

McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime

Bratin Saha*, Ali-Reza Adl-Tabatabai*, Richard L. Hudson*, Chi Cao Minh**, Benjamin Hertzberg**

*Programming System Lab

Microprocessor Technology Lab

Intel Corporation

{Bratin.Saha, Ali-Reza.Ad-Tabatabai, Rick.Hudson}
@Intel.com

**Computer Architecture Lab

Stanford University

Palo Alto California

{caominh, elektrik}@stanford.edu

ABSTRACT

Applications need to become more concurrent to take advantage of the increased computational power provided by chip level multiprocessing. Programmers have traditionally managed this concurrency using locks (mutex based synchronization). Unfortunately, lock based synchronization often leads to deadlocks, makes fine-grained synchronization difficult, hinders composition of atomic primitives, and provides no support for error recovery. Transactions avoid many of these problems, and therefore, promise to ease concurrent programming.

We describe a software transactional memory (STM) system that is part of McRT, an experimental Multi-Core RunTime. The McRT-STM implementation uses a number of novel algorithms, and supports advanced features such as nested transactions with partial aborts, conditional signaling within a transaction, and object based conflict detection for C/C++ applications. The McRT-STM exports interfaces that can be used from C/C++ programs directly or as a target for compilers translating higher level linguistic constructs.

We present a detailed performance analysis of various STM design tradeoffs such as pessimistic versus optimistic concurrency, undo logging versus write buffering, and cache line based versus object based conflict detection. We also show a MCAS implementation that works on arbitrary values, coexists with the STM, and can be used as a more efficient form of transactional memory. To provide a baseline we compare the performance of the STM with that of fine-grained and coarse-grained locking using a number of concurrent data structures on a 16-processor SMP system. We also show our STM performance on a non-synthetic workload – the Linux sendmail application.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features – *Concurrent programming structures, Frameworks.*

General Terms Algorithms, Performance, Languages.

Keywords software transactional memory, atomic constructs, runtime environment, two-phase locking and read-versioning.

1. Introduction

The advent of multi-core processors has brought concurrency into mainstream applications. Programmers have traditionally used

locks to enforce mutual exclusion in concurrent applications. This requires the programmer to set up an association between a lock and the data (more abstractly a set of shared resources) that it protects, and implement a consistent locking protocol throughout the application. Lock-based synchronization can lead to deadlock, makes fine-grained synchronization error-prone, precludes composition of atomic primitives, and provides no support for error recovery. Transactional programming addresses these problems and provides an alternative synchronization mechanism [9][14][16]. With transactions, the programmer marks the regions or operations that should execute atomically; the compiler and runtime system take care of the implementation. There are several proposals related to linguistic constructs for supporting transactional memory [1][4][9]. This paper concentrates on the underlying runtime primitives and the interface needed to support the various semantics for transactional memory. These primitives include the ability to start a potentially nested transaction, read and write values within a transaction, abort a transaction, and commit a transaction.

Runtime transactional memory primitives can be provided either by hardware [16][25][19][9] (HTM) or by software [1][9][14][17] (STM) with both approaches having their pros and cons. An HTM provides a significant performance advantage, and enforces atomicity not only between transactional memory accesses, but also between transactional and non-transactional memory accesses. However, a HTM either restricts the size of the transactional code block, or requires complicated HW support. STMs can easily support unbounded transactions, nested transactions with partial rollbacks, and conditional signaling [14], which makes them more convenient from a programming perspective – all published HTM proposals either ignore or restrict the semantics of these features. Finally, an STM can be more easily integrated with existing tools, offers an easier adoption route for programmers, and permits experimentation with semantics of language features.

In this paper, we present the interface and the implementation of a high performance STM system built within an experimental multi-core runtime called McRT. Most prior STM systems have gone to great lengths to guarantee non-blocking properties. In contrast, the McRT-STM implements transactions using strict two-phase locking [8] and contains commit and abort sequences that are blocking. This makes the implementation more efficient, and also allows McRT-STM to implement a range of design alternatives. This paper makes the following novel contributions:

1. It is the first to perform a detailed quantitative analysis of the pros and cons of various STM design tradeoffs and overheads, such as optimistic versus pessimistic concurrency,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPoPP'06 March 29-31, 2006, New York, NY, USA.
Copyright © 2006 ACM 1-59593-189-9/06/0003...\$5.00

write buffering versus undo logging, and cache line based versus object based conflict detection.

2. It presents the first object-based conflict detection algorithm for C/C++ and other applications that use explicit memory management. We believe the McRT-STM is the first STM that can simultaneously support both object-based and cache-line based conflict detection.
3. It describes a novel STM design that leverages other McRT components. Prior research has concentrated on stand-alone STM systems with a non-blocking design which imposes a significant overhead. In contrast, the McRT-STM avoids a non-blocking design but instead leverages the McRT cooperative scheduler to prevent an inactive transaction from blocking running transactions.
4. It describes a novel software MCAS (atomic multi-word compare and swap) implementation that coexists with the STM, and works on arbitrary values. The software MCAS implementation allows a number of optimizations over the general STM.
5. It provides detailed performance results of the STM on a number of concurrent data structures, compares the performance with that of fine-grained and coarse-grained locking, provides a breakdown of the STM overheads, and presents the contention behavior. We show how the STM performs on a “real” application (the sendmail spam filter). We believe this is among the first studies of STM on a large non-synthetic benchmark.

The rest of the paper is organized as follows. Section 2 describes our experimental framework. Section 3 evaluates the different STM design tradeoffs. Section 4 describes the McRT-STM design. Section 5 describes our MCAS implementation. Section 6 evaluates the STM implementation. The last two sections present related work and conclusions.

2. Experimental framework

We have built our STM within McRT, an experimental multi-core runtime. At its core, McRT contains a thread scheduler, a synchronization framework, a scalable memory manager, and the STM system, as well as other components. Sitting on top of the core services, a set of adapters translate the threading calls in different programming models into the McRT core API. McRT supports OpenMP, Pthreads, and the ORP [5] Java virtual machine. The whole stack runs on a variety of platforms such as IA-32 Linux and IA-32 Windows.

As concurrent workloads, we use a hashtable, a balanced binary search tree, a B-Tree, and a linked list. The hashtable is organized as 256 buckets with each bucket being a linked list of elements. All operations search the hashtable for a given element, with updates and deletes also modifying the hash table. Threads pick an element at random and then perform insertion, deletion or lookup. The probability of any two threads colliding is high $C(16,2)/256$, but the probability of many threads colliding is low. Thus, the hashtable emulates a workload that has good throughput, even though transactions may abort. The binary search tree has conflicts when updates cause rotations; moreover, nodes near the root of the tree become a bottleneck when they are updated, thus hindering throughput. The B-Tree is an 11-order balanced B-Tree, a complex data structure commonly used in

databases. The linked list emulates a workload with the worst case STM behavior since list operations exhibit very little concurrency. These benchmarks represent common data structures used in many applications.

The measurements presented in this paper were gathered on a 16-processor IBM x445 SMP system with Xeon MP 2.2 GHz Xeon processors running RedHat Linux EL3. The SMP system is arranged in clusters of 4 processors, with processors within each cluster sharing a 64MB L4 cache. Each processor has private L1 (8KB), L2 (512KB), and L3 (2MB) cache.

