

# Understanding Tradeoffs in Software Transactional Memory

Dave Dice  
Sun Microsystems

Nir Shavit  
Tel-Aviv University and Sun Microsystems Research

## Abstract

*There has been a flurry of recent work on the design of high performance software and hybrid hardware/software transactional memories (STMs and HyTMs). This paper re-examines the design decisions behind several of these state-of-the-art algorithms, adopting some ideas, rejecting others, all in an attempt to make STMs faster.*

*We created the transactional locking (TL) framework of STM algorithms and used it to conduct a range of comparisons of the performance of non-blocking, lock-based, and Hybrid STM algorithms versus fine-grained hand-crafted ones. We were able to make several illuminating observations regarding lock acquisition order, the interaction of STMs with memory management schemes, and the role of overheads and abort rates in STM performance.*

## 1 Introduction

A goal of current multiprocessor software design is to introduce parallelism into software applications by allowing operations that do not conflict in accessing memory to proceed concurrently. The key tool in designing concurrent data structures has been the use of locks. Unfortunately, coarse-grained locking is easy to program with, but provides very poor performance because of limited parallelism. Fine-grained lock-based concurrent data structures perform exceptionally well, but designing them has long been recognized as a difficult task better left to experts. If concurrent programming is to become ubiquitous, researchers agree that one must develop alternative approaches that simplify code design and verification. This paper is interested in rote, “mechanical”, methods for transforming sequential code or coarse-grained lock-based code into concurrent code. By mechanical we mean that the transformation, whether done by hand, by a preprocessor, or by a compiler, does not require any program specific information (such as the programmer’s understanding of the data flow relationships). Moreover, we wish to focus on techniques that can be deployed to deliver reasonable performance across a wide range of systems today, yet combine easily with spe-

cialized hardware support as it becomes available.

### 1.1 Transactional Programming

The *transactional memory* programming paradigm [20] is gaining momentum as the approach of choice for replacing locks in concurrent programming. Combining sequences of concurrent operations into atomic transactions seems to promise a great reduction in the complexity of both programming and verification, by making parts of the code appear to be sequential without the need to program fine-grained locks. Transactions will hopefully remove from the programmer the burden of figuring out the interaction among concurrent operations that happen to conflict when accessing the same locations in memory. Transactions that do not conflict in accessing memory will run uninterrupted in parallel, and those that do will be aborted and retried without the programmer having to worry about issues such as deadlock. There are currently proposals for hardware implementations of transactional memory (HTM) [3, 12, 20, 33], purely software based ones, i.e. software transactional memories (STM) [10, 14, 19, 24, 25, 29, 34, 35], and hybrid schemes (HyTM) that combine hardware and software [4, 23, 29].<sup>1</sup>

The dominant trend among transactional memory designs seems to be that the transactions provided to the programmer, in either hardware or software, should be “large scale”, that is, unbounded, and dynamic. *Unbounded* means that there is no limit on the number of locations accessed by the transaction. *Dynamic* (as opposed to *static*) means that the set of locations accessed by the transaction is not known in advance and is determined during its execution.

Providing large scale transactions in hardware tends to introduce large degrees of complexity into the design [20, 33, 3, 12]. Providing them efficiently in software is a difficult task, and there seem to be numerous design parameters and approaches in the literature [10, 14, 19, 25, 29, 34, 36], as well as requirements to combine well with hardware transactions once those become available [4, 23, 29].

---

<sup>1</sup>A broad survey of prior art can be found in [14, 24, 32].

## 1.2 Software Transactional Memory

The first STM design by Shavit and Touitou [35] provided a non-blocking implementation of static transactions. They had transactions maintain transaction records with read-write information, access locations in address order, and had transactions help those ahead of them in order to guarantee progress. The first non-blocking dynamic schemes were proposed by Herlihy et al. [19] in their dynamic STM (DSTM) and by Fraser and Harris in their object-based STM [14] (OSTM). The original DSTM was an excellent proof-of-concept, and the first obstruction-free [19] STM, but involved two levels of indirection in accessing data, and had a costly implementation in the Java™ programming language. This implementation was improved on later by the ASTM of Marathe et al. [25]. The OSTM of Fraser and Harris took a slightly different programming approach than DSTM, requiring programmers to open and close objects within a transaction in order to improve performance based on the programmer’s understanding of the data structure being implemented. We found that the latest C-based versions of OSTM, which involve one level of indirection in accessing data, are the most efficient non-blocking STMs available to date [14]. A key element of being non-blocking is the maintenance of publicly shared transaction records with undo or copy-back information. Lock-based STMs are relieved of that burden. As our empirical data will show however, OSTM performs reasonably well across the concurrency range.

A recent paper by Ennals [10] suggested that on modern operating systems, deadlock avoidance is the only compelling reason for making transactions non-blocking, and there is little reason to provide non-blocking transactions at user-level. We agree with Ennals in the sense that we believe blocking STM implementations can provide many of the desirable properties of non-blocking STMs. For instance preemption of lock holders can be mitigated by existing mechanisms such as solaris’ *schedctl* which allows threads to transiently defer preemption. Alternatively, when encountering a lock held by a preempted thread, the contending thread might yield its quantum to the lock holder in order that it make progress and drop the lock. Ennals [10] proposed an all-software lock-based implementation of software transactional memory using the object-based approach of [15]. His idea was to have transactions acquire write locks as they encounter locations to be written, writing the new values in place and having pointers to an undo set that is not shared with other threads (we call this approach *encounter-order*). A transaction collects a read-set which it validates before committing and releasing the locks. If a transaction must abort, its executing thread can restore the values before releasing the locks on the locations being written. The use of locks eliminates the need for indirect

tion and shared transaction records as in many of the non-blocking STMs, it still requires, however, a closed memory system. A *closed memory system* is one that does not allow objects recycled by transactional methods to be reused by non-transactional methods, and vice versa. Deadlocks and livelocks are dealt with using timeouts and the ability of transactions to request other transactions to abort. As we show, Ennals’s algorithm exhibits impressive performance on several benchmarks. It is not clear why his work has not gained more recognition. A recent paper by Saha et al. [34], concurrent and independent of our own work, uses a version of the Ennals’s lock-based algorithm within a run-time system. It uses encounter-order, and maintains undo sets that are shared to allow transactions to actively abort others. Moir et al. [29] were the first to suggest that the pointers to transaction records (called “Orecs”) in non-blocking transactions could be used to coordinate hardware and software transactions to form hybrid transactional schemes. Their algorithm acquires locks as they are encountered and uses a write set instead of an undo set.

Our paper reexamines the design decisions behind these state-of-the-art STM algorithms. Building on the body of prior art together with our new understanding of what makes software transactions fast, we introduce the *transactional locking* (TL) algorithm which we believe to be the simplest and best performing STM to date.

