# What Really Makes Transactions Faster?

Dave Dice
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803-0903

dice@sun.com

Nir Shavit
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803-0903

shanir@sun.com

## 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 reexamines 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.

The results of our evaluation led us to the design of a *transactional locking* (TL) algorithm which we believe to be the simplest, most flexible, and best performing STM/HyTM to date. It combines seamlessly with hardware transactions and with any system's memory life-cycle, making it an ideal candidate for multi-language deployment today, long before hardware transactional support becomes commonly available.

Most important of all however were the results we derived from a comprehensive comparison of the performance of non-blocking, lock-based, and Hybrid STM algorithms versus fine-grained hand-crafted ones. Contrary to our intuitions, concurrent code generated in a mechanical fashion using our TL algorithm and several other STMs, scaled better than the hand-crafted fine-grained lock-based and lock-free data structures, even though their throughput was lower. We found that it was the lower latency of the hand-crafted data structures that made them faster than STMs, and not better contention management or optimizations based on the programmer's understanding of the particulars of the structure. This holds great promise for future mechanical generation of concurrent code using hardware transactional support.

## 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, course 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 "mechanical" methods for transforming sequential code or course-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 specialized hardware support as it becomes available.

### 1.1 Transactional Programming

The *transactional memory* programming paradigm [19] 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, 11, 19, 30], purely software based ones, i.e. software transactional memories (STM) [9, 13, 16, 18, 22, 23, 27, 31, 32, 33, 34], and hybrid schemes (HyTM) that combine hardware and software [4, 21, 27].[1]

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

---

[1] A broad survey of prior art can be found in [13, 22, 29].

introduce large degrees of complexity into the design [19, 30, 3, 11]. Providing them efficiently in software is a difficult task, and there seem to be numerous design parameters and approaches in the literature [9, 13, 16, 18, 23, 27, 31, 32], as well as requirements to combine well with hardware transactions once those become available [4, 21, 27].

## 1.2 Software Transactional Memory

The first STM design by Shavit and Touitou [33] 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 [18] 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 [17] STM, but involved two levels of indirection in accessing data, and had a costly Java^TM-based implementation. This Java-based implementation was improved on later by the ASTM of Marathe et al [23]. The OSTM of Fraser and Harris took a slightly different programming approach than DSTM, allowing 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 [13]. A key element of being non-blocking is the maintenance of publicly shared transaction records with undo or copy-back information. This tends to make the structures more susceptible to cache behavior, hurting overall performance. As our empirical data will show however, OSTM performs reasonably well across the concurrency range.

A recent paper by Ennals [9] suggested that on modern operating systems, deadlock avoidance is the only compelling reason for making transactions non-blocking, and that there is no reason to provide it for transactions at the user level. We second this claim, noting that mechanisms already exist whereby threads might yield their quanta to other threads and that Solaris' *schedctl* allows threads to transiently defer preemption while holding locks. Ennals [9] 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*, it is typically used in conjunction with an undo set [31]). 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 back before releasing the locks on the locations being written. The use of locks eliminates the need for indirection and shared transaction records as in the non-blocking STMs, it still requires however a closed memory system. 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 [31], concurrent and independent of our own work, uses a version of the Ennals's lock-based algorithm within a runtime system. It uses encounter order, but also keeps shared undo sets to allow transactions to actively abort others.

Moir [27] has suggested that the pointers to transaction records in non-blocking transactions can be used to coordinate hardware and software transactions to form hybrid transactional schemes. His HybridTM scheme has an implementation that acquires locks in encounter order.

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, most flexible, and best performing STM/HyTM to date.

## 1.3 Our Findings

The following are some of the results and conclusions presented in this paper:

- Ennals [9] suggested to build deadlock-free lock-based STMs rather than non-blocking ones [13, 27]. Our empirical findings support Ennals's claims: non-blocking transactions [13, 27] were less efficient than our TL lock-based ones on a variety of data structures and across concurrency ranges, even when they used a more complex yet advantageous non-mechanical programming interface [13]. Given that, as we show, locks provide a simple interface to hardware transactions, we recommend that the design of HyTMs shift from non-blocking to lock-based algorithms.