3. STM design tradeoffs

This section discusses the different STM design tradeoffs and explains why we chose particular points in the design space. In most cases, we implemented the different STM variants to enable a quantitative comparison. This section also presents the quantitative data to reinforce the McRT-STM design decisions.

3.1 Non-blocking guarantees

The McRT STM uses a strict two-phase locking protocol [8] to implement the transaction manager. The STM maps each memory location to a unique lock, and acquires all relevant locks before committing a transaction. Unlike conventional STMs [11][15], the McRT STM has small code sections, related to commit and abort sequences, which are blocking. However, if a transaction T1 is waiting for a lock to be released by another active transaction T2, then T1 can abort T2. Other than the obvious performance reasons we made the design choice for the following complimentary reasons:

- We believe that non-blocking properties should be enforced through appropriate scheduler hooks to control preemption in designated code sections. The McRT scheduler uses cooperative pre-emption that reduces (and in many cases eliminates) blocking problems. Moreover, the scheduler executes at the user-level, so the STM-scheduler interaction is inexpensive since it happens through function calls.
- As single thread performance hits the power wall, processor architects have turned to chip multiprocessors (CMP). Emerging CMPs will use Moore’s law to aggressively increase the number of processing cores, perhaps to tens of cores by the end of the decade [20]. This makes preemption much less of an issue.

The McRT-STM benefits in the following ways by abandoning the non-blocking guarantee:

- Transaction aborts are reduced. We initially used a non-blocking implementation, but performance was poor due to a large number of transaction aborts. The lock-based STM implementation was also simpler and more efficient.
- Optimization opportunities are exposed. In many cases we can detect early that a transaction will eventually commit. Section 5 shows an example of how we take advantage of this; a compiler should also be able to exploit this.
- Memory management is simplified. Prior STMs had to resort to complex memory management schemes such as hazard pointers[22] since the STMs were designed to be non-blocking. On the other hand, the McRT-STM can use a standard memory allocator for its internal data structures, for example allocating and freeing transaction logs.

- Integrates better with a transaction monitor. A transactional memory implementation may need to integrate with a transaction processing monitor, which entails a blocking implementation. The TM implementation may also need to support other features that require blocking implementations, such as remote procedure calls.

3.1.1 Deadlock avoidance

Since transactions wait for locks to be released, the STM needs to avoid deadlock. One option is for the STM to detect deadlock by creating a graph of waiting transactions. The STM could then detect a cycle of waiting transactions, and abort one of the transactions. This would require maintaining additional state whenever locks are acquired, which would slow down all transactional reads and writes.

Another option is to wait for a finite amount of time for a lock to be released and then abort. This does not incur an overhead in the absence of conflicts, but could lead to some “false positives”. Prior work in the database community [8] has shown that the probability of deadlocks is proportional to NW^4/L^2 and hence small: (N =no. of concurrent transactions, W = no. of locks acquired on average (~no. of stores), L = total no. of locks that can be acquired (~total no. of shared objects)). While database access patterns may differ from shared variable access patterns in an application, nevertheless we considered this to be a good starting point and adopted this approach in our STM.

3.1.2 Data conflicts and contention

In our STM, data conflicts manifest as lock contention. If a reader or a writer tries to access a location that has been updated by another active transaction, then the reader will find that the write lock has been acquired. A writer may end up accessing a location that has been read by another active transaction, but the conflict manifests when the transaction validates its read set.

Therefore, contention management between transactions boils down to reader/writer actions when they find that a lock is taken. Our STM tries to maximize throughput by almost always making readers/writers wait when a lock is not available. A transaction will abort another transaction only when the thread running the other transaction has yielded the processor. The McRT-STM leverages the scheduler for this: If a transaction T_1 finds that a lock has been acquired, it queries the McRT scheduler, and only if the lock owner T_2 is not running does T_1 abort T_2 .

3.2 Locking mechanism

A lock-based transaction implementation can use two different locking algorithms for enforcing transactional semantics: It can use reader-writer locks, or it can use read-versioning combined with writer locks.

3.2.1 Reader-writer locking

In this scheme, the lock words corresponding to memory locations are used as reader-writer locks. A reader takes a read lock before loading the memory value, while a writer takes a write lock before a speculative update. Multiple readers or a single writer are allowed to acquire the lock at any time. Acquiring a read lock prevents any writer from updating a location that an active transaction has read. Acquiring the write lock prevents any reader from loading a speculative value written by an active transaction. A transaction maintains a log of the locks (and their flavor) that it has acquired. At commit, the transaction releases all its locks. Thus the reader-writer lock mechanism enforces atomicity.

Further, maintaining an undo-log for aborts also allows a transaction to make in-place updates. The advantage of using reader locks is that it prevents a future writer from creating a data conflict; thus, it allows a compiler to reorder code to proactively acquire locks and once all the locks are acquired, to optimize the code generation knowing the transaction is not going to get aborted.

Unfortunately, conventional reader-writer locks can not be used in a STM. This is because a transaction may first read a location, and then write into it later. This implies that a reader lock may need to be converted into a writer lock, yet the reader-writer lock semantics still needs to be preserved. The McRT STM uses a novel, yet efficient scheme to implement reader-writer locks with support for dynamically upgrading readers into writers.

We use a 32 bit integer as the reader-writer lock. The lower 3 bits have special meaning: the *Notify* (or N) bit is set when a reader has requested notification, and unset when there are no waiting readers. The *Upgrade* (or U) bit is set when a reader wants to upgrade, it is unset when no readers are waiting for an upgrade. The *Reader* (or R) bit is always 0 when a writer owns the lock, otherwise it is 1. When a transaction acquires a write lock, it stores a pointer to its descriptor (a transaction local structure). The pointer is allocated on an 8 byte boundary, so the lower 3 bits are all zero. Once a writer has acquired the lock, neither upgrades nor notifications can be requested since no readers will be able to acquire the lock until the current transaction releases the write lock (at a commit or abort). This ensures that the lower 3 bits remain unset while a writer has acquired the lock. Readers first check that no writer has the lock (the lower 3 bits are non-zero) and then atomically increment the value of the lockword by $0x8$ to acquire the lock, and decrement it by the same amount on release. This ensures that the bit pattern of the lower bits remains unperturbed. When readers have acquired the lock, the upper 29 bits store the number of readers. The initial value of the lock word is $0x4$ which implies that no writer has it, and no readers have acquired it. The upgrade and notification bits are unset.

When a reader wants to upgrade, it atomically tries to set the U bit. If it succeeds, it waits for the current readers to release the read locks, and then acquires the write lock by inserting a pointer to its transaction descriptor. Moreover, incoming readers back off from acquiring a read lock if they notice that the upgrade bit is set. During upgrade, if a reader finds that the U bit is already set, then it aborts since another reader has obtained the right to upgrade, and the consequent write would create a data conflict.

A transaction may sometimes wait for values in its read set to change. We use the N bit to set this up. Every lock has an associated wait list. When a reader wants notification, it sets the N bit, adds itself to the list of waiters for this lock, and then releases the read lock. There is no race condition between setting the N bit and adding oneself to the set of waiters since no writer can acquire the lock before the read lock release. Moreover, setting the N bit is an idempotent operation, so readers can set it multiple times. When a writer releases a lock, it checks the wait list, and wakes up any waiting readers.