## 1.3 Our Findings

We focus on the performance of small transactions. It is unclear to us if large optimistic transactions are viable given their increased vulnerability to interference. Large transactions are also more likely to embed I/O operations which, can not be rolled back at abort time. As such, different mechanisms such as conservative transactions that still interoperate with small transactions may be needed.

The following are some of the results and conclusions presented in this paper for systems with small transactions.

- Ennals [10] suggested building deadlock-free lock-based STMs rather than non-blocking ones [14, 29]. Our empirical findings support Ennals’s claims: non-blocking transactions [14, 29] were less efficient than our TL lock-based ones on a variety of data structures and across concurrency ranges, even when they used an advantageous non-mechanical programming interface [14] that allows opening and closing objects. Given that, as we show, locks provide a simple interface to hardware transactions, we recommend that the design of HyTMs give due consideration to lock-based algorithms.
- Both Ennals and Saha et al. [10, 34] have transactions acquire write locks as they encounter them and

use an “undo set” to recover from failures. Saha et al. [34] claim that this is a conscious design choice. Both of the above papers failed to observe that *encounter-order* transactions perform well on uncontended data structures but degrade on contended ones. We use variations of our TL algorithm to show that this type of degradation is not a fluke, and is possibly inherent to encounter-order lock acquisition.

- In its default operational mode, our new TL algorithm acquires locks only at *commit-time*, using a Bloom filter [5] for fast look-aside into the write-buffer to allow reads to always view a consistent state of its own modified locations. Slow look-aside was cited by Saha et al. [34] as a reason for choosing encounter-order locking and undo writing in their algorithm (one should note though that we do not support nesting in our STM). As we explain, the undo set used in the Ennals’ style encounter order algorithms requires type-stable memory or specialized malloc/free implementations. Unlike the undo set, the write-set approach used by our commit-time locking fits well with the memory lifecycle in languages like C and C++, allowing transactionally accessed memory to be moved in and out of the general memory pool using *regular* malloc and free operations. We note that one could also use a write-set with encounter order locking as in Moir et al. [29], but the opposite is not true: one cannot implement an undo-set with commit time locking.
- Of all the algorithms we tested, lock-free, or lock-based, the TL algorithm which acquires locks at commit-time exhibits the best scalability across most contention ranges. Moreover, we found the advantage of encounter-order algorithms, when they do exhibit better performance, to be small enough so as to bring us to conclude that even from a pure performance standpoint, one should always default to using commit-time locking.
- Both Ennals and Saha et al. [10, 34] provide mechanisms for one transaction to abort another to allow progress. In the case of Saha et al. this mechanism might add a significant cost to the implementation because write-sets must be shared so one transaction can completely undo another. We claim these mechanisms are unnecessary, and show that they can be effectively replaced by time-outs.
- Perhaps most importantly, and contrary to our initial intuition, for the data structures we tested, it is the overhead of the STM implementations (measured, for example, by single thread performance cost) that is best correlated with overall performance, not the superior contention management hand-crafted structures

can deliver based on the programmer’s understanding of the data structures (This is not to say that there aren’t structures where hand-crafting will increase scalability to a point where it dominates performance). Lower overheads benefit transactions in two ways: (1) shorter transactions are less exposed to interference and (2) shorter transactions imply a higher rate of arrival at the commit point. We are in the process of collecting more data to support this claim.

- Finally, our findings bode well for HTM support, which we expect will suffer from the same abort rates as our TL algorithm, yet will reduce the overhead of operations significantly. For HTM designers, our findings suggest that hardware transactional design should focus on overhead reduction. However, interestingly enough, some of the preliminary data we present seems to suggest that providing only read-set validation support in hardware [8], as opposed to full blown HTM, may provide sufficient “bang for the buck.”

## 2 Transactional Locking

According to the transactional locking approach, rather than trying to improve on hand-crafted lock-based implementations by being non-blocking, we try to build lock-based STMs that will get us as close to their performance as one can with a completely mechanical approach, that is, one that simplifies the job of the concurrent programmer.

Our algorithm operates in two modes which we will call *encounter* mode and *commit* mode. These modes indicate how locks are acquired and how transactions are committed or aborted. We will begin by describing our commit mode algorithm, later explaining how TL operates in encounter mode similar to algorithms by Ennals [10] and Saha et al [34]. The availability of both modes will allow us to show the performance differences between them.

We associate a *versioned lock-word* with every transacted memory location. The mapping between transacted upon locations and versioned lock-words can be either many to one or one to one; each transacted upon location is associated with (we say “covered by”) a single lock-word. Our lock-words are simple single-word spinlocks, similar to a Linux “seqlocks”, that use an atomic compare-and-swap (CAS) to acquire the lock and an ordinary store to release it. Since one needs only a single bit to indicate that the lock is taken we use the rest of the word to hold a version number. This number is incremented by every successful lock release. Lock-words constitute transactional metadata and are not exposed to the application programmer.

We allocate a collection of versioned lock-words. We use various schemes for associating locks with shared memory:

*per object* (PO), where a lock is assigned per shared object, *per stripe* (PS), where we allocate a separate large array of locks and memory is stripped (divided up) using some hash function to map each location to a separate stripe, and *per word* (PW) where each transactionally referenced variable (word) is collocated adjacent to a lock. Other mappings between transactional shared variables and locks are possible. The PW and PO schemes require either manual or compiler-assisted automatic insertion of lock fields whereas PS can be used with unmodified data structures. Since in general PO showed better performance than PW we will focus on PO and do not discuss PW further. PO might be implemented, for instance, by leveraging the header words of objects in the Java programming language [2]. A single PS stripe-lock array may be shared and used for different TL data structures within a single address-space. For instance an application with two distinct TL red-black trees and three TL hash-tables could use a single PS array for all TL locks. As our default mapping we chose an array of  $2^{20}$  entries of 32-bit lock words with the mapping function masking (applying a bitwise “And”) the variable address with “0x3FFFFC” and then adding in the base address of the lock array to derive the lock address.

The following is a description of the PS algorithm although most of the details carry through verbatim for PO and PW as well. We maintain thread local read- and write-sets as linked lists. A read-set entry contains the address of the lock and the observed version number of the lock associated with the transactionally loaded variable. A write-set entry contain the address of the variable, the value to be written to the variable, and the address of the associated lock. The write-set is kept in chronological order to avoid write-after-write hazards.

## 2.1 Commit Mode

We now describe how TL executes a sequential code fragment that was placed within a TL transaction. We use our preferred commit mode algorithm. As we explain, because of the use of a read-set (rather than an undo-set) this mode does not require type-stable garbage-collection, and works seamlessly with the memory life-cycle of languages like C and C++.

1. Run the transactional code, reading the locks of all fetched-from shared locations and building a local read-set and write-set.