- Both Ennals and Saha et al [9, 31] have transactions acquire write locks as they encounter them (an "undo-set" algorithm). Saha et al [31] 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 degradation is 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 [31] 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, unlike encounter order locking which seems to require type-stable memory or specialized malloc/free implementations, 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.

- Of all the algorithms we tested, lock-free, or lock-based, the TL algorithm which acquires locks at commit time, is the only one that exhibits scalability across all 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 perfor-

mance standpoint, one should always default to using commit time locking.

- Both Ennals and Saha et al [9, 31] 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, we show that concurrent code generated mechanically using our new TL algorithm has scalability curves that are superior to those of all fine-grained hand-crafted data structures even when varying size and contention level. This implies that contrary to our belief, it is the overhead of the STM implementations (measured, for example, by single thread performance cost) that limits their 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.

In summary, TL's superior performance together with the fact that it combines seamlessly with hardware transactions and with any system's memory life-cycle, make it an ideal candidate for multi-language deployment today, long before hardware transactional support becomes commonly available.

## 2. TRANSACTIONAL LOCKING

The transactional locking approach is thus that rather than trying to improve on hand-crafted lock-based implementations by being non-blocking, we try and 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 [9] and Saha et al [31]. The availability of both modes will allow us to show the performance differences between them.

We associate a special versioned-write-lock with every transacted memory location. A *versioned-write-lock* is a simple single-word spinlock that uses a *compare-and-swap* (CAS)

operation to acquire the lock and a *store* to release it. Since one only needs a single bit to indicate that the lock is taken, we use the rest of the lock word to hold a version number. This number is incremented by every successful lock-release. In encounter mode the version number is displaced and a pointer into a threads private *undo log* is installed.

We allocate a collection of *versioned-write-locks*. 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 put 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 Java$^{TM}$ objects [2, 8]. 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 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, 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 (use a *safe* load operation to avoid de-referencing invalid pointers as a result of reading an inconsistent view of memory).

   A transactional load first checks (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 provides the illusion of processor consistency 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.

Transactional stores to shared locations are handled by saving the address and value into the thread's local write-set. The shared variables are not modified during this step. That is, transactional stores are deferred and contingent upon successfully completing the transaction. 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. 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.

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.

## 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 the 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 value 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 for the value in the write-set. If the transactional load operation finds that the lock is held it will 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. Acquire the locks associated with the write-set in any convenient order, using bounded spinning to avoid deadlock.

3. 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.

4. The prior observed reads in step (1) have been validated as forming an atomic snapshot of memory. The transaction is now *committed*.

5. 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 a 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

For type-safe garbage collected managed runtime environments such as Java any of the TL lock-mapping policies (PS, PO, or PW) and modes (Commit or Encounter) are safe, as the GC assures that transactionally accessed memory will only be released once no references remain to the object. 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 [31].

Concurrent mixed-mode transactional and non-transactional accesses are proscribed. 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. (When any accesses to an object are transactional, *all* accesses must be transactional). 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. 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(2, 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 converging the read-set. Thread T1 stalls. Thread T2 executes delete(2). The delete() operator traverses the linked list and attempts to splice-out Node B by setting A.Next to C. T2 successfully commits. The commit operator stores 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 free()ing B. This allows T1's latent transactional ST to drain into B before B is free()ed 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. Lets 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 lock-word is collocated with B.Value, so the the CAS operation transiently change the lock-word contents. T2 then validates the read-set, recognizes that A.Next changed (because of T1's delete()) and aborts, restoring the original 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. 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 log, 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 enhancing the encounter protocol to validate the read-set after each lock acquistion but before storing into the shared variable. This confers safety, but at the cost of additional performance.

As such, we advocate using PS/Commit for normal C code as the lock-words (metadata) are stored separately in type-stable memory distinct from the data protected by the locks. 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 [25] or Fraser-stye Epoch-Based Reclamation [10]) 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.

We note that hand-crafted data structures can always have an advantage over TL, as 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 a 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 "hand over hand" hand-coded locking scheme would have the advantage of allowing this desired concurrency. 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 versioned-write-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 [13], lock-based [9] STMs on a set of microbenchmarks that include the now standard concurrent red-black tree structure [18], as well as concurrent skiplists [13]and a concurrent shared queue [26].