3.2.2 Read versioning and write locking

In this scheme, a writer takes a lock before modifying the memory location, but a reader uses versioning to detect data conflicts. This scheme is similar to the one presented in [11].

We use a 32 bit integer as the lock-word that can be in one of two states. It can either be owned by a writer, or it can contain a version number. The lower 3 bits have special meaning as before. The N bit is used for notifications, the U bit is unused, and the R bit is 1 for version numbers and 0 for a writer lock. A writer acquires a lock by storing a pointer to its descriptor (all descriptors are allocated on 8 byte boundaries.)

Before reading a memory location, a reader checks the lock-word to make sure no writers currently own the lock (the lower 3 bits are non-zero). The reader adds the lock-word to its read set and remembers the version number. At commit, readers validate that the version numbers of the locks in their read set haven't changed. A writer acquires the lock before updating the memory location. The writer also remembers the version number in its write set. During lock release, it inserts a new version number into the lock-word; the new version number is obtained by adding 0x8 to the old version number. Thus, whenever a location changes, the version number of the corresponding lock-word monotonically increases. This ensures that the R bit remains set in the new version number. The initial value of the lock word is 0x4. While our current implementation uses 32 bits, nevertheless we intend to switch to 64 bit version numbers to avoid overflow.

Readers use the notification bit for wakeup signals as before. The only difference is that the wait locations are guarded by a mutex. Readers and writers acquire the mutex to allow race-free signaling. Upgrades are handled during validation. During a write lock acquire transactions record the lock's current version number in their write sets; during validation we check that the version number in the read set matches the corresponding one in the write set.

Figure 1 compares the read-versioning with the reader-writer locks. We implemented reader-writer locks both with reader priority and with writer priority (implemented using an additional bit to indicate waiting readers and writers). The performance was similar in both cases; therefore, we compare the read-versioning scheme to the reader lock with reader priority. A value greater than 1 implies that the workload takes longer to finish with reader-locks. As is evident from the figures, *read versioning performs an order of magnitude better than reader locking*. There are 2 primary reasons for this: (1) read versioning eliminates readers from atomically writing to the lock word, improving the cache effects; and (2) dynamic reader to writer upgrades are expensive because an upgrade needs to wait for all current readers to relinquish their read lock; moreover, relinquishing the read lock requires each reader to perform an atomic operation on the lock with destructive cache effects. Reader upgrades also trigger a chain of aborts in all but one of the concurrent readers. In some cases, a compiler can proactively acquire the write lock and avoid the upgrade, but in many cases the upgrade is unavoidable. Reader locking performs worse with more processors because the caching effects get aggravated.

Some researchers have also proposed maintaining explicit reader lists. *Maintaining explicit reader lists would also effectively convert reads into writes and suffer from the same cache degradation and poor performance compared to read-versioning.*

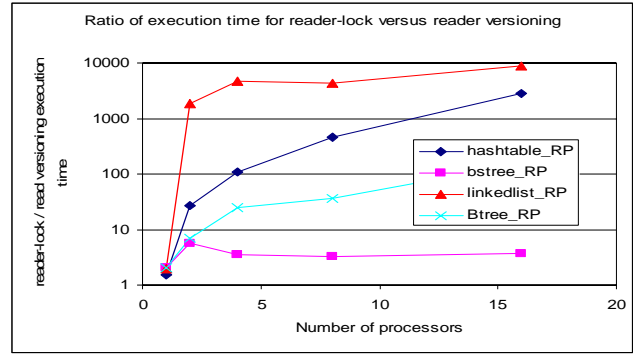


Figure 1: Comparison of reader-version with reader-lock

Reader locks perform much worse for hashtable and linked lists since the transactional region contains many more reads than for trees. In the hashtable and the linked list, the average number of reads per transaction is proportional to the number of elements, whereas in the trees, it is proportional to the logarithm of the number of elements.

3.3 Write buffering versus undo logging

A lock-based STM can handle transactional updates in two ways: (1) the STM can update memory in-place during the transaction and maintain an undo log to rollback state on an abort; or (2) the STM can buffer speculative writes till the commit point, and update memory only after the transaction has committed.

One advantage of an undo-based scheme is that it makes the commit path faster since values do not need to be installed. Another advantage is that read after write (RAW) cases can be handled trivially. A STM must make sure that reads from a transactional location return the most recent store. In an undo-based scheme, the most recent value is stored in memory; therefore, the STM can easily read the most recent value. The advantage of write-buffering is that lock acquires can be postponed until the commit point, which reduces the time during which locks are held. In addition, the locks can be acquired in a canonical order (for example, address order), which eliminates the chance of deadlocks. The disadvantage is that the write buffers must be searched on a read to get the most recent update. This requires some form of hashing which makes the implementation of nested transactions inefficient.

Undo logging is also more amenable to compiler optimizations such as CSE and hoisting of read and write barriers. Such optimizations are shown in [1].

We implemented both undo-logging and write-buffering in the McRT-STM, and Figure 2 shows that *undo logging performs better than write buffering*. The main overhead of the write buffering scheme arises from having to search the write logs for the most recent speculative value. Again, since the hashtable transactions have a larger read set, write buffering performs worse. These results do not take any compiler optimizations into account, and the workloads did not have any nested transactions; undo logging would perform even better in the above scenarios.

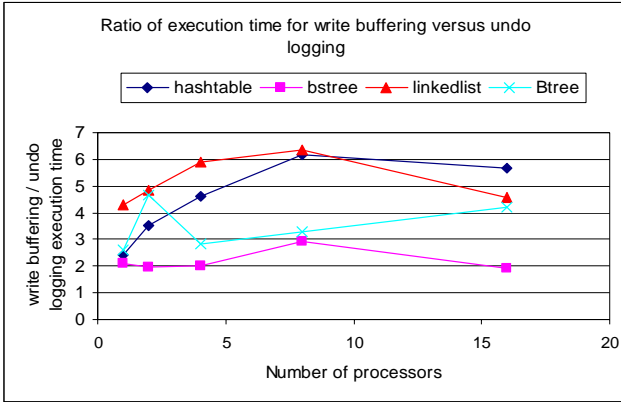


Figure 2: Comparison of undo logging versus write buffering

3.4 Object versus cache-line conflict detection

STMs can detect conflicts at different granularities; for example, at a cache line level or at an object level. Object based conflict detection is more intuitive for the user, and lets a compiler aggressively optimize the transactional code sequences. Consider the code sequence below that manipulates a node in a linked structure.

```

transaction{
    node->key = value;
    node->left = ...;
    node->right = ...
}

```

With object based locking the compiler can generate code to acquire ownership of the node object once for all the updates. This is more difficult in a cache line based scheme since the object may be split across cache lines.