A transactional load first performs a look-aside check (using a Bloom filter [5]) to see if the load address appears in the write-set. If so the transactional load returns the last value written to the address. This will allow us to provide linearizability [21] and avoids so-called read-after-write hazards. If the address is not

found in the write-set the load operation then fetches the lock value associated with the variable, saving the version in the read-set, and then fetches from the actual shared variable. If the transactional load operation finds the variable locked the load may either spin until the lock is released or abort the operation.

During the operation of the transaction we periodically validate the read-set. If the read-set is found to be invalid we abort the transaction. We say that a transaction that is still running after having read an inconsistent view of global data is a *zombie* transaction. Such zombies are unable to successfully commit. Periodic read-set validation avoids zombies being trapped in infinite loops. We use *safe* loads to avoid dereferencing invalid pointers arising from zombie execution. (On SPARC™ safe loads are implemented with an alternative load operation. Elsewhere, they can be implemented in software with the assistance of trap handlers that either step over the faulting instruction or directly restart the afflicted transaction).

2. Attempt to commit the transaction. Acquire the locks of locations to be written. If a lock in the write-set (or more precisely a lock associated with a location in the write-set) also appears in the read-set then the acquire operation must atomically (a) acquire the lock *and*, (b) validate that the current lock version subfield agrees with the version found in the earliest read-entry associated with that same lock. An atomic CAS can accomplish both (a) and (b). Acquire the locks in any convenient order using bounded spinning to avoid indefinite deadlock. If bound is reached, release the locks, abort the transaction, and retry.
3. Re-read the locks of all read-only locations to make sure version numbers haven't changed. If a version does not match, roll-back (release) the locks, abort the transaction, and retry.
4. The prior observed reads in step (1) have been validated as forming an atomic snapshot of memory [1]. The transaction is now *committed*. Write-back all the entries from the local write-set to the appropriate shared variables.
5. Release all the locks identified in the write-set by atomically incrementing the version and clearing the write-lock bit (using a simple store).

A few things to note. The write-locks have been held for a brief time when attempting to commit the transaction. This helps improve performance under high contention. The Bloom filter allows us to determine if a value is not in the write-set and need not be searched for by reading the single filter word. Though locks could have been acquired

in ascending address order to avoid deadlock, we found that sorting the addresses in the write-set was not worth the effort. We intend to explore the optimizations described in [16] to further improve this implementation.

## 2.2 Encounter Mode

The following is the TL encounter mode transaction. For reasons we explain later, this mode assumes a type-stable closed memory pool or garbage-collection.

1. Run the transactional code, reading the locks of all fetched-from shared locations and building a local read-set and write-set (the write-set is an undo set of the values before the transactional writes).

Transactional stores to shared locations are handled by acquiring locks as they are encountered, saving the address and current value into the thread's local write-set, and pointing from the lock to the write-set entry. The shared variables are written with the new values during this step.

A transactional load checks to see if the lock is free or is held by the current transaction and if so reads the value from the location. There is thus no need to look-aside for the value in the write-set. If the transactional load operation finds that the lock is held it will use a bounded spin. During the operation of the transaction we periodically validate the read-set. If the read-set is found to be invalid we abort the transaction. This avoids the possibility of a doomed transaction (a transaction that has read inconsistent global state) from becoming trapped in an infinite loop.

2. Attempt to commit the transaction. Re-read the locks of all read-only locations to make sure version numbers haven't changed. If a version does not match, restore the values using the write-set, roll-back (release) the locks, abort the transaction, and retry.
3. The prior observed reads in step (1) have been validated as forming an atomic snapshot of memory. The transaction is now *committed*.
4. Release all the locks identified in the write-set by atomically incrementing the version and clearing the write-lock bit.

We note that the locks in encounter mode are held for a longer duration than in commit mode, which accounts for weaker performance under contention. However, one does not need to look-aside and search through the write-set for every read.

## 2.3 Contention Management

As described above TL admits live-lock failure. Consider where thread T1's read-set is A and its write-set is B. T2's read-set is B and write-set is A. T1 tries to commit and locks B. T2 tries to commit and acquires A. T1 validates A, in its read-set, and aborts as B is locked by T2. T2 validates B in its read-set and aborts as B was locked by T1. We have mutual abort with no progress. To provide liveness we use bounded spin and a back-off delay at abort-time, similar in spirit to that found in CSMA-CD MAC protocols. The delay interval is a function of (a) a random number generated at abort-time, (b) the length of the prior (aborted) write-set, and (c) the number of prior aborts by the current thread for this transactional attempt.

## 2.4 The Pathology of Transactional Memory Management

Concurrent mixed-mode transactional and non-transactional accesses to a given object or memory location are proscribed. In other words, when a particular object is being accessed with transactional load and store operations it must not be accessed with normal non-transactional load and store operations. Objects recycled in one mode should be made available for reuse in the other, an approach we call an *open memory system*. We will say that a memory recycling system that correctly implements such an open memory system is *safe*.

For type-safe garbage-collected managed runtime environments such as that of the Java programming language, any of the TL lock-mapping policies (PS, PO, or PW) and modes (Commit with a write-set or Encounter with an undo-set) are safe, as the GC assures that transactionally accessed memory will only be released once no references remain to the object. A specialized malloc/free mechanism that works safely, even when malloc and free are called from within transactions, was suggested in [31]. Here we are interested in STM mechanisms that will work with a system's normal malloc and free operations.

In C or C++ TL preferentially uses the PS/Commit locking scheme to allow the C programmer to use normal malloc() and free() operations to manage the lifecycle of structures containing transactionally accessed shared variables. Using PS was also suggested in [29, 15, 34]. In PS/Commit mode an object can exit the transactional domain and subsequently be accessed with normal non-transactional loads and stores, but we must wait for the object to *quiesce* before it leaves. For any given location in an object about to leave the transactional domain, there can be at most one transaction holding the transactional lock, and quiescing means waiting for that lock to be released, implying that all pending transactional stores to the location have been "drained",

before allowing the object to exit the transactional domain and subsequently to be accessed with normal load and store operations. Once it has quiesced, the memory can be freed and recycled in a normal fashion, because any transaction that may acquire the lock and reach the disconnected location will fail its read-set validation.