The red-black tree tested with transactional locking was derived from the `java.util.TreeMap` implementation found in the Java 6.0 JDK. 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 [28]. We would have preferred to use the exact Fraser-Harris red-black tree but that code was written to to their specific transactional 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, returning an indication if the key was present in the data structure. 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$^{TM}$ V890 which is a cache coherent multiprocessor with 1.35Ghz UltraSPARC-IV® processors running Solaris$^{TM}$ 10.

Our benchmarked algorithms included:

**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 [20], and MCSLock is the queue lock of Mellor-Crummey and Scott [24].

**stm_fraser** This is the state-of-the-art non-blocking STM of Harris and Fraser [13]. 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 [9] and provided in LibLTX [13]. 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 [13]. Though we did not have access to code for the Saha et al algorithm [31], 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 [13] 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 and per-object locking. We alternately also use commit mode (CMT) or per-stripe locking (PS).

**hanke** This is the hand-crafted lock-based concurrent relaxed red-black tree implementation of Hanke [12] as coded by Fraser [13]. 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.
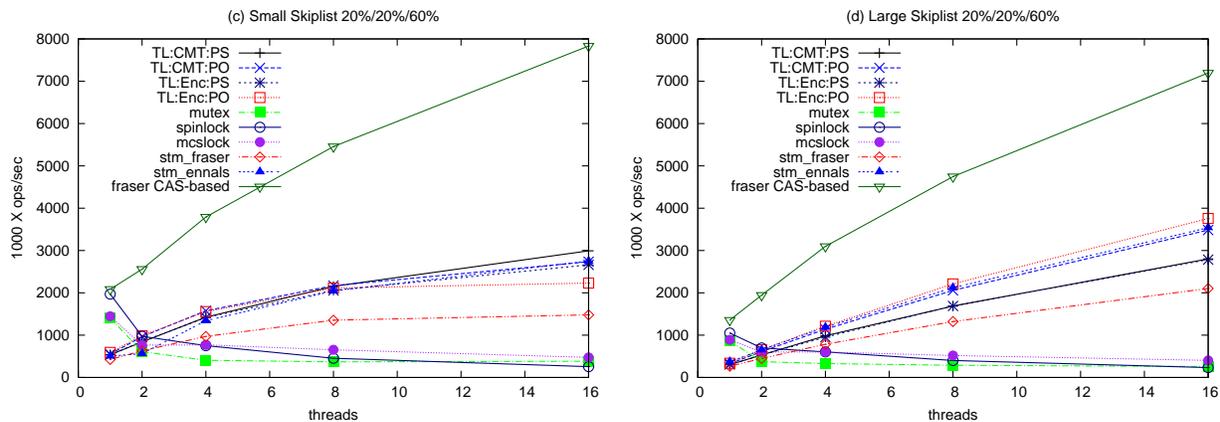
**Figure 1: Throughput of Skip Lists with 20% puts, 20% deletes, and 60% gets**

**fraser CAS-Based** This is a lock-free skiplist due to Fraser [13] (A Java variant of this algorithm by Lea is included in JDK 1.6).

**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 [26] 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.

## 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 (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 is 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 course 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 common 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.

There are several interesting points to notice about these graphs.