For managed environments such as Java, object based conflict detection can be implemented easily [1]. For unmanaged environments such as C, this is more difficult since we need to map an arbitrary interior pointer to the object base. The McRT-STM leverages the McRT memory manager (McRT-Malloc) in a novel way to enable object based conflict detection in C/C++. McRT-Malloc divides the heap into several smaller blocks which are used to segregate objects based on their size. McRT-Malloc uses the size-segregated heap for objects less than 8K bytes; larger objects are allocated from a large object space (a different heap) and are not size-segregated. Object based locking is only provided for objects that are allocated in the size segregated heap. Large objects, stack-allocated objects, and global variables fall back on cache line locking. This is desirable since object-based locking for large objects would coarsen the granularity of conflict detection.

During startup, McRT-Malloc divides the virtual address space into large and small object spaces. Given an object pointer, a single range check ($pointer - small_object_base_address < small_object_area_size$) suffices to determine whether the object resides in the small object heap. Every size-segregated memory block is aligned on a 16K boundary. At the base of each of these blocks is a small 64 byte area that holds the meta information for this block including the size of all the objects residing in this block. To obtain the block header address we mask the low-order bits of a pointer ($header_addr = pointer \& (block_size - 1)$). The size of the objects in the block is obtained from the header, allowing efficient computation of the “index” of the object in the

block ($index = (pointer - header_size - header_addr) / object_size$).

We investigated two approaches to object-based conflict detection. The first approach allocates locks inline with objects, which incurs an overhead even when no transaction accesses an object, but has the benefit of good cache locality. In particular, acquiring the lock gets the cache line in exclusive state which avoids any subsequent cache miss on a store. The second approach associates a table of per-object locks with each memory block used by the allocator. By placing the locks on the side, it is possible to allocate the table in a demand-driven fashion, reducing memory wasted for non-transactional execution at the cost of worse cache locality.

Cache line (hashing) based conflict detection is easier to implement in the STM. Given an address, we mask off the lower bits of the address to get the cache line address. The STM maintains an array of locks that is indexed with the cache line address (of the memory location) to retrieve the corresponding lock. (We also mask off the higher bits. This reduces the number of locks that the STM maintains, but does not affect performance noticeably). This boils down to a single masking operation, a shift, and an addition to realize the address of the lock.

Figure 3 compares all the approaches. Values less than 1 indicate that object-based execution time was less than the cache-line based execution time. The workloads performed 64K operations with 80% being updates. In the hashtable and the binary search tree the inlined locks perform the best, while in the linked list the cache-line based scheme works best. The hashtable has low contention; therefore the inlined locks have a beneficial prefetching effect. The linked list sees very high contention; therefore, the inlining leads to cache-line ping-ponging and hurts the performance. The object-based conflict detection does not include the effect of any compiler optimizations; thus we expect better performance after integrating with compiler optimizations.

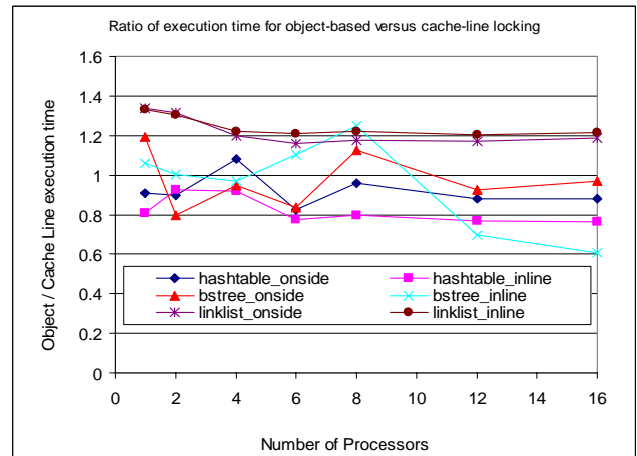


Figure 3: Comparison of object based and cache line locking

4. McRT-STM design

Using the data in Section 3 as a guide, *McRT-STM implements read-versioning and undo-logging. We also support both object-based and cache-line based conflict detection.*

4.1 McRT-STM API

The McRT-STM provides the following runtime primitives:

```
McrtSTMDDescriptor* stmStart (void);
volatile uint32* stmGetLock (McrtSTMDDescriptor *descriptor,
                             void *addr);
uint32 stmReadVersion (McrtSTMDDescriptor* descriptor,
                      volatile uint32* lock);
uint32 stmWriteLockAcquire (McrtSTMDDescriptor* descriptor,
                           volatile uint32* lock);
void stmUndoLog (McrtSTMDDescriptor* descriptor,
                volatile uint32* addr);
Bool stmValidate(McrtSTMDDescriptor* descriptor);
Bool stmCommit (McrtSTMDDescriptor* descriptor);
void stmAbort (McrtSTMDDescriptor* descriptor,
              uint32 reason);
void stmAbortCurrent (McrtSTMDDescriptor* descriptor,
                    uint32 reason);
void stmMapLogFromBegin(McrtSTMSSB* ssb,
                       void (*callback) (McrtSTMSSB* ssb),
                       McrtSTMSSB* stop);
void stmMapLogFromEnd(McrtSTMSSB* ssb,
                    void (*callback) (McrtSTMSSB * ssb),
                    McrtSTMSSB* stop);
void stmAddAbortHook(McrtSTMDDescriptor* descriptor,
                    void (*callback)(McrtSTMDDescriptor* dsc, void* arg),
                    void* arg);
void stmAddCommitHook(McrtSTMDDescriptor* descriptor,
                    void (*callback)(McrtSTMDDescriptor* dsc, void* arg),
                    void* arg);
McrtSTMDDescriptor* stmGetDescriptor();
McrtSTMSSB* stmGetReadSet(McrtSTMDDescriptor*);
McrtSTMSSB* stmGetWriteSet(McrtSTMDDescriptor*);
McrtSTMSSB* stmGetUndoLog(McrtSTMDDescriptor*);
```

A call to `stmStart` initiates a transaction letting the STM initialize its internal data structures. The `stmStart` function also maintains the dynamic nesting depth. The `stmGetLock` function is used to map an address to a unique lock; the mapping can be either on a cache-line basis or on an object basis and can be set dynamically. Clients of the McRT-STM can also override the default `stmGetLock` function and provide their own function to map addresses to locks. This allows us to decouple the granularity of conflict detection from the unit of logging/updates. The `stmReadVersion`, `stmWriteLockAcquire`, and `stmUndoLog` functions are used to access shared memory inside a transaction. The `stmReadVersion` takes a lock address (corresponding to a memory location) and stores the version number in the read set if the lock is currently not owned. The function returns immediately if the calling transaction owns the lock. Otherwise it calls the

contention manager which may cause it to wait and retry, or ultimately abort the transaction. Since we use in-place updates, reads are done directly from memory. The `stmWriteLockAcquire` takes ownership of a lock if it is currently not owned, or returns immediately if the calling transaction owns it. If some other transaction owns the lock, it calls the contention manager, which may decide to wait and retry, or may ultimately decide to abort the transaction. Both the `stmReadVersion` and `stmWriteLockAcquire` functions return the version number of the lock. The `stmUndoLog` is used to remember the old value of a location before doing an in-place update. Updates are always word sized. The `stmValidate` function validates the transaction by checking that the version numbers in the read set match the current version numbers of the locks. The `stmCommit` function marks the end of a transaction. It validates the transaction and releases all the locks acquired by the transaction (and recorded in the write set). The transaction may be aborted at any time due to a data conflict. The `stmAbort` aborts the entire transaction, while the `stmAbortCurrent` aborts only the innermost transaction. On an abort the memory values are reverted, the write locks are released, and the contention manager is invoked which ultimately retries the transaction. The McRT-STM supports explicit user aborts that are used for implementing the `retry-orElse` [14] construct. The `stmMapLogFromBegin` and `stmMapLogFromEnd` functions take a call back function and iterate over the logs from the beginning and from the end respectively. The log entry pointed to by `stop` tells the iterator when to terminate. The McRT-STM also allows a client to add abort and commit callbacks (`stmAddAbortHook` and `stmAddCommitHook`) that are invoked if a transaction gets aborted or committed. The transaction descriptor is threaded through all calls, therefore the STM exports a function (`stmGetDescriptor`) to get the current transaction descriptor. The remaining functions, (`stmGetReadSet`, `stmGetWriteSet`, `stmGetUndoLog`) are used for accessing the transaction's logs which can then be passed to the iterators.