To motivate the need for quiescing, consider the following scenario with PS/Commit. We have a linked list of 3 nodes identified by addresses A, B and C. A node contains Key, Value and Next fields. The data structure implements a traditional key-value mapping. The key-value map (the linked list) is protected by TL using PS. Node A's Key field contains 1, its value field contains 1001 and its Next field refers to B. B's Key field contains 2, its Value field contains 1002 and its Next field refers to C. C's Key field contains 3, the value field 1003 and its Next field is NULL. Thread T1 calls `put(key=2, value=2002)`. The TL-based `put()` operator traverses the linked list using transactional loads and finds node B with a key value of 2. T1 then executes a transactional store into B.Value to change 1002 to 2002. T1's read-set consists of A.Key, A.Next, B.Key and the write-set consists of B.Value. T1 attempts to commit; it acquires the lock covering B.Value and then validates that the previously fetched read-set is consistent by checking the version numbers in the locks covering the read-set. Thread T1 stalls. Thread T2 executes `delete(key=2)`. The `delete()` operator traverses the linked list and attempts to splice-out Node B by setting A.Next to C. T2 successfully commits. T2 successfully commits, storing C into A.Next. T2's transaction completes. T2 then calls `free(B)`. T1 resumes in the midst of its commit and stores into B.Value. We have a classic modify-after-free pathology. To avoid such problems T2 calls `quiesce(B)` after the commit finishes but before freeing B. This allows T1's latent transactional Store to drain into B before B is freed and potentially reused. Note, however, that TL (using quiescing) did not admit any outcomes that were not already possible under a simple coarse-grained lock. Any thread that attempts to write into B will, at commit-time, acquire the lock covering B, validate A.Next and then store into B. Once B has been unlinked there can be at most one thread that has successfully committed and is in the process of writing into B. Other transactions attempting to write into B will fail read-set validation at commit-time as A.Next has changed.

Consider another following problematic lifecycle scenario based on the A,B,C linked list, above. Let's say we're using TL in the C language to moderate concurrent access to the list, but with either PO or PW mode where the lock word(s) are embedded in the node. Thread T1 calls `put(2, 2002)`. The TL-based `put()` method traverse the list and locates node B having a key value of 2. Thread T2 then calls `delete(2)`. The `delete()` operator commits successfully. T2 waits for B to quiesce and then calls `free(B)`. The

memory underlying B is recycled and used by some other thread T3. T1 attempts to commit by acquiring the lock covering B.Value. The versioned lock-word is collocated with B.Value, so the the CAS operation transiently changes the versioned lock-word contents. T2 then validates the read-set, recognizes that A.Next changed (because of T1's `delete()`) and aborts, restoring the original versioned lock-word value. T1 has cause the memory word underlying the lock for B.value to "flicker", however. Such modifications are unacceptable; we have a classic modify-after-free error.

Finally, consider the following pathological scenario admitted by PS/Encounter because of its use of an undo set. T1 calls `put(2,2002)`. `Put()` traverses the list and locates node B. T2 then calls `delete(2)`, commits successfully, calls `quiesce(B)` and `free(B)`. T1 acquires the lock covering B.Value, saves the original B.Value (1002) into its private write undo set, and then stores 2002 into B.Value. Later, during read-set validation at commit-time, T1 will discover that its read-set is invalid and abort, rolling back B.Value from 2002 to 1002. As above, this constitutes a modify-after-free pathology where B recycled, but B.Value transiently "flickered" from 1002 to 2002 to 1002. We can avoid this problem by acquiring locks in encounter order and validating after every lock acquisition or using a write-set instead of an undo set. The latter solution confers safety, but at the cost of dropping the main reason for encounter order locking, the use of an undo set.

As such, we advocate using PS/Commit for normal C code as the versioned lock-words (transactional metadata) are stored separately in type-stable memory distinct from the data they protect. This provision can be relaxed if the C-code uses some type of garbage-collection (such as Boehm-style [6] conservative garbage-collection for C, Michael-style hazard pointers [27, 18] or Fraser-style Epoch-Based Reclamation [11]) or type-stable storage for the nodes.

## 2.5 Mechanical Transformation of Sequential Code

As we discussed earlier, the algorithm we describe can be added to code in a mechanical fashion, that is, without understanding anything about how the code works or what the program itself does. In our benchmarks, we performed the transformation by hand. We do however believe that it may be feasible to automate this process and allow a compiler to perform the transformation given a few rather simple limitations on the code structure within a transaction. This should be an interesting topic for future research.

We note that hand-crafted data structures can always have an advantage over TL, since for example TL has no way of knowing that prior loads executed within a transaction might no longer have any bearing on results produced by transaction.

Consider the following scenario where we have a TL-protected hashtable. Thread T1 traverses a long hash bucket chain searching for the value associated with a certain key, iterating over “next” fields. We’ll say that T1 locates the appropriate node at or near the end of the linked list. T2 concurrently deletes an unrelated node earlier in the same linked list. T2 commits. At commit-time T1 will abort because the linked-list “next” field written to by T2 is in T1’s read-set. T1 must retry the lookup operation (ostensibly locating the same node). Given our domain-specific knowledge of the linked list we understand that the lookup and delete operations didn’t really conflict and could have been allowed to operate concurrently with no aborts. A clever optimistic hand-coded locking scheme would have the advantage of allowing this desired concurrency [17]. Nevertheless, as our empirical analysis later in the paper shows, in the data structure we tested, the beneficial effect of this added concurrency on overall application scalability does not seem to be as profound as one would think.

## 2.6 Software-Hardware Inter-Operability

Though we have described TL as a software based scheme, it can be made inter-operable with HTM systems. On a machine supporting dynamic hardware, transactions executed in hardware need only verify for each location that they read or write that the associated lock is free. There is no need for the hardware transaction to store an intermediate locked state into the lock word(s). For every write they also need to update the version number of the associated stripe lock upon completion. This suffices to provide inter-operability between hardware and software transactions. Any software read will detect concurrent modifications of locations by a hardware writes because the version number of the associated lock will have changed. Any hardware transaction will fail if a concurrent software transaction is holding the lock to write. Software transactions attempting to write will also fail in acquiring a lock on a location since lock acquisition is done using an atomic hardware synchronization operation (such as CAS or a single location transaction) which will fail if the version number of the location was modified by the hardware transaction.

## 3 An Empirical Evaluation of STM Performance

We present here the a comparison of algorithms representing state-of-the-art non-blocking [14] and lock-based [10] STMs on a set of microbenchmarks that include the now standard concurrent red-black tree structure [19], as well as concurrent skiplists [14] and a concurrent shared queue [28].

The red-black tree tested with transactional locking was derived from the `java.util.TreeMap` implementation found in the Java programming language JDK 6.0. That implementation was written by Doug Lea and Josh Bloch. In turn, parts of the Java `TreeMap` were derived from the Cormen et al. [7]. The skiplist was derived from Pugh [30]. We would have preferred to use the exact Fraser-Harris red-black tree but that code was written to their specific transactional and memory allocation interface and could not readily be converted to a simple form. We use large and small versions of the data structures, with 20,000 keys or 200 keys. We found little difference when we further increased the size of the trees a hundred-fold.