- 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 this is not a fluke. 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 continues to the extreme. Under high contention, Ennals's algorithm degrades to become worst 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. The scalability of the two commit mode TL algorithms gets further support if one looks at the normalized throughput graphs of Figure 5. It is quite clear that commit mode TL STMs are the only ones that show overall scalability. Our conclusion is that one should clearly not settle on encounter order locking as the default as suggested by Saha et al [31], 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 scalability and performance. We present sample abort rate graphs in Figure 5 that correspond to the normalized scalability graphs above them. As can be seen PO does better than PS, a conclusion agrees with that of Saha et al [31]. 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 scalability 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 twice the
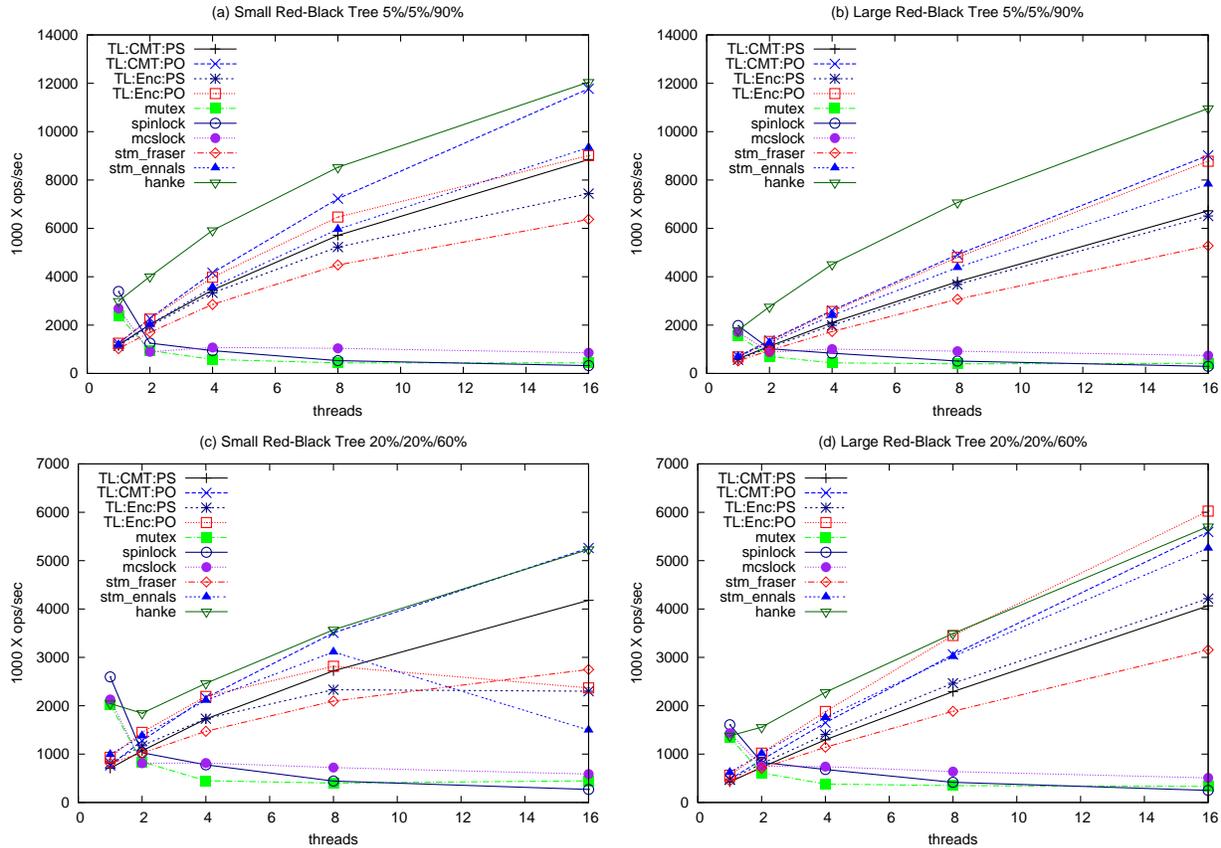
Figure 2: **Throughput of Red-Black Tree with 5% puts and 5% deletes and 20% puts, 20% deletes**

scalability of TL/ENC/PS and twice the performance if one looks at the graph in Part C of Figure 2. In general, Abort rates seem to be shadowed by the better locality of reference (accessing the lock and object together) provided by PO. Unfortunately, as we noted earlier, in languages like C and C++ one must use PS mode to allow interoperability with the normal malloc-free memory lifecycle.

Our third benchmark in Figure 4 shows the performance of various locking and STM methods in implementing a 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 What Makes Transactions Faster?