For the applications studied in this paper all the calls to the STM library are introduced by a simple manual expansion of all shared-memory locations accessed inside atomic regions. The library calls may also be introduced automatically by a compiler [1].

4.2 McRT-STM data structures

Every transaction uses a descriptor for storing transaction meta-data. The descriptor is created during thread initialization and stored in the thread local storage (TLS). The `stmStart` function retrieves the descriptor from the TLS; the descriptor is then passed to all the runtime functions to avoid repeated TLS lookups. The descriptor contains the following fields

- `transactionState` → Active/Committed/Aborted/Wait
- `transactionDepth` → Nesting depth
- `writeLocksAcquireLog`, `currentIndex` → write set locks, pointer to the head of the log
- `readLocksAcquireLog`, `currentIndex` → read set, pointer to the head of the log
- `updatedLocationsLog`, `currentIndex` → original value log, pointer to the head of the log

The logs store the read and write sets (address-value pairs) and are organized as a sequential store buffer (SSB)[18][3]. These store buffers are allocated in chunks, and when the current chunk runs out, a new chunk is allocated and linked to the last chunk of the SSB as shown in Figure 4. (For simplicity the figure shows the

SSBs allocated as 2 entry chunks, in the implementation we use 128 entry chunks). The descriptor contains a pointer, *currentIndex*, to the head of each log. We assume that nested transactions follow a stack discipline; therefore the logs contain an *index stack* that is used for tracking the read and write sets of transactions at different nesting depths.

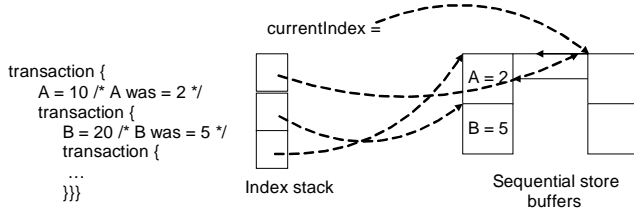


Figure 4: Structure of the transaction logs

Figure 4 shows the updatedLocations log. When a nested transaction is started, the *currentIndex* into the log is pushed onto the index stack. When a nested transaction is committed, the index stack is popped. This effectively merges the read-write sets of the nested transaction with that of the parent transaction, which provides closed nesting semantics. The entries from the top of the stack to the head of the log (*currentIndex*) comprise the state of the currently executing transaction. This allows easy rollback of a nested transaction without affecting the parent transaction. For example, in Figure 4, when the innermost transaction is started, the *currentIndex* points to the head of the log, while the stack contains the log indices where each parent transaction started. As the innermost transaction starts, the write set will be made up of the elements from the top of the stack (the third entry). When the innermost transaction commits, the stack will be popped, and the top of the stack will then point to the second entry and the write set will be made up of the elements from the second entry on to the end of the SSB. Thus the state of the inner transaction will get subsumed into the parent transaction.

5. MCAS implementation

Our STM allows a very efficient multiword compare and swap, MCAS, [13] implementation that works on arbitrary values and coexists with the general STM. Having an efficient MCAS is important for two reasons: (1) MCAS can be used by expert programmers to write concurrent data structures, and (2) MCAS can be a convenient interface to a hardware transactional memory (HTM) implementation. Most HTM proposals [16] have an upper bound on the number of locations that can be accessed inside a transaction. Since a MCAS specifies upfront the number of transactional memory accesses, the transactional library can decide whether to execute the MCAS as a HTM or as a STM. The MCAS API is defined as:

```

Bool MCAS(int k, void* addr[], uint32 oldValue[], uint32 newValue[])

```

The first parameter provides the number of memory locations that will be accessed, the second parameter provides the set of addresses, the third parameter provides the set of expected values, and the last parameter provides the set of new values that will be swapped in if all the memory locations contain the expected values. The operation returns *True* if the new values are swapped in, and *False* otherwise.

The STM and MCAS are integrated and both compute the locks for the transactional memory locations using the same algorithm; therefore, the locations are protected from concurrent access. Only if all the locks are successfully acquired does the MCAS update the memory locations and returns *True*, otherwise it returns *False*, which

enables some optimizations. The workloads we used for evaluating our STM (e.g. hashtable, or binary search tree) are not amenable to the use of MCAS, so we compared the STM with the MCAS on a bounded FIFO queue with concurrent enqueueers and dequeuers. Figure 5 shows the comparison between the MCAS and the STM implementation.

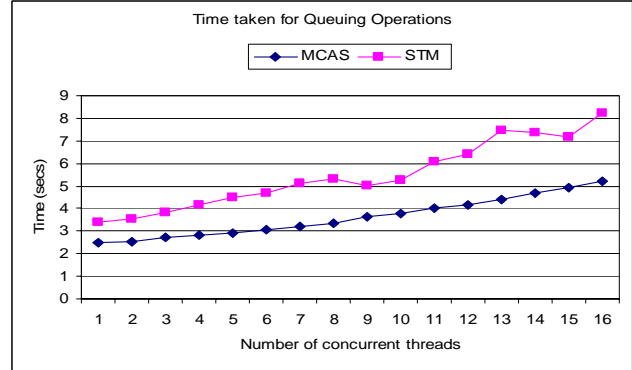


Figure 5: MCAS versus STM

6. STM performance

This section compares the performance of STM with lock-based algorithms and then examines the McRT-STM overheads. The transactional workloads use cache line based conflict detection in all cases. This paper focuses solely on a high performance STM without regard to compiler optimizations; therefore, we did not consider object based conflict detection in the results.

6.1 STM versus locking

Figure 6 provides a baseline comparison between STM and the different locking schemes on the hashtable benchmark. The coarse-grained locking scheme uses a single lock for the entire hashtable. The fine-grained locking scheme uses a lock per bucket. The STM version replaces the lock acquire and lock release calls of the coarse grained version with *stmStart* and *stmCommit* calls. Thus, the programming effort is the same as that of coarse grained locks. The STM version initially starts out with a much higher overhead, but as the number of processors increases, it starts approaching the fine-grained performance. At 16 processors, the STM is about 1.8X the performance of fine-grained locking. In the experiment we set the number of updates to 80% of the hashtable operations. With a higher number of lookups, the STM performs better and approaches the fine-grained locking performance.

