ABSTRACT

KARIATH, RIYA RAJU. Performance Comparison of Software Transactional Memory Implementations. (Under the direction of Dr. Edward F Gehringer.)

Software Transactional Memory (STM), an optimistic concurrency control mechanism for controlling accesses to shared memory, is a promising alternative to lock-based mutual exclusion strategies. A transaction in this context is each piece of code that executes indivisibly in a shared memory region. Like database transactions, STM transactions preserve linearizability and atomicity properties.

This thesis project presents performance comparisons based on memory, indirection and compute overheads of different STM implementations. More precisely, it compares three STM systems – a non-blocking STM due to Fraser (FSTM), a lock-based STM due to Ennals, and a lock-based STM (TL2) with global version-clock validation due to Dice et.al. A comparison employing diverse classes of STMs helps in a deeper understanding of various design choices and potential trade-offs involved. In particular, suitability of an STM is analyzed versus another STM in a given scenario.

The empirical evaluations done as part of this thesis conclude that Ennals' STM has an edge over TL2 and FSTM, as it performs consistently well on low and high contention settings. The results also suggest that lock-based STMs use less memory than lock-free STMs due to better cache locality.

PERFORMANCE COMPARISON OF SOFTWARE TRANSACTIONAL MEMORY IMPLEMENTATIONS

by
RIYA RAJU KARIATH

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER SCIENCE

Raleigh, North Carolina

2007

APPROVED BY:

_____	_____
Dr. Xiaosong Ma	Dr. Yan Solihin

_____
Dr. Edward F. Gehringer
Chair of Advisory Committee

DEDICATION

To my parents, my husband, Balu and my sister ……

For being there for me always ……

BIOGRAPHY

Riya Raju Kariath was born on 5th December 1981 in Mulanthuruthy, India. She received her Bachelor of Technology in Computer Science and Engineering degree from Model Engineering College, Cochin, India in 2003. She worked as a Software Engineer with iGate Global Solutions Ltd, Bangalore, India from August 2003 to April 2005. She joined the graduate program at the Computer Science Department in North Carolina State University in Fall 2005. She has been working with Dr. Edward F. Gehringer on her master's thesis since May 2006. With the defense of this thesis she is receiving the degree, Master of Science in Computer Science from North Carolina State University.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to the following people without whose help and support I could not have achieved what I have.

My mentor and advisor, Dr. Ed Gehringer for his continued guidance and support. I cannot thank him enough for the numerous hours he spent reading and helping me perfect my thesis.

My parents, for their omniscient words of wisdom and incessant optimism.

My husband, for his encouragement, patience and sense of pride in everything I do.

My sister, for her reassuring words when I needed them most.

Prasad Wagle and Balaji Iyengar for their valuable inputs and suggestions on my thesis work.

Dr. Xiaosong Ma and Dr. Yan Solihin for being on my committee and making time for me.

Dr. Eric Sills for letting me use the HPC lab machines.

Dr. Henry Schaffer for letting me run experiments on SunFire V880.

To God Almighty…

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

In the database community, transactions have long been the natural way of dealing with concurrent programming [1]. Transactions do not suffer from deadlocks, priority inversion, convoying or lack of fault tolerance. Employing transactions provides a simpler programming model and rids the programmer of the need to keep track of locks in the program. These benefits have resulted in a renewed interest in transactional programming [2,3] in recent times. Transactional Memory (TM) has its roots in applying transactions to control accesses to a shared memory region. Transactions, in this context, are any piece of code that accesses a shared memory region. These transactions like their database counterparts, preserve linearizability and atomicity properties [2]. Transactional memory is an optimistic concurrency-control mechanism, analogous to database transactions, for controlling accesses to a shared memory region in concurrent programming.

The original idea of a transactional memory with hardware support was proposed by Herilhy et. al [2]. Shavit and Touitou [3] further extended this idea to a software only TM (STM). These pioneering works have triggered the development of myriad versions and extensions of hardware, software and hybrid TM implementations [4].

To compare the performance of different STM implementations, various micro-benchmarks are used. Nearly all of these micro-benchmarks measure the effectiveness of an STM as CPU time per operation per μs by varying contention and scalability parameters. These micro benchmarks more precisely are different classes of parallel search structures [5-7] – red-black trees, skip lists, or binary search trees. These parallel search structures are especially suited for STM performance comparisons as they inherently exploit the level of concurrency of the underlying STM implementation.