The graphs in Figure 5 possibly contain our most telling data. These are graphs that depict the scalability of the various methods by recasting the data we presented earlier in Figures 2 and 1 at 20%/20%/60%, normalizing the graphs based on the single thread performance. Contrary to all of our conjectures, the STMs, and in particular TL using commit order, have the best overall scalability, outperforming the hand-crafted red-black tree structures (results for skiplists were similar). As can be seen, this scalability is supported by the fact that the overall abort rates for TL
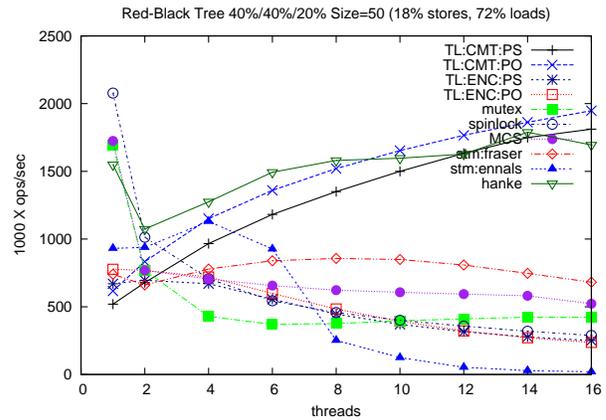


Figure 3: **Throughput of Red-Black Tree under high contention**

are low. This is rather surprising, since we thought the great advantage of hand-crafted data structures, as opposed to mechanically generated STM code, was the programmers ability to control contention based on his knowledge of the data flow relationships. For example, both the Hanke lock-based red-black tree and the Fraser lock-free skiplist, allow traversals to ignore ongoing modifications to the data struc-
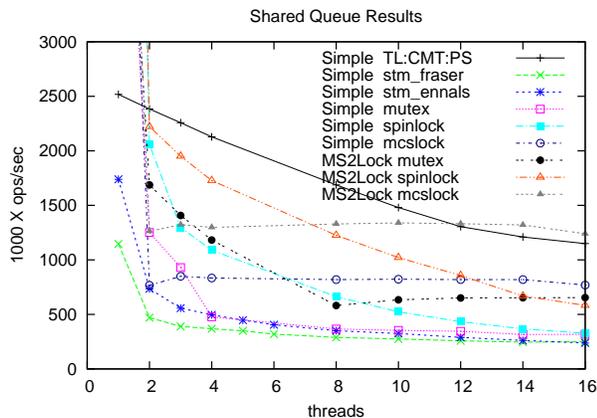
Figure 4: Throughput of Concurrent Queue

ture. However, as seen in Figure 5, and as we found out in similar benchmarks with 95% get operations (not presented here), TL, as well as other STMs, scaled better than these structures.

A couple of interesting data points we found were that our TL algorithm in commit mode scaled, for example, three times more than Hanke's algorithm at 16 processors, and yet both algorithms had the same throughput. On the red-black tree, TL commit mode scaled well both in PO and PS mode.

In conclusion, it is really the relative overheads, as can be seen from the single thread performance numbers in Figures 2 and 1, that determine which algorithm will perform better on a given benchmark. Our TL algorithm in commit mode is in fact algorithmically very similar to suggested hardware transaction schemes, implying that hardware transactions "in general" will fail in the same cases that software ones fail. Given that hardware transactions will lower the overheads of transactional execution, this holds great hope that HTM-based mechanically transformed sequential code can be as fast, or even faster, than hand-crafted data structures.

## 3.4 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 compare the different methods in terms of the compatibility with the memory life-cycle of garbage collected languages like Java, or C programs that use a closed memory pool, versus C programs that use only malloc and free style allocation. The table shows which techniques work safely only with GC or a closed pool 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 write-lock, but if we were to instead use a RW lock (with so-called visible readers) then all the VBC forms ({ENC,CMT} x {PO,PS}) will work safely with malloc and free. In addition, RW locks don't admit so-called zombie transactions, ones that may dereference invalid pointers or enter infinite loops because they read an inconsistent state. We decided against RW locks early on in our algorithm design because they generate 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's HyTM [27] 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 [31] 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 best-of-breed performance for all objects and all concurrency levels, and even on large uncontended objects when ENC/PO delivers better throughput than CMT/PO. It would thus be the best choice for languages like Java 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 she 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 reasonable.