The skiplist and red-black tree implementations expose a key-value pair interface of *put*, *delete*, and *get* operations. The *put* operation installs a key-value pair. If the key is not present in the data structure *put* will insert a new element describing the key-value pair. If the key is already present in the data structure *put* will simply update the value associated with the existing key. The *get* operation queries the value for a given key, indicating if the key was present in the data structure or not. Finally, *delete* removes a key from the data structure, returning an indication if the key was found to be present in the data structure. The benchmark harness calls *put*, *get* and *delete* to operate on the underlying data structure. The harness allows for the proportion of *put*, *get* and *delete* operations to be varied by way of command line arguments, as well as the number of threads, trial duration, initial number of key-value pairs to be installed in the data structure, and the key-range. The key range describes the maximum possible size (capacity) of the data structure.

The harness spawns the specified number of threads. Each of the threads loops, and in each iteration the thread first computes a uniformly chosen random number used to select, in proportion to command line argument mentioned above, if the operation to be performed will be a *put*, *get* or *delete*. The thread then generates a uniformly selected random key within the key range, and, if the operation is a *put*, a random value. The thread then calls *put*, *get* or *delete* accordingly. All threads operate on a single shared data structure. At the end of the timing interval specified on the command line the harness reports the aggregate number of operations (iterations) completed by the set of threads.

For our experiments we used a 16-processor Sun Fire™ V890 which is a cache coherent multiprocessor with 1.35Ghz UltraSPARC-IV® processors running Solaris™ 10. In our benchmarks, our TL STMs used the Solaris *schedctl* mechanism to allow threads to request short-term preemption deferral by storing to a thread-specific location which is read by the kernel-level scheduler. Preemption deferral is advisory - the kernel will try to avoid preempting a thread that has requested deferral.

Our benchmarked algorithms included:

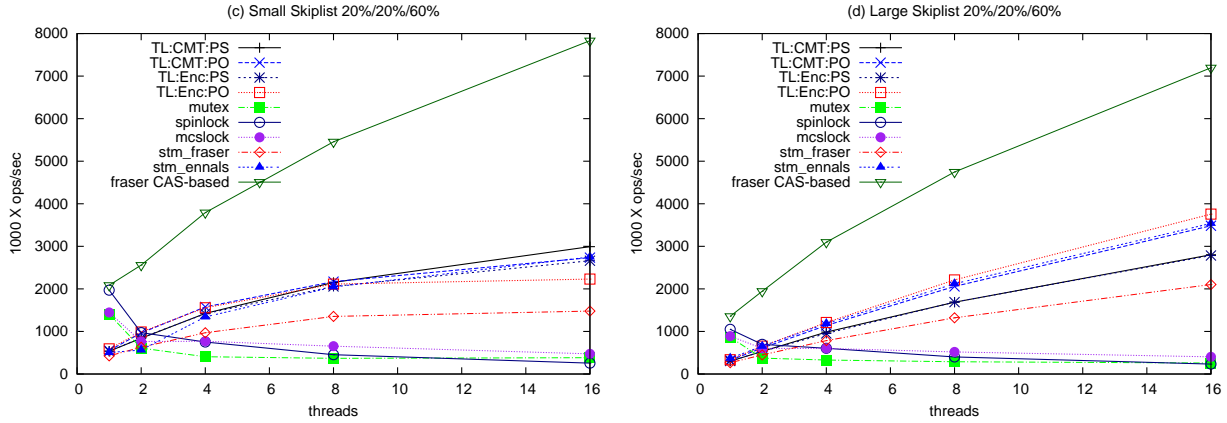


Figure 1. Throughput of Skip Lists with 20% puts, 20% deletes, and 60% gets

**Mutex, SpinLock, MCSLock** We implemented three variations of mutual exclusion locks. Mutex is a Solaris Pthreads mutex, Spinlock is a lock implemented with a CAS based Test-and-test-and set [22], and MCSLock is the queue lock of Mellor-Crummey and Scott [26].

**stm\_fraser** This is the state-of-the-art non-blocking STM of Harris and Fraser [14]. We use the name originally given to the program by its authors. It has a special record per object with a pointer to a transaction record. The transformation of sequential to transactional code is not mechanical: the programmer specifies when objects are transactionally opened and closed to improve performance.

**stm\_ennals** This is the lock-based encounter-order object-based STM algorithm of Ennals taken from [10] and provided in LibLTX [14]. Note that LibLTX includes the original Fraser and Harris lockfree-lib package. It uses a lock per object and a non-mechanical object-based interface of [14]. Though we did not have access to code for the Saha et al. algorithm [34], we believe the Ennals algorithm to be a good representative this class of algorithms, with the possible benefit that the Ennals structures were written using the non-mechanical object-based interface of [14] and because unlike Saha et al, Ennals’s write-set is not shared among threads.

**TL** Our new transactional locking algorithm. We use the notation TL/ENC/PO for example to denote a version of the algorithm that uses encounter mode lock acquisition, an undo-set, and per-object locking. We alternatively also use commit mode with a write-set (CMT) and per-stripe locking (PS). We will not present results

for encounter mode lock acquisition with a write-set, as we found its performance was always inferior to ENC with an undo-set.

**hanke** This is the hand-crafted lock-based concurrent relaxed red-black tree implementation of Hanke [13] as coded by Fraser [14] and referred to as “rb\_lock\_mutex” in the LibLTX library. The idea of relaxed balancing is to uncouple the re-balancing from the updating in order to speed up the update operations and to allow a high degree of concurrency. The algorithm also uses an understanding of the structures data relationships to allow traversals of the data structure ignore the fact that nodes are being modified while they are traversed.

**fraser CAS-Based** This is a lock-free skiplist due to Fraser [14] (A variant of this algorithm by Lea is included in the Java programming language JDK 6.0).