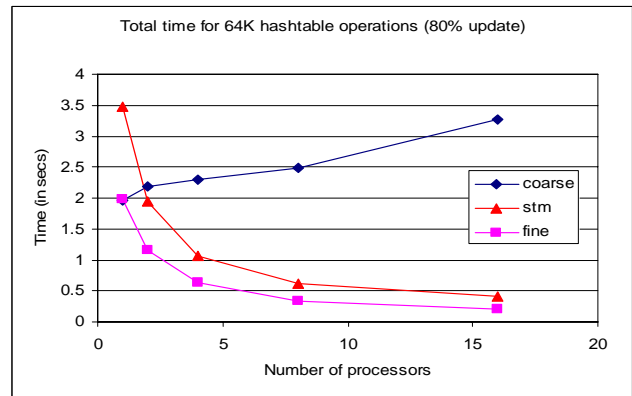


Figure 6: STM versus locking on hashtable

We compare the STM and locking on binary search trees in Figure 7. The lock implementation uses a single lock for the entire tree. The STM performs better than the locking when the proportion of updates is lowered. This arises because the balancing propagates changes across the tree and increases the number of aborts. More importantly, the balancing propagates updates to the root of the tree which severely limits concurrency. The comparison for BST operations without balancing is shown in Figure 8, and the STM outperforms the locking even with higher number of updates. The abort ratios are shown in Figure 9.

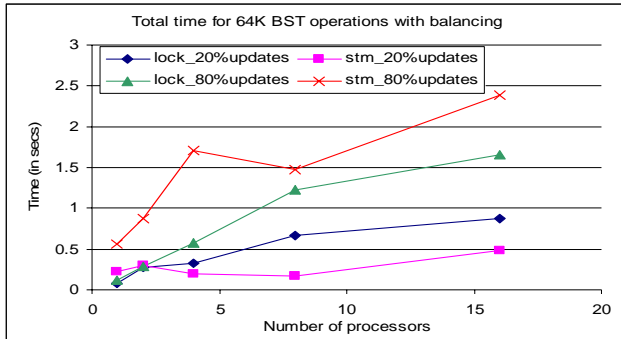


Figure 7: STM versus locking on binary search tree with balancing

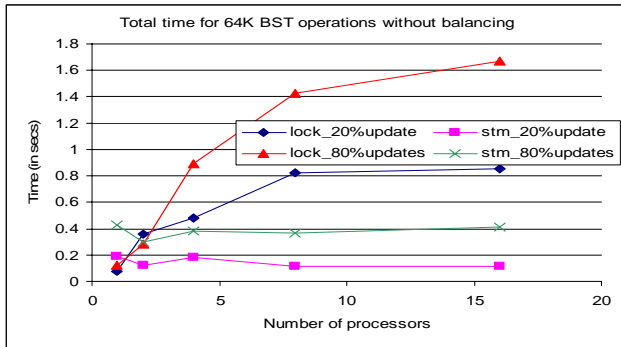


Figure 8: Comparison of STM and locks without balancing binary search tree

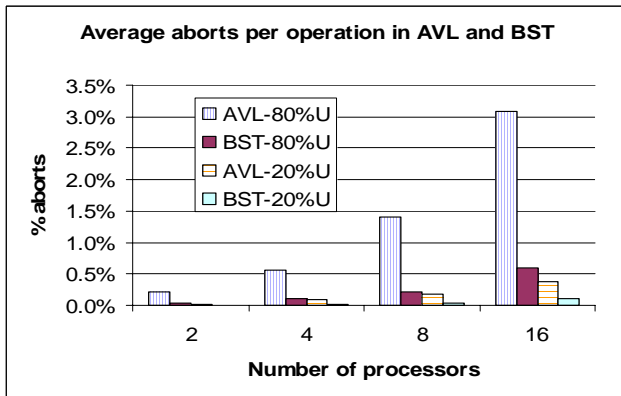


Figure 9: Aborts in the transactional AVL and BST

We compare the STM and lock performance on a sorted linked list in Figure 10. For the locking measurements, we used a single lock for the entire list. When the proportion of updates is lower, the STM performs better, but as the updates increase the performance becomes comparable since the number of aborts increases. Figure 11 shows the results for an unsorted list. In the unsorted list, all insertions happen at the front of the list, which provides no concurrency for the update operations, while the updates are spread out in a sorted list. The abort ratios are given in Figure 12.

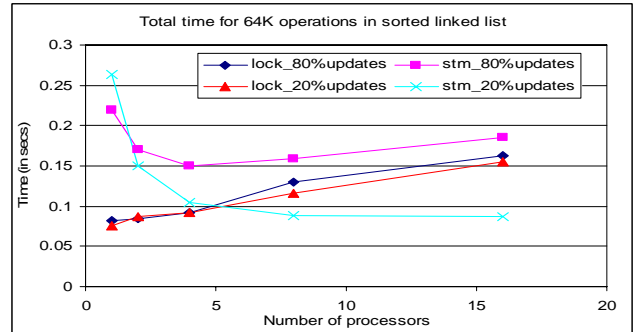


Figure 10: Comparison of STM and lock on sorted link list

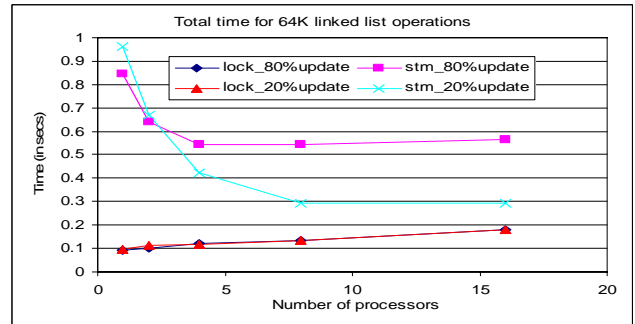


Figure 11: Comparison of lock and STM on unsorted link list

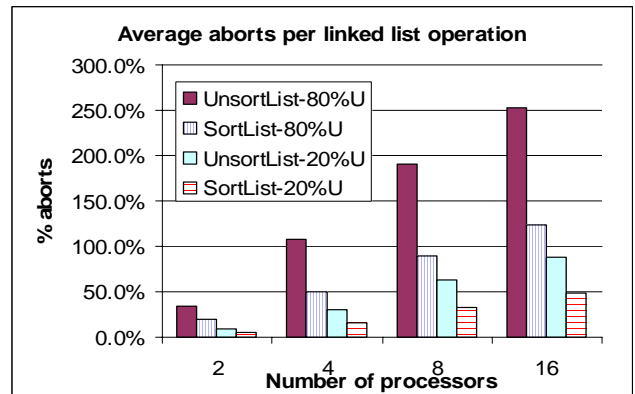


Figure 12: Aborts in the linked list

We also implemented fine-grained locking versions (containing locks at each node) for both the binary search tree and the linked list, but the fine-grained locking performs worse than the STM or

the coarse-grained locking by an order of magnitude. Unlike the hashtable, fine-grained locking requires many lock operations on the linked list and the binary search tree. Since lock operations are expensive on the Xeon, fine-grained locking does not provide any benefit.

We show the STM and lock comparison for the B-Tree in Figure 13. The B-Tree sees few aborts, and therefore the STM outperforms the lock-based code. Even with 80% updates at 16 processors, B-Tree operations get aborted less than 0.5% of the time. The STM performs better as the proportion of lookups increases.

Both the linked list and the binary search tree results show the importance of good contention management in a transactional system. Our cache-line versus object-based conflict detection results also show that contention can play a significant role in determining how the STM performs.

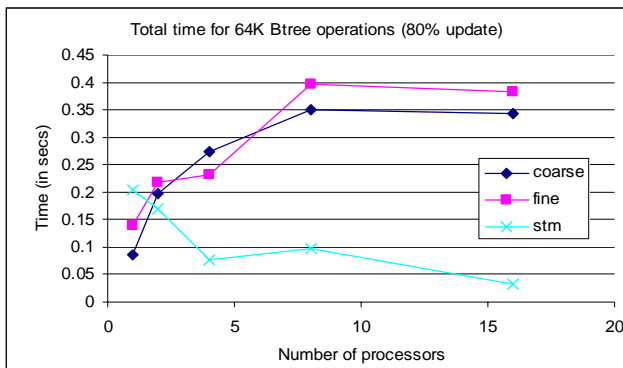


Figure 13: Comparison of STM and locking on B-Tree

6.2 STM overhead breakdown

The breakdown of costs in the transactional workloads is shown in Figure 14. The cost is dominated by the read barrier and the validation costs in all the benchmarks. The hashtable buckets are organized as linked lists, so the number of elements scanned is proportional to the number of insertion operations which contributes to the high overhead from `stmReads`. In the tree on the other hand, the maximum number of reads is proportional to the logarithm of the number of insertion operations. The TLS accesses also arise from the STM; the TLS accesses are mainly for accessing the descriptor and the logs.

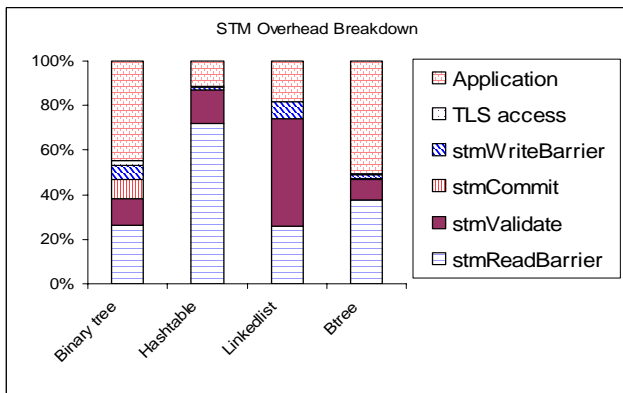


Figure 14: STM cost breakdown in the hashtable

Our STM needs to insert validation checks [11][15] at backward edges to guard against infinite loops and other error conditions. The linked list is traversed in a loop, with the number of backward jumps being equal to the number of list elements. As a result, 80% of the validation calls in the linked list arise from checks on backward edges, and the remaining from validation at commit. If we consider only the commit validations, then the validation cost has the same proportion as the other benchmarks. Techniques like early release [15] would help in reducing the overhead; however, we didn't use early release since it seems to impose the same programming burden as fine-grained locking.

6.3 STM preemption

Since McRT-STM does not guarantee non-blocking properties, we need to ensure that performance does not degrade if the application uses more threads than processors. In this section, we show how the STM performs as the application increases the number of threads. We run the workloads using 16 processors, but use up to 128 user threads. The McRT scheduler multiplexes the user level threads onto 16 kernel threads. Figure 15 shows the execution time of the benchmarks as we increase the number of threads relative to the execution time for the benchmarks at 16 threads. A value greater than 1 indicates that the workload takes longer to execute than with 16 threads.

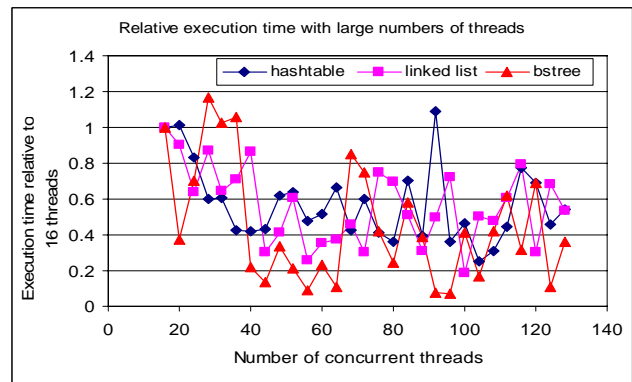


Figure 15: STM performance with large number of threads

As is evident from the charts, there is no performance loss in going from 16 to 128 threads, with a gain in some cases. The increase in performance is due to better load balancing. McRT-STM does not adjust transaction priorities dynamically (in other words does not impose fairness), some transactions win most of the conflicts, while other transactions lose most of the conflicts. Thus, a few of the transactions finish early, while some transactions finish late; the difference between the fastest and the slowest transaction (or the load imbalance) is proportional to the work done by each thread, and decreases as we increase the number of threads. Hence the execution time falls as we increase the number of threads, since less time is wasted in idling. The slight increase in the execution time as we get up to 128 threads arises from runtime and scheduler inefficiencies at high number of threads. The saw-tooth nature of Figure 15 also arises from cooperative preemption since the load balancing works best when the number of application threads is a multiple of 16, and gets worse at other thread counts.

6.4 STM on a non-synthetic workload

This section shows McRT-STM performance on a non-synthetic application. The concurrent workloads serve as a good testbed,

but it is difficult to simulate application characteristics closely through those workloads; for example, contention behavior, proportion of time spent in atomic code, mix of reads and writes, granularity of locking, and so on. So we took the well known sendmail (v8.13.4) application and converted the mutex calls into transaction calls. Sendmail consists of a multithreaded mail filter (milter) API called libmilter (v0.3.0). Through this API, sendmail can make callbacks to sendmail-milter, which in turn calls Mail-SpamAssassin (v3.0.4) to filter out spam from incoming mail. The workload consists of several threads sending emails (50% spam) to the same account. Sendmail goes through the chain of programs mentioned above to filter the spam from these emails. We profiled the lock-based sendmail execution, and found that the application spends about 10% of its time in critical regions, large enough that a significant STM overhead would slow the benchmark noticeably.

As seen in Figure 16, McRT-STM performs comparably to the lock performance. We show the execution time till 8 threads since the other threads are used for sending mail to drive the workload. The key point about STMs is their software engineering benefits, and the challenge for an implementer is to provide the benefits at a reasonable cost. The sendmail result provides preliminary evidence that on commercial applications, the STM and locking performance are comparable.

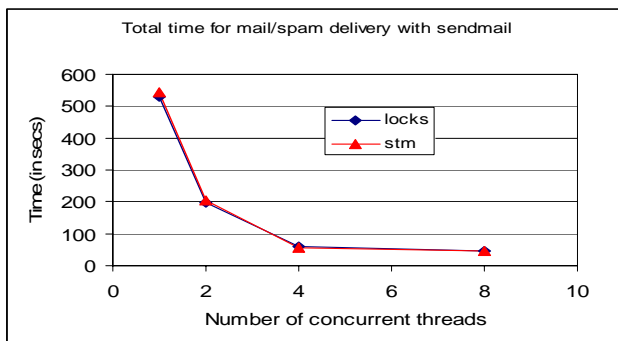


Figure 16: STM and lock behavior for sendmail spam filter

7. Related work

Transactional memory, as applied to programming languages, was first studied by Herlihy and Moss [16], and later by Stone et al. [25]. Both of those relied on a HW implementation, while ours is a completely SW implementation. Shavit and Touitou [24] proposed the first SW only solution scheme handling transactions with statically known read and write sets. More recently, Herlihy et al. [15], Harris and Fraser [11] and Welc, Jagannathan and Hosking [26] have built non-blocking STMs that run on stock hardware and handle transactions with dynamically known read and write sets. Herlihy uses an object based scheme for Java, whereas Harris uses a cache line based scheme. These systems were designed with preemption safety as a major concern. Our system investigates design options made available by hardware with sufficient threads to ameliorate preemption concerns.

Our work differs in several other respects: We leverage the other parts of the runtime system to build a lock-based implementation, we do a detailed quantitative analysis of the various design tradeoffs, we provide a MCAS interface that can be used for interacting with HW transactions, and our STM also provides support for nested transactions with partial aborts.

Ennals [7] discusses a lock-based STM for Java that shares some properties similar to our STM; this paper makes several additional contributions such as the object-based C/C++ STM, leveraging the runtime scheduler and memory manager, and a detailed performance analysis of the STM design space. Marathe, Scherer and Scott [23] do a quantitative analysis of several non-blocking STMs, but their emphasis is on studying different contention policies rather than STM tradeoffs. Harris and Fraser [12] present a locking mechanism for their write logging STM but assume an adversarial scheduler where preemption is a concern. Harris [13] describes a MCAS that works on pointer values, but our MCAS works on arbitrary values.

8. Conclusions

Transactions are a powerful linguistic construct for managing concurrency since they eliminate deadlocks, make it easier to compose atomic primitives, and automatically provide fine-grained concurrency. Transactions may be implemented either in HW or in SW. A software implementation is more versatile and programmer friendly since it imposes no size constraints on the transaction, allows advanced features such as nesting, allows experimentation with usage models, and is easier to interface with tools. In this paper, we present the design and implementation results for McRT-STM, a software transactional memory library for a multi-core runtime.

This paper makes a number of novel contributions: (1) It shows how a STM can leverage other parts of the runtime (e.g., the scheduler and the memory manager) to provide a high-performance STM implementation. (2) It provides the first quantitative analysis of various STM design tradeoffs. (3) It shows performance comparisons with a number of lock-based concurrent data structures, a breakdown of the different STM overheads that can guide further optimizations, and a comparison of the lock-based and transactional versions of the sendmail filter. (4) It also shows a novel MCAS implementation that can be used as a more efficient (but more restricted) form of transactional memory.

9. References

- [1] Adl-Tabatabai, A., Lewis, B.T., Menon, V.S., Murphy, B.R., Saha, B., and Shpeisman, T. Compiler and runtime optimizations for efficient software transactional memory. *To appear PLDI 2006*.
- [2] Allan, E., Chase, D., Luchango, V., Maessen, J., Ryu, S., Steele Jr., G., and Tobin-Hochstadt, S. The Fortress language specification, version 0.618. *Sun Microsystems Technical Report*, April 2005.
- [3] Appel, A. W. 1989. Simple generational garbage collection and fast allocation. *Softw. Pract. Exper.* 19, 2 (Feb. 1989).
- [4] Charles, P., Donawa, C., Ebcioğlu, K., Grothoff, C., Kielstra, A., von Praun, C., Saraswat, and V., Sarkar, V. X10: An object oriented approach to non-uniform cluster computing. *OOPSLA 2005*.
- [5] Cierniak, M., Eng, M., Glew, N., Lewis, B., and Stichnoth, J. The Open Runtime Platform: a flexible high-performance managed runtime environment: Research Articles. *Concurr. Comput. : Pract. Exper.* 17, 5-6 (Apr. 2005).
- [6] Cray Inc. The Chapel language specification, version 0.4. *Technical Report*, Cray Inc. Feb 2005.

- [7] Ennals, R. Cache sensitive software transactional memory. *Technical Report*.
- [8] Gray, J. and Reuter A. Transaction processing: concepts and techniques.
- [9] Hammond, L., Carlstrom, B.D., Wong, V., Hertzberg, B., Chen, M., Kozyrakis, C., and Olukotun, K. Transactional coherence and consistency. *ASPLOS* 2004.
- [10] Harris, T.L. Design choices for language based transactions. *University of Cambridge Computer Laboratory, Tech Report*, Aug 2003.
- [11] Harris, T.L. and Fraser, K. Language support for lightweight transactions. *OOPSLA* 2003
- [12] Harris, T. and Fraser, K. 2005. Revocable locks for non-blocking programming. *PPoPP* 2005
- [13] Harris, T.L., Fraser, K., and Pratt, I.A. A practical multi-word compare and swap operation. *Proceedings of the 16th International Symposium on Distributed Computing*, 2002.
- [14] Harris, T.L., Marlow, S., Peyton Jones, and S., Herlihy, M. Composable memory transactions. *PPoPP* 2005.
- [15] Herlihy, M., Luchango, V., Moir, M., and Scherer III, W.N. Software transactional memory for dynamic sized data structures. *PODC* 2003.
- [16] Herlihy, M. and Moss, J.E.B. Transactional memory: architectural support for lock-free data structures. *ISCA* 1993.
- [17] Hosking, A. and Moss, J.E.B. Nested transactional memory: Model and preliminary Sketches, *SCOOL* 2005.
- [18] Hosking, A. L., Moss, J. E., and Stefanovic, D. A comparative performance evaluation of write barrier implementation. *OOPSLA* 1992.
- [19] Rajwar, R., Herlihy, M., and Lai, K. Virtualizing transactional memory. *ISCA*, 2005.
- [20] Rattner, J. Multicore to the masses. *PACT, Keynote*. 2005.
- [21] Marathe, V. J., Scherer, W. N., and Scott, M. L. 2004. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-Time Support For Scalable Systems* (Houston, Texas, October 22 - 23, 2004). LCR '04, vol. 81. ACM Press, New York, NY, 1-7.
- [22] Michael, M. M. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. *PODC*, 2002.
- [23] Scherer III, W. N. and Scott, M. Contention management in dynamic software transactional memory. *PODC Workshop on Concurrency and Synchronization in Java programs*, 2004.
- [24] Shavit, N., and Touitou, D. Software transactional memory. *PODC* 1995.
- [25] Stone, J. M., Stone, H. S., Heidelberger, P., and Turek, J. 1993. Multiple Reservations and the Oklahoma Update. *IEEE Parallel Distrib. Technol.* 1, 4 (Nov. 1993), 58-71.
- [26] Welc, A, Jagannathan, S and Hosking, A Transactional Monitors for Concurrent Objects, *ECOOP*, 2004
- [27] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R. 2000. Hoard: a scalable memory allocator for multithreaded applications. *ASPLOS* 2000