## 3.5 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
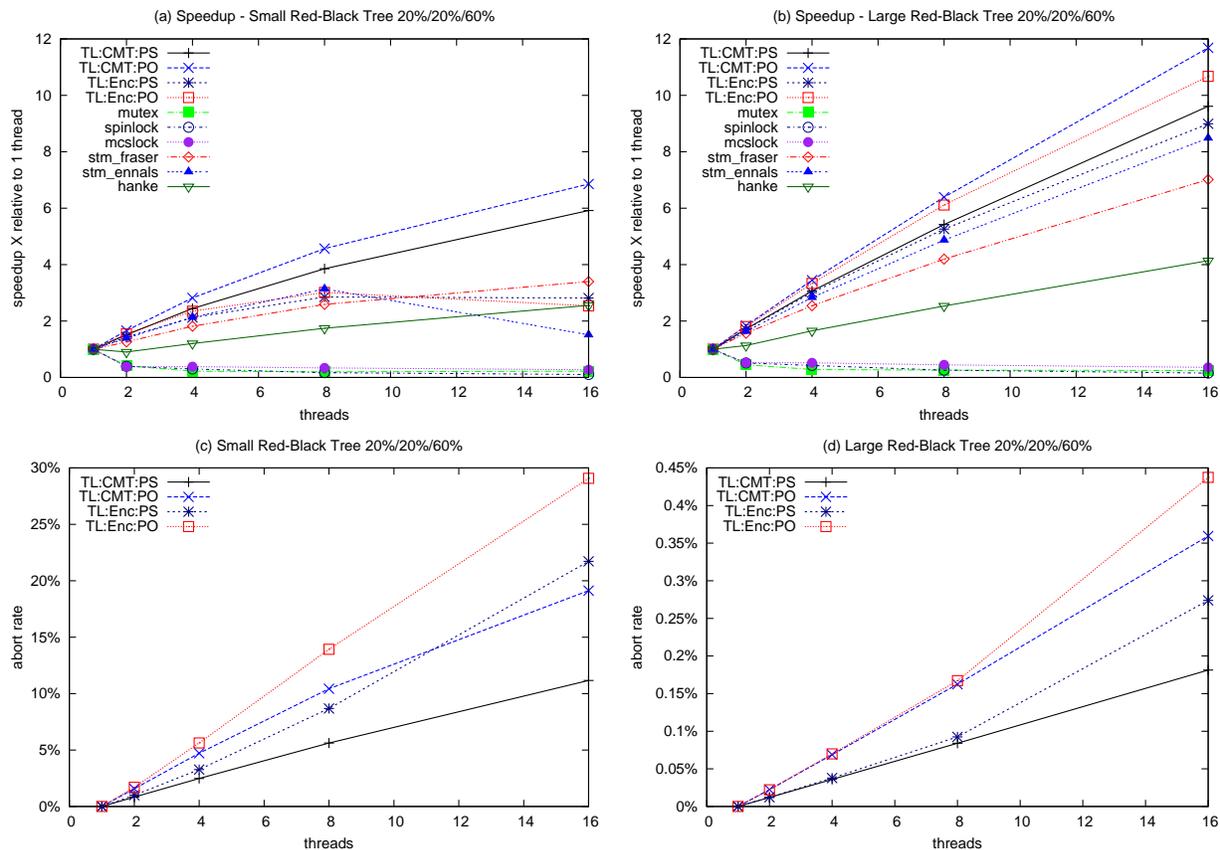
**Figure 5: Normalized throughput graphs of Red-Black Tree and below them the corresponding abort rates for TL/ENC versus TL/CMT. As can be seen, the dominant scalability factor is locality of reference (PO versus PS) and not the abort rate.**

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 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.

## 5. ACKNOWLEDGMENTS

## 6. REFERENCES

[1] AFEK, Y., ATTIYA, H., DOLEV, D., GAFNI, E., MERRITT, M., AND SHAVIT, N. Atomic snapshots of shared memory. *J. ACM 40*, 4 (1993), 873–890.

[2] AGESEN, O., DETLEFS, D., GARTHWAITE, A., KNIPPEL, R., RAMAKRISHNA, Y. S., AND WHITE, D. An efficient meta-lock for implementing ubiquitous synchronization. *ACM SIGPLAN Notices 34*, 10 (1999), 207–222.

[3] ANANIAN, C. S., ASANOVIC, K., KUSZMAUL, B. C., LEISERSON, C. E., AND LIE, S. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 316–327.

[4] ANANIAN, C. S., AND RINARD, M. Efficient software transactions for object-oriented languages. In *Proceedings of Synchronization and Concurrency in Object-Oriented Languages (SCOOL)* (2005), ACM.

[5] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (1970), 422–426.

[6] BOEHM, H.-J. Space efficient conservative garbage collection. In *SIGPLAN Conference on Programming Language Design and Implementation* (1993), pp. 197–206.

| | 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**

[7] Cormen, T. H., Leiserson, Charles, E., and Rivest, R. L. *Introduction to Algorithms.* MIT Press, 1990. COR th 01:1 1.Ex.

[8] Dice, D. Implementing fast java monitors with relaxed-locks. In *Java Virtual Machine Research and Technology Symposium* (2001), USENIX, pp. 79–90.

[9] Ennals, R. Software transactional memory should not be obstruction-free. www.cambridge.intel-research.net/~rennals/notlockfree.pdf. www.cambridge.intel-research.net/ rennals/notlockfree.pdf.

[10] Fraser, K. *Practical lock freedom.* PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[11] Hammond, L., Wong, V., Chen, M., Carlstrom, B. D., Davis, J. D., Hertzberg, B., Prabhu, M. K., Wijaya, H., Kozyrakis, C., and Olukotun, K. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture* (Washington, DC, USA, 2004), IEEE Computer Society, p. 102.

[12] Hanke, S. The performance of concurrent red-black tree algorithms. In *WAE '99: Proceedings of the 3rd International Workshop on Algorithm Engineering* (London, UK, 1999), Springer-Verlag, pp. 286–300.

[13] Harris, T., and Fraser, K. Concurrent programming without locks.

[14] Harris, T., and Fraser, K. Language support for lightweight transactions. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications* (2003), ACM Press, pp. 388–402.

[15] Harris, T., and Fraser, K. Language support for lightweight transactions. *SIGPLAN Not. 38*, 11 (2003), 388–402.

[16] Herlihy, M. The SXM software package, http://www.cs.brown.edu/~ mph/sxm/readme.doc.

[17] Herlihy, M., Luchangco, V., and Moir, M. Obstruction-free software transactional memory for supporting dynamic data structures. Tech. Rep. Technical Report, Sun Microsystems, May 2002.

[18] Herlihy, M., Luchangco, V., Moir, M., and Scherer, III, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing* (2003), ACM Press, pp. 92–101.

[19] Herlihy, M., and Moss, E. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture* (1993).

[20] Kruskal, C., Rudolph, L., and Snir, M. Efficient synchronization of multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems 10*, 4 (1988), 579–601.

[21] Kumar, S., Chu, M., Hughes, C., Kundu, P., and Nguyen, A. Hybrid transactional memory. In *To appear in PPoPP 2006* (2006).

[22] Marathe, V. J., Scherer, W. N., and Scott, M. L. Design tradeoffs 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).

[23] Marathe, V. J., Scherer, W. N., and Scott, M. L. Adaptive software transactional memory. In *To Appear in the Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)* (2005).

[24] Mellor-Crummey, J., and Scott, M. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (1991), 21–65.

[25] Michael, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst. 15*, 6 (2004), 491–504.

[26] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Symposium on Principles of Distributed Computing* (1996), pp. 267–275.

[27] Moir, M. HybridTM: Integrating hardware and software transactional memory. Tech. Rep. Archivist 2004-0661, Sun Microsystems Research, August 2004.

[28] Pugh, W. A skip list cookbook. Tech. rep., College Park, MD, USA, 1990.

[29] Rajwar, R. Transactional memory online, http://www.cs.wisc.edu/trans-memory.

[30] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 494–505.

[31] Saha, B., Adl-Tabatabai, A.-R., Hudson, R. L., Minh, C. C., and Hertzberg, B. A high performance software transactional memory system for a multi-core runtime. In *To appear in PPoPP 2006* (2006).

[32] Shalev, O., and Shavit, N. Predictive log-synchronization. In *EuroSys 2006, to appear* (2006).

[33] Shavit, N., and Touitou, D. Software transactional memory. *Distributed Computing 10*, 2 (February 1997), 99–116.

[34] Welc, A., Jagannathan, S., and Hosking, A. L. Transactional monitors for concurrent objects. In *Proceedings of the European Conference on Object-Oriented Programming* (2004), vol. 3086 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 519–542.