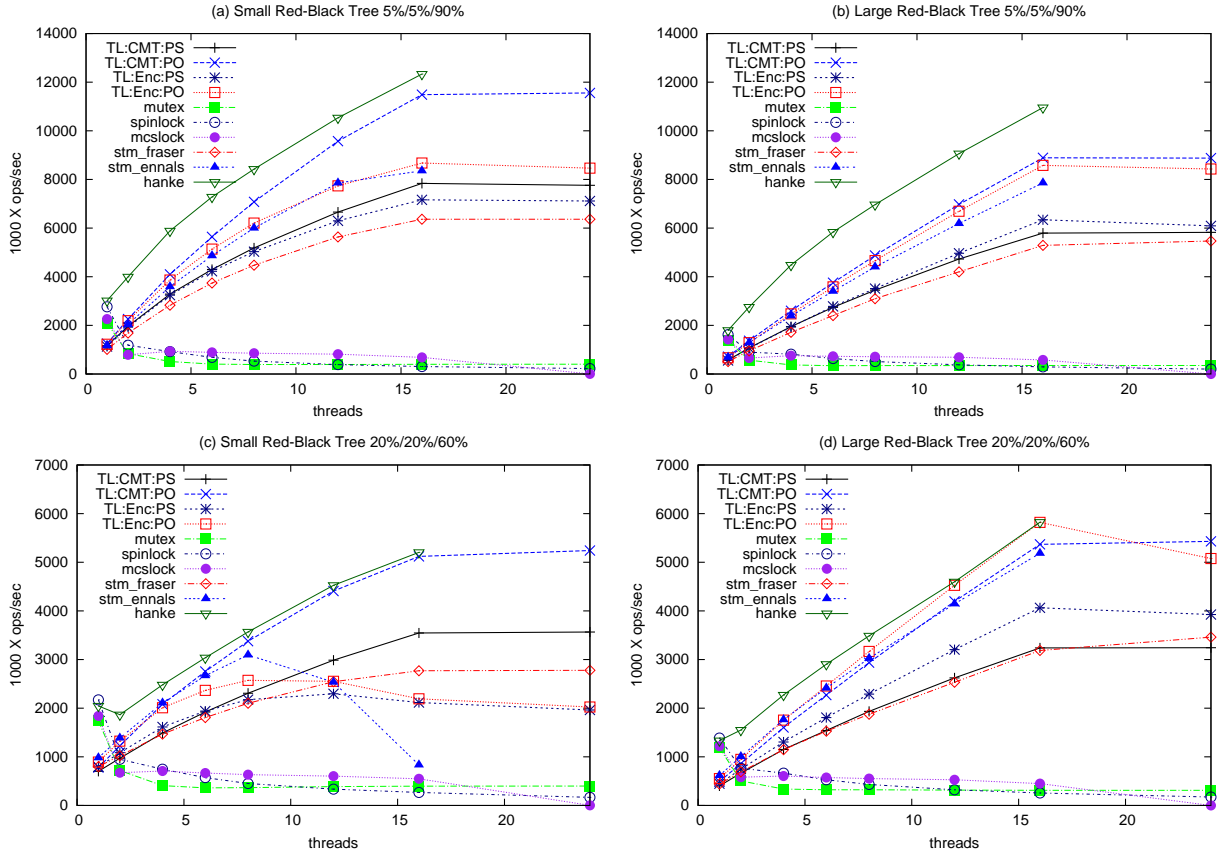
**MS2Lock, SimpleLock** Using the Mutex, Spinlock, and MCSLock locking algorithms to implement locks, we show three variants of Michael and Scott’s concurrent queue implemented [28] using two separate locks for the head and tail pointers, and three additional variants of a simple implementation using a single lock for both the head and tail.

We note that unfortunately we could not introduce the use of schedctl-based preemption deferral into the *hanke*, *stm\_ennals*, or *stm\_fraser*.

### 3.1 Locking vs Non-Blocking

In our first benchmark we tested a skiplist tree data structure in various configurations varying the fraction of modifying the fraction of puts, deletes, and get operations





**Figure 2. Throughput of Red-Black Tree with 5% puts and 5% deletes and 20% puts, 20% deletes**

(method calls). We only show the case of 20% puts, 20% deletes, and 60% gets because all other cases were very similar. As can be seen in Figure 1, Fraser’s hand-crafted lock-free CAS-based implementation has twice the throughput or more than the best STMs. Of the STM methods, the lock-based TL and Ennals STMs outperform all others. They are twice as fast as Fraser and Harris’s lock-free STM, and more than five times faster than coarse-grained locks. Though the single thread performance of STMs is inferior to that of locks, the crossover point is two threads, implying that with any concurrency, choose the STM. This benchmark indicates that improving both latency and single thread performance should be a goal of future STM design. The TL implementation with encounter-order and PO locks is the best performer on large data structures but is the first to deteriorate as the size of the structure decreases, increasing contention.

### 3.2 Encounter vs Commit and PO vs PS

In our second benchmark we tested a red-black tree data structure in various configurations considered to be com-

mon application usage patterns. As can be seen in Figure 2, the TL lock-based algorithm outperforms Ennals’s lock-based and Fraser’s non-blocking STMs. On large data structures under contention (Part D) it even outperforms Hanke’s hand-crafted implementation. Moreover, through the use of the *schedctl* mechanism, performance does not deteriorate when the number of threads exceeds the number of processes, lending credibility to our claim that lock based STMs can deliver good performance across the concurrency range. The performance of the Ennals and hanke algorithms is not plotted beyond 16 threads because we were unable to add *schedctl* use to them and their performance deteriorates when the number of threads increases beyond 16. It is interesting to notice that the performance of Fraser’s non-blocking STM does not deteriorate even though we were unable to add to it *schedctl* use.

There are several interesting points to notice about these graphs in the concurrency range below 16 threads.

- Overall the TL algorithm in commit (CMT) mode using PO locking does as well as the Ennals and TL encounter-order (ENC) algorithms.

- The performance of both the Ennals encounter-order algorithm deteriorates as the data structure becomes smaller (or as the number of modifying operations increases). Part C of Figure 2 shows that the encounter-order TL algorithm exhibits the same performance drop.
- If one looks at the high contention benchmark in Figure 3, where 80% of the operations modify the data structure and where 72% of all transactional references are loads, one can see that this behavior continues to the extreme. Under high contention, Ennals’s algorithm degrades to become worse than any of the locks, the TL in encounter-order and the lock-free Harris and Fraser STM stop scaling, the hand-crafted Hanke algorithm starts to flatten out, and the two commit mode TL STMs continue to scale. Our conclusion is that one should clearly not settle on encounter-order locking as the default, as suggested by Saha et al. [34], and pending investigation with larger set of benchmarks, it may well be that one could settle on always using commit-time lock acquisition.

- Perhaps surprisingly, abort rates seem to have little effect on overall performance. We present sample abort rate graphs in Figure 5 that correspond to the performance graphs in Parts C and D of Figure 2. As can be seen, PO does better than PS, a conclusion agrees with that of Saha et al. [34]. This is true even though, as seen in the large data structure abort rate graphs, PO introduces up to 50% more transaction failures than PS, yet the performance of PO is better. Moreover, as can be seen in small red-black trees in which the failure rates increase tenfold when compared to large ones, TL/CMT/PO and TL/ENC/PS have the same abort rates yet TL/CMT/PO has, for example, twice the performance if one looks at the graph in Part C of Figure 2. In general, the performance impact of abort rates seems to be much less than the impact conferred by the better locality of reference (accessing the lock and object together) provided by PO. In the extreme worst case PS will have twice the data-cache and data-TLB miss rates (worse data locality) as the compared to PO as the stripe array is separate. In addition the width of the data covered by PS stripes impacts performance. Wide stripes may result in fewer high-latency atomic CAS instructions when acquiring the locks that cover the write-set, but narrower stripes offer better potential parallelism. Finally, as we noted earlier, in languages like C and C++, one must use PS mode to allow interoperability with the normal memory lifecycle.

Our third benchmark in Figure 4 shows the performance of various locking and STM methods in implementing a

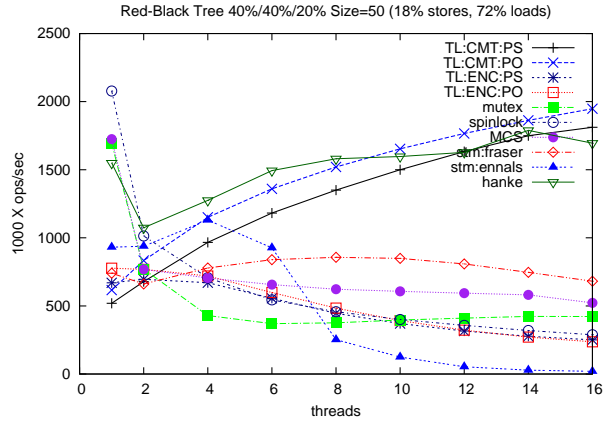


Figure 3. Throughput of Red-Black Tree under high contention

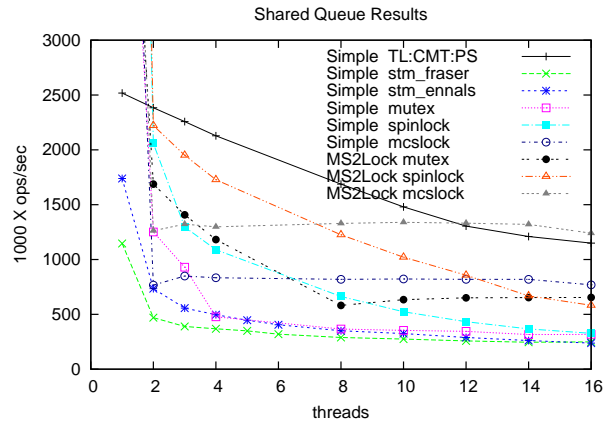
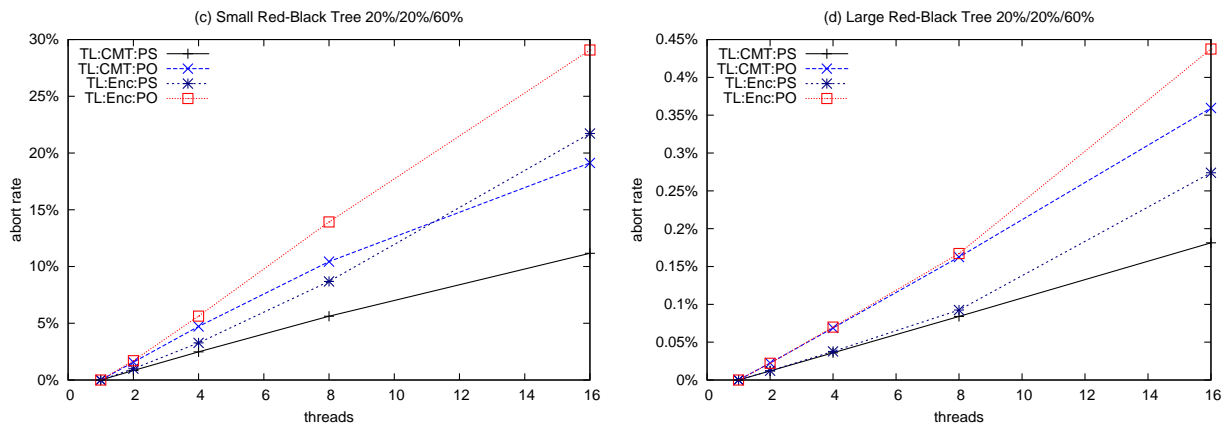


Figure 4. Throughput of Concurrent Queue

shared queue algorithm. A shared queue is a natural example of a small data structure with high levels of contention. As we show, a TL queue mechanically generated from sequential code delivers the same performance as the hand-crafted Michael and Scott two Lock algorithm (MS2Lock).

### 3.3 Summarizing the Comparison Among Approaches

Table 1 summarizes our comparison of the different methods of constructing lock-based STMs. There are three algorithmic elements being compared: encounter-order locking of written locations (ENC) versus commit-time locking (CMT), per stripe locking (PS) versus per object locking (PO), and validation of the read-set on every write (VOW) or only before committing (VBC). We com-



**Figure 5. Graphs of abort rates for TL/ENC versus TL/CMT on a Red-Black Tree matching the performance graphs in Parts C and D of Figure 2.**

	GC or Closed Pool	Malloc/Free	Small High Load	Small Low Load	Large High Load	Large Low Load
ENC/PS/VOW	Safe	Safe	Very Poor	Very Poor	Very Poor	Very Poor
ENC/PO/VOW	Safe	Unsafe	Very Poor	Very Poor	Very Poor	Very Poor
ENC/PS/VBC	Safe	Unsafe	Very Poor	Good	Good	Good
ENC/PO/VBC	Safe	Unsafe	Very Poor	Good	Best	Best
CMT/PS/VBC	Safe	Safe	Good	Good	Good	Good
CMT/PO/VBC	Safe	Unsafe	Best	Best	Best	Best

**Table 1. Comparison Table**

pare the different methods in terms of the compatibility with the memory lifecycle of garbage-collected languages like the Java programming language, or C programs that use a closed memory system, versus C programs that use only malloc and free style allocation. The table shows which techniques work safely only with GC or a closed system such as Fraser’s Epoch-based reclamation scheme. The discussion based on which these table entries were derived appears in Section 2.4. We rank performance using a scale which includes very poor, poor, good, better, and best for any given category of data structure and load, based on the benchmarks presented earlier in this section. We do not show entries for the combination of commit time locking (CMT) and validation on every write (VOW) since VBC is significantly less costly than VOW and it suffices for commit time locking.

We note that TL uses a versioned lock-word, but if we were to instead use a RW lock-word (with so-called visible readers) then all the VBC forms ( $\{\text{ENC, CMT}\} \times \{\text{PO, PS}\}$ ) will work safely with malloc and free. In addition, RW locks don’t admit so-called zombie transactions, transactions that have read inconsistent data and are eventually fated to abort, but are still running. We decided against RW locks early on in our algorithm design because they gen-

erate excessive cache coherency traffic on traditional SMP systems.

The following is a summary of the findings the table reveals.

- A quick glance at the table reveals that the performance of VOW schemes is very poor. We based this data on benchmarking we performed on Moir et al’s HyTM [29] which uses a mechanism similar to ENC/PS/VOW in order to allow programmers to freely use malloc and free. It is not clear to us at this point how to categorize the work of Saha et al. [34] who use, to the best of our understanding, ENC/PS/VBC. They make some assumptions on the runtime/memory system that keep it closed.
- As can be seen, it would seem that ENC locking is the best approach only on large objects using PO locking. However, ENC delivers very poor performance on small data structures. The CMT locking approach, on the other hand, delivers the best performance for most objects and concurrency levels. It would thus be the best choice for languages like the Java programming language or systems that have a closed memory system to use CMT/PO as provided by the TL algorithm.

- It would seem that the CMT/PS used in TL is the *only* scheme to deliver good performance for systems in which programmers wish to use malloc and free style allocation. ENC/PS/VOW is non-viable because of the overhead of the repeated validation. We note that the throughput of CMT/PS is not as good as CMT/PO (or ENC/PO on large unloaded structures) because of the extra cache traffic due to the separate lock locations, but is nevertheless reasonable.

### 3.4 Finer Analysis of Overhead

To better understand what the sources of the overhead in the TL design were, we looked at the single thread performance of our TL algorithm. We note that HTMs attempt to cut down the costs of both reads and writes. We wanted to find out what the benefit of using an HTM transaction to acquire all write locks at commit-time might be. We conducted a simple benchmark in which the TL algorithm ran on a red-black tree of size 50 with 40% put, 40% delete, and 20% get operations in single threaded mode, replacing all expensive CAS-based lock acquisitions with simple reads and writes. We found that in our benchmark with a 1:4 ratio of transactional reads to writes, the number of operations per second with CAS was 5.2 million and if we converted CAS to non-atomic reads and writes it yielded 5.8 million operations per second, an improvement of .6 million, or about 10%. Even here it turned out that speeding up lock acquisition is simply not worth it.

We then asked ourselves if eliminating the construction of a read-set might have a significant effect. We again ran red-black tree benchmark but did not construct a read-set and made only one pass through the transactional code, as would be done by a transaction that had hardware support for determining if the read set was consistent. Our transactional loads still had to look-aside into the write-set. The transactional load operation fetched the versioned lock-word and then the data. The result was an increase of the total number of completed operations to 8.2 million per second.

## 4 Conclusion

We presented an evaluation of the factors affecting the performance of STM algorithms. Perhaps surprisingly, we found that the determining performance factors were the “fixed” costs/overheads associated with STM mechanisms (such as read-set validation), and not factors associated with scalability (such as transaction abort rates). This led us to the design of the *transactional locking* (TL) algorithm, which tries to minimize these costs.

In the performance section we showed preliminary evidence that seems to suggest that providing only read-set

validation support, as opposed to full blown HTM, may deliver a significant performance benefit. We note that typical read/write ratios in real-world applications tend to follow the 80/20 rule, that is, chances are good that transacted methods will have large read sets and much smaller write sets. The idea then, is to decouple the transactional read-set and the write-set, and to focus only on reducing the cost of constructing a read-set and validating it by providing support in hardware, leaving the write set implementation in software [8]. We leave this as a subject for further research.

The code for our TL algorithms and some of our benchmarks as well as the code for next generation TL2 algorithm [9] can be found at <http://research.sun.com/scalable>.

## References

- [1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, 1993.
- [2] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. *ACM SIGPLAN Notices*, 34(10):207–222, 1999.
- [3] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.
- [4] C. S. Ananian and M. Rinard. Efficient software transactions for object-oriented languages. In *Proceedings of Synchronization and Concurrency in Object-Oriented Languages (SCOOOL)*. ACM, 2005.
- [5] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.
- [6] H.-J. Boehm. Space efficient conservative garbage collection. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 197–206, 1993.
- [7] T. H. Cormen, E. Leiserson, Charles, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990. COR th 01:1 1.Ex.
- [8] D. Dice. [http://blogs.sun.com/dave/entry/hardware\\_assisted\\_transactional\\_read\\_set](http://blogs.sun.com/dave/entry/hardware_assisted_transactional_read_set).
- [9] D. Dice, O. Shalev, , and N. Shavit. Transactional locking ii. In *Proc. International Symposium on Distributed Computing*. Springer Verlag, 2006.
- [10] R. Ennals. Software transactional memory should not be obstruction-free. [www.cambridge.intel-research.net/\(tilda\)rennals/notlockfree.pdf](http://www.cambridge.intel-research.net/(tilda)rennals/notlockfree.pdf). [www.cambridge.intel-research.net/rennals/notlockfree.pdf](http://www.cambridge.intel-research.net/rennals/notlockfree.pdf).
- [11] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.
- [12] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the*

- 31st annual international symposium on Computer architecture, page 102, Washington, DC, USA, 2004. IEEE Computer Society.
- [13] S. Hanke. The performance of concurrent red-black tree algorithms. In *WAE '99: Proceedings of the 3rd International Workshop on Algorithm Engineering*, pages 286–300, London, UK, 1999. Springer-Verlag.
- [14] T. Harris and K. Fraser. Concurrent programming without locks.
- [15] T. Harris and K. Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.
- [16] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25, New York, NY, USA, 2006. ACM Press.
- [17] S. Heller, M. Herlihy, V. Luchangco, M. Moir, N. Shavit, and I. W. N. Scherer. A lazy concurrent list-based set algorithm. In *9th Intl. Conf. on Principles of Distributed Systems (OPDIS)*, 2005.
- [18] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting lock-free dynamic-sized data structures. In *Proceedings of the 16th International Symposium on Distributed Computing*, volume 2508, pages 339–353. Springer-Verlag Heidelberg, January 2002. A improved version of this paper is in preparation for journal submission; please contact authors.
- [19] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM Press, 2003.
- [20] M. Herlihy and E. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.
- [21] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
- [22] C. Kruskal, L. Rudolph, and M. Snir. Efficient synchronization of multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, 1988.
- [23] S. Kumar, M. Chu, C. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *To appear in PPOPP 2006*, 2006.
- [24] V. J. Marathe, W. N. Scherer, and M. L. Scott. Design trade-offs in modern software transactional memory systems. In *In Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR'04)*, 2004.
- [25] V. J. Marathe, W. N. Scherer, and M. L. Scott. Adaptive software transactional memory. In *To Appear in the Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, 2005.
- [26] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, 1991.
- [27] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.
- [28] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Symposium on Principles of Distributed Computing*, pages 267–275, 1996.
- [29] M. Moir, P. Damron, A. Fedorova, Y. Lev, and V. Luchangco. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM Press.
- [30] W. Pugh. A skip list cookbook. Technical report, College Park, MD, USA, 1990.
- [31] A. R. R. L. Hudson B. Saha and B. Hertzberg. Mcrt-malloc: A scalable transactional memory allocator. In *In Proceedings of the 2006 International Symposium on Memory Management*, pages 197–206, June 2006.
- [32] R. Rajwar. Transactional memory online, <http://www.cs.wisc.edu/trans-memory>.
- [33] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 494–505, Washington, DC, USA, 2005. IEEE Computer Society.
- [34] B. Saha, A. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. A high performance software transactional memory system for a multi-core runtime. In *To appear in PPOPP 2006*, 2006.
- [35] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.
- [36] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 519–542. Springer-Verlag, 2004.