On the flip side, all of these micro-benchmarks focus only on how fast an STM is relative to another by measuring the throughput in low- and high-contention settings. In the TL2 paper [6] by Dice et.al , the throughput of small and large red-black trees with read-dominated and write-dominated workloads for locks and various STM implementations are presented. The red-black trees are classified as small (100, 200) and large (1000, 2000) based on their key range. These results are further used to substantiate the claim that "TL2 is ten-fold faster than a single lock."

These kinds of comparisons overlook many subtle STM overheads including memory usage and memory bandwidth. An STM that is fast but is memory intensive may not be better than locks because locks tend to use less memory. This work argues that being faster than another with respect to throughput should not be the only criterion by which the merit of an STM is determined.

To make this case, in my thesis project I attempt to present performance comparisons based on memory, indirection and compute overheads of three different STM implementations. They include a non-blocking (lock-free) STM due to Fraser (FSTM), a lock-based STM due to Ennals, and a lock-based STM (TL2) with global version-clock validation due to Dice et.al. In particular, to assess the suitability of one STM versus another in a given scenario, we need an objective comparison involving non-blocking STMs with lock-based STMs. This research is part of a bigger picture that assesses the utility of STMs for improving garbage collection in object-oriented languages like Java™.

The thesis is structured as follows. Chapter 2 gives an overview of the past work that has been done in STMs. Chapter 3 introduces the concepts and terminology used in this research. The performance metrics for comparing the STMs and the experimental framework are described in Chapter 4. Chapter 5 concludes this thesis.

Chapter 2

RELATED WORK

Shavit and Touitou [3] proposed the first ever software transactional memory. Their STM implementation was non-blocking, did no recursive helping and could be implemented using limited hardware support. To measure the throughput on varying numbers of processors, they employed four micro-benchmarks that differ in the size of data structure and amount of parallelism. They are counting, resource allocation, priority queue and doubly linked queue. The authors concluded that the non-blocking STM had the most potential for parallelism compared to its predecessors. Their algorithm only works for static transactions—transactions for which the set of locations accessed is known in advance. This is a significant limitation, because, for objects created dynamically, it is not always possible to determine statically the set of locations accessed.

Herlihy et al. [8] addressed this limitation by implementing an obstruction-free STM called the DSTM, with support for dynamic transactions. The idea of contention management was introduced to overcome the livelock problem [8] common with obstruction-free STMs. Choosing the appropriate contention manager is critical to performance. Herlihy et al. have provided implementations of aggressive and polite contention managers. Performance of DSTM is assessed for three micro-benchmarks: the two contention managers using integer set with and without early release, and red-black tree. The basis for comparison is a simple linked list synchronized with a single lock. The red-black tree benchmark was found to be most competitive among the benchmarks and clocked the best results. There was a marked degradation in performance for all the micro-benchmarks when the number of threads exceeded the number of processors.

A lock-free object-based STM (FSTM), was developed by Fraser [5]. This STM employs recursive helping—the current transaction *helps* a higher priority conflicting transaction to

finish. This means that the current transaction executes and passes the parameters that the conflicting transaction require to complete the operation. The transaction that is being helped might already be helping another conflicting transaction; thus this is "recursive helping." The STM enforces a global total order for acquired transactions to ensure that at least one process progresses despite contention. A detailed study of FSTM is presented later in chapter 3 of the thesis. For benchmarking purposes the effect of varying contention on red-black trees is studied. The scalability of the FSTM implementation is studied on a red-black tree benchmark as parallelism is increased to up to 90 processors. The red-black tree benchmark shows a better performance for FSTM than DSTM [8].

To ensure comparable performance on both read-dominated and write-dominated workloads Marathe et al. [9] proposed the obstruction-free adaptive software transactional memory (ASTM). This basic ASTM design (referred to simply as "ASTM") is adaptive in the sense that it can switch between direct or indirect object referencing to take advantage of read-dominated and write-dominated workloads, respectively. There are no levels of indirection to access the object's data from the object's metadata with direct object referencing. A history-based heuristic is also deployed to further benefit from workload distribution. Two variants – *eager ASTM* and *lazy ASTM* are presented in [9]. In eager ASTM all objects opened in write mode are immediately acquired. The acquiring of writable objects is delayed until commit time in Lazy ASTM. IntSet and LFUCache micro-benchmarks are employed to study the write-dominated workloads. To study read-dominated workloads RBTree and InsetRelease benchmarks are utilized. In all of write-dominated workloads ASTM performance is comparable to DSTM [8]. The basic ