performs this validation automatically as part of the open operation. The DSTM designers are currently experimenting with a version of the DSTM in which reads are visible to competing transactions. For visible reads, a TMObject typically contains a *reader transaction list* as well. A reader needs to add itself in this list during the read operation. The acquirer in turn has to traverse this reader list of the target TMObject for resolving read-write conflicts. This version avoids the cost of revalidation, but increases the overhead of the acquire operation; the readers, on the other hand, have to add and remove themselves from the reader list of a target TMObject.

Early release serves to shrink the window of time during which transactions may be recognized as competitors. The consistency of transactions, however, must then be guaranteed by the application programmer, based on object-specific semantics.

3. FSTM

The FSTM system, named after its author, was developed by Keir Fraser at the University of Cambridge as part of his doctoral research [5]. (In more recent work [6], Fraser and Harris refer to the system as OSTM, for Object-based Software Transactional Memory.) Unlike DSTM, FSTM is lock-free, meaning that it guarantees forward progress (though not livelock freedom) for the system as a whole: within a bounded number of steps, from any thread's point of view, *some* thread is guaranteed to complete a transaction. To make this guarantee, FSTM employs *recursive helping*. When a transaction A detects a conflict with an-

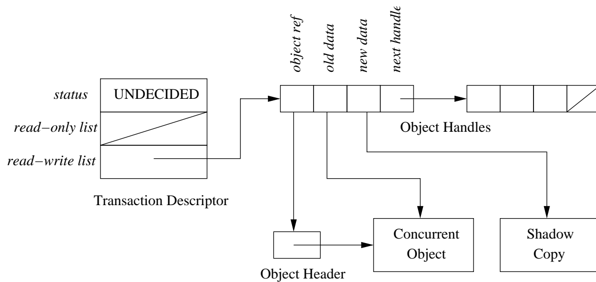

Figure 2: The basic Transactional Memory Structure in FSTM

other transaction B, A uses B's transaction descriptor to make B's updates on its behalf and then typically aborts and restarts itself. Consequently, even if B's owner thread terminates or stalls halfway through completion of B, other threads may *help* complete transaction B.

Each concurrent object in FSTM is wrapped in an *object header*. A transaction gains access to objects by *opening* their corresponding headers. Each transaction uses a *transaction descriptor* to maintain a list of in-use objects. Along with a transaction status flag, the transaction descriptor contains a *read-only list* for the objects opened in read mode and a *read-write list* for the objects opened in write mode. Both lists contain *object handles*. An object handle contains references to the object header, the object itself, and a *shadow copy* of the object. The transaction performs all its updates on the shadow copy, which is local to the transaction. In contrast to the conventions of DSTM, multiple transactions may open the same object in write mode; each has its own shadow copy.

A transaction may be in any of four states: `UNDECIDED`, `ABORTED`, `COMMITTED`, or `READ-CHECKING`. A transaction always begins in the `UNDECIDED` state. Figure 2 depicts an example transaction descriptor used by a transaction to access an object in write mode. The object header is a simple pointer to the concurrent object, through which any other interested transaction would attempt to access the object. What is not clear from the figure is that the object header may also point to a transaction descriptor when the corresponding transaction *acquires* the object header.

A transaction *opens* object headers while in the `UNDECIDED` state, creating object handles and adding them to the read-only or read-write list, as appropriate. The fact that the objects are open does not become visible to other transactions, however, until the current transaction enters its *commit phase*. If a conflict is detected, the transaction recursively helps the conflicting transaction.

A transaction in commit phase first *acquires* all objects it has opened in write mode, in some global total order (typically based on virtual address—this requires a sorting step) using atomic CASes. Each CAS replaces the pointer in the object header with a pointer to the acquiring transaction's descriptor. Pointers are *tagged* in a low-order bit[1] to indicate whether they refer to an object or a transaction descriptor. If an acquiring transaction discovers a conflict (the pointer it is attempting to change already points to the descriptor of some competing transaction), the acquiring transaction recursively helps the competitor. Global total ordering ensures that there will be no helping cycles.

[1]Our Java implementation uses the `instanceof` operator to identify the pointer type.

After a successful acquire phase, the transaction atomically switches (via CAS) to the `READ-CHECKING` state and validates the objects in its read-only list. Validation consists of verifying that the object header still refers to the version of the object that it did when the object handle in the read-only list was created. If it refers to a different version, the transaction must abort. If it refers to the descriptor of a competing transaction, the current transaction may again perform recursive helping. Recursive helping proceeds only if the competitor transaction precedes the potential helper in some global total order (e.g., based on thread and transaction ids). Additionally, the competitor must also be in its `READ-CHECKING` state. If the potential helper precedes the competitor, the competitor is aborted. The current transaction then proceeds (even in the case where the competitor is in `UNDECIDED` state) if the competitor's old version of the object was the same as the one in the current transaction's object handle; otherwise the current transaction must also abort.

After successful validation in the `READ-CHECKING` state, the transaction atomically switches to the `COMMITTED` state and *releases* the acquired objects by swinging their object handles to refer to the new version of the data.

4. COMPARATIVE EVALUATION

In the preceding two sections we sketched designs of recent object-based STM systems, the DSTM and the FSTM. While both have significant constant overhead, they are substantially simpler than previous approaches—enough to make them serious candidates for practical use, particularly given the semantic and software engineering advantages of nonblocking algorithms relative to lock-based alternatives. In this section we highlight the design tradeoffs embodied by the two designs, and their impact on the performance of various concurrent data structures. A more detailed but qualitative comparison can be found in our earlier technical report [14].

Our experimental results were obtained 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 [13].

We report results for four simple benchmarks (a stack and three variants of a list-based set) and one slightly more complex benchmark (a red-black tree). In the list and red-black benchmarks, threads repeatedly but randomly insert or delete integers in the range 0. . . 255 (keeping the range small increases the probability of contention).

We measured the total throughput over 10 seconds for each benchmark varying the number of worker threads between 1 and 48. Results were averaged over a set of six test runs. In all the experiments we have used the Polite contention

manager for DSTM, which performs reasonably well [15] on the benchmarks we have chosen. Except where stated otherwise, we perform incremental validation in both DSTM and FSTM, rechecking the consistency of previously opened objects when opening something new.

4.1 Object Acquire Semantics

In DSTM, a transaction acquires exclusive (though abortable) access to an object when it first opens it for write access. Because this strategy makes the transaction visible to potential competitors early in its lifetime, we call it *eager acquire*. In FSTM, transactions acquire exclusive access only when they enter their commit phase. This strategy makes the transaction visible to potential competitors much later in its lifetime; we call it *lazy acquire*. Eager acquire enables earlier detection and resolution of conflicts than lazy acquire; lazy acquire may cause transactions to waste significant computational resources on doomed transactions before detecting a conflict. Another advantage of eager acquire is that unrelated threads can make progress faster if the thread detecting an early conflict decides to yield the processor. On the flip side, lazy acquire tends to minimize the window during which transactions may be identified as competitors; if application semantics allow both transactions to commit, lazy acquire may result in significantly higher concurrency.

DSTM could in principle be modified to use lazy acquire. The open operation would not contain a CAS, but the commit operation would require a heavyweight "multi-word CAS" like that of FSTM. FSTM, however, could not be modified to use eager acquire without abandoning the dynamic selection of objects to be modified. Lock freedom in FSTM requires that objects be acquired in global total order, to avoid cyclic helping. Only after all objects have been opened can a transaction in general know which objects it will need; only then can it acquire them in order. In short, lazy acquire is necessary to the lock-free semantics of dynamic transactions in FSTM; DSTM is able to use eager acquire because it is merely obstruction-free.

In this subsection we present performance results of the red-black tree benchmark (RBTree) as a plausible candidate example of the direct impact of acquire semantics. Red-black trees provide a beautiful illustration of STM's software engineering benefits: using DSTM or FSTM we can construct a correct, highly concurrent red-black tree via simple mechanical transformation of correct sequential source. A comparable implementation using explicit fine-grain locks is notoriously difficult to write.

Figure 3 plots throughput (in transactions/sec.) against concurrency for DSTM and FSTM versions of RBTree. We conjecture that the higher performance of FSTM here is a direct result of lazy acquire semantics. As illustrated in Figure 4, DSTM's wider contention window significantly increases the number of times that transactions are judged to be in conflict; this number climbs steadily with the level of true concurrency (recall that our machine has 16 processors).

It is conceivable that a better contention manager would improve the performance of DSTM, but Scherer and Scott report [15] that the Polite contention manager (used in these experiments) is among the best performers on RBTree. As a cross check we tested other top contention managers (Karma and Kindergarten [15]), with similar results.

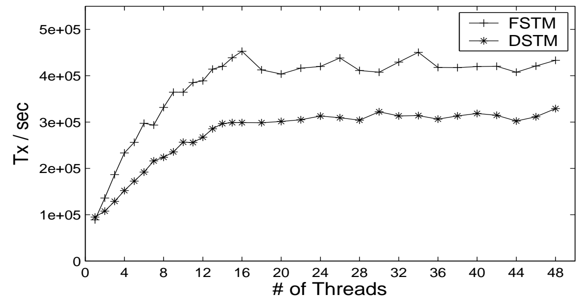

Figure 3: RBTree Performance Results

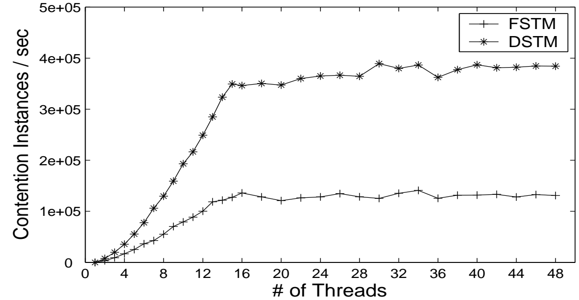

Figure 4: RBTree – Number of Contention Instances

Fraser [5] also compared the performance of FSTM and DSTM on a red-black tree benchmark. While he, too, found that FSTM provided higher throughput, the difference remained more or less constant at all thread counts. We attribute the difference to the implementations of FSTM and DSTM. Our versions are written in Java and rely on automatic garbage collection, whose performance does not scale with the number of threads. Fraser's versions are written in C and use manual storage reclamation, which parallelizes nicely.

4.2 Bookkeeping and Indirection

A comparison of Figures 1 and 2 reveals an extra level of indirection in DSTM [5]: where an FSTM object header points to the object data, a DSTM TMObject points to a Locator, which in turn points to the object data. In effect, FSTM maintains a locator (object handle) only for objects being used by some transaction; it chains these off the transaction descriptor. The extra indirection of DSTM may result in slower reads and writes of open objects—and thus slower transactions—when contention is low, particularly if most transactions are read-only.

At the same time, indirection allows DSTM to commit with a single CAS on the status field of the transaction's descriptor. FSTM requires a substantially more complex multi-word CAS. In the absence of contention, if a transaction updates N concurrent objects, DSTM requires a total of $N+1$ CAS operations; FSTM requires $2N+2$. Indirection also eliminates the overhead of inserting object han-

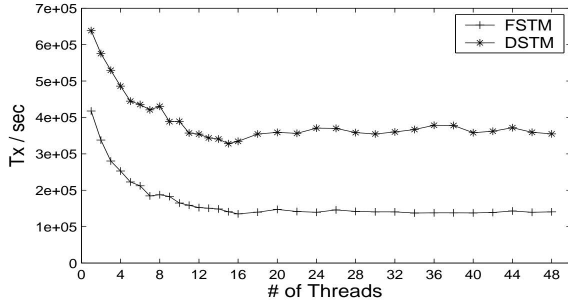

Figure 5: Stack Performance Results

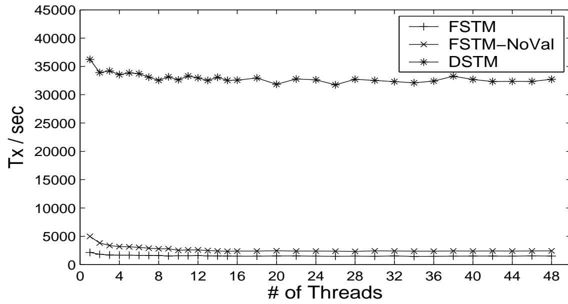

Figure 6: IntSet Performance Results

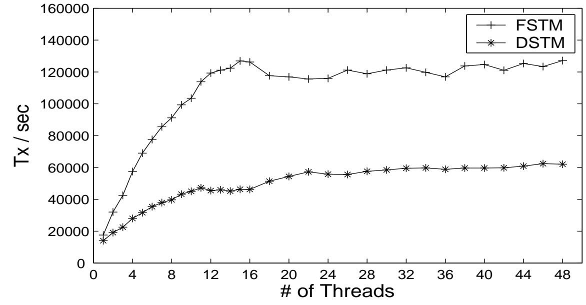

Figure 7: IntSetRelease Performance Results

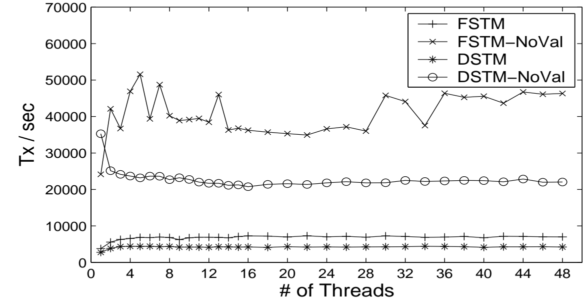

Figure 8: IntSetUpgrade Performance Results

dles into transaction descriptor chains. Transactions with a large number of writes may be faster in DSTM. (Our implementation keeps the FSTM write list sorted, so insertion takes linear time, but we use an auxiliary hash table for fast lookups.). Here, the eager acquire semantics in DSTM serve as a foundation to exploit the benefits of the extra level of indirection for faster writes. Lazy acquire semantics mandate extra bookkeeping as in FSTM.

We use the Stack, IntSet, and IntSetRelease benchmarks to illustrate the impact of these overheads. Stack illustrates very high contention for a single word (the top-of-stack pointer). As shown in Figure 5, DSTM outperforms FSTM by a factor of more than two. We attribute this difference to the extra bookkeeping, in FSTM, associated with opening objects in *write* mode.

IntSet maintains a sorted list. Every *insert* or *delete* operation opens all objects from the beginning of the list in write mode. As a result, successful transactions are serialized. Figure 6 shows an order of magnitude higher throughput for DSTM. FSTM suffers from extra bookkeeping overhead, sorting overhead, and extra CASes, in decreasing order of significance. We initially suspected that incremental validation might account for much of the performance difference in IntSet: FSTM must inspect the handle of every open object, but in the absence of read-only objects, DSTM need only verify that the current transaction has not yet aborted. As it turns out, however, this extra overhead in FSTM is dwarfed by sorting and bookkeeping. The third curve in Figure 6 reveals less than a two-fold improvement in throughput when validation is removed from FSTM.

IntSetRelease is a variant of IntSet in which only the object to be modified is opened in write mode; all other objects are opened temporarily in read mode and then either *released* (when moving on to the next node in the list) or *upgraded* to write mode. In this case results strongly favor FSTM: not only does it outperform DSTM by more than a factor of two; overall throughput is also higher than that of DSTM in IntSet by nearly a factor of four due to a higher degree of concurrency. The explanation appears to lie with DSTM's overhead due to the extra level of indirection.

4.3 Contention Management and Helping

FSTM employs recursive helping to ensure lock freedom. Although lock freedom may be desirable from a semantic point of view, obstruction freedom tends to simplify the implementation of ad hoc concurrent objects, and may lead to better performance [8]. In particular, helping has been shown in some applications to lead to significant thrashing of cache blocks. Contention management in obstruction-free algorithms, however, requires nontrivial overhead (5–10% in our experiments), and must be carefully designed to eliminate livelock. Careful evaluation of these tradeoffs is a subject for future research. In particular, there may be value in providing helping as an option (in addition to waiting and aborting) in a contention management system.

4.4 Transaction Validation

Invisible reads and (in the case of FSTM) lazy acquire may allow a transaction to enter an inconsistent state during its execution. Inconsistency in turn may lead to memory access violations, infinite loops, arithmetic faults, etc. In certain cases the programmer may be able to reason that consis-

tency is not a problem, but this is not a general-purpose solution. Herlihy et al. [9] therefore perform incremental validation automatically at open time in DSTM. As an alternative, Fraser proposes a mechanism based on exception handling [5] to catch problems when they arise, rather than prevent them. On a memory access violation, the exception handler is designed to validate the transaction that caused the exception. The responsibility of detecting other inconsistencies is left to the application programmer.

We use the IntSet and IntSetUpgrade benchmarks to evaluate the cost of incremental validation. We have already discussed IntSet. IntSetUpgrade, like IntSetRelease, acquires a write lock on only the nodes that need to be changed. It does not, however, perform an early release on nodes that have been passed. The fact that most open nodes are read-only introduces some concurrency (if transaction A starts down the list and then transaction B passes it, B can make a modification toward the end of the list without causing A to abort). The invisibility of reads in DSTM, however, means that both DSTM and FSTM require validation overhead at open time linear in the number of already open nodes—quadratic time overall.

Figure 8 depicts the performance of IntSetUpgrade under the two STM systems with and without incremental validation. FSTM outperforms DSTM in both cases, by roughly a factor of two, as a result of DSTM's indirection costs. For both systems, however, the cost of validation is dramatic: throughput increases by roughly a factor of five if we forgo validation. These results suggest substantial potential benefits from using application-specific reasoning to eliminate the need for (or at least the frequency of) incremental validation. Unfortunately, the need for such reasoning is strongly counter to the software engineering goals of STM. Moreover the IntSet benchmark illustrates that simply removing incremental validation (if possible) may not yield as high throughput as expected. In fact, since incremental validation is a tool for early detection of conflicts, the performance of some benchmarks may degrade with the removal of incremental validation.

5. RELATED WORK

After a flurry of activity in the early to mid 1990s (cited in Section 1), STM research went largely dormant until the last few years. We focus here on recent work.

Harris and Fraser [7] have proposed a *word-based* STM that hashes shared memory words into *ownership records*. A transaction acquires these ownership records before making any updates to the corresponding shared words. Contention is resolved by abort the conflicting transaction, so the system as a whole is obstruction-free. (We conjecture that one could introduce contention management mechanisms here to increase throughput significantly [15].) Harris and Fraser also introduce a novel *stealing* mechanism for corner cases to avoid the cache thrashing that might result from recursive helping. Marathe and Scott [14] have proposed an alternative method of stealing and helping, using the load-linked and store-conditional instructions, that reduces the complexity of the word-based STM significantly. Their method also resolves a scalability issue of the stealing mechanism proposed by Harris and Fraser.

Cole and Herlihy [4] propose an optimization to the DSTM to reduce the bookkeeping overhead for objects opened in *read* mode. We conjecture that this optimization could be extended to FSTM as well. Scherer and Scott [15] focus on the contention management problem in DSTM. They propose and evaluate several different contention managers, and show that performance depends critically on choosing the right one for a given application. Our work in this paper differs from these contributions in that we focus on the higher level design decisions of DSTM and FSTM, showing their impact on overall performance for various concurrent data structures.

6. CONCLUSIONS

In this paper we evaluated tradeoffs in the design of practical, object-based STM systems [5, 9]. DSTM tends to do better—potentially much better—for transactions that open objects mostly in write mode, due to both the early detection of conflicts and the avoidance of bookkeeping overhead. For transactions that open objects mostly in read-only mode, both systems incur significant bookkeeping overhead, but FSTM does not pay the ordering cost that it does with writes, while DSTM still has to pay for its extra level of indirection. For the benchmarks we considered the result is a roughly two-fold throughput advantage for FSTM. Our experiments were all conducted with invisible reads. Using visible reads in DSTM [15] might significantly alter performance; this is a topic for future research.

Acquire semantics play a key role in the relative performance of DSTM and FSTM. Eager acquire tends to help in the early detection of conflicts, whereas lazy acquire reduces the window in which transactions may be seen as competitors. Lazy acquire also permits the use of ordering, allowing FSTM to offer lock-free semantics. Eager acquire, on the other hand, helps reap the benefits of the extra level of indirection for faster writes in DSTM.

Incremental validation relieves the programmer from the burden of ensuring intra-transaction consistency, but incurs significant costs. Automatic techniques to reduce the cost of validation seem eminently worth pursuing. Other topics of future interest include experiments with additional data structures and applications, comparison to lock-based implementations, and the development of compiler support for the automatic construction of STM-based nonblocking objects.

Acknowledgment

We are grateful to the Scalable Synchronization Group at Sun Microsystems Laboratories, Boston, for donating the SunFire machine, and for providing us with a copy of their DSTM system.

7. REFERENCES

[1] J. H. Anderson and M. Moir. Universal Constructions for Large Objects. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 168–182. Springer-Verlag, 1995.

[2] J. H. Anderson and M. Moir. Universal Constructions for Multi-Object Operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193, 1995.

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

[4] C. Cole and M. P. Herlihy. Snapshots and Software Transactional Memory. In *Proceedings of Workshop on Concurrency and Synchronization in Java Programs*, 2004.

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

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

[7] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, 2003.

[8] M. P. Herlihy, V. Luchangco, and M. Moir. Obstruction Free Synchronization: Double-Ended Queues as an Example. In *Proceedings of 23rd International Conference on Distributed Computing Systems*, pages 522–529, May 2003.

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

[10] M. P. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

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

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

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

[14] V. J. Marathe and M. L. Scott. A Qualitative Survey of Modern Software Transactional Memory Systems. Technical Report TR 839, Department of Computer Science, University of Rochester, June 2004.

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

[16] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[17] 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, November 1993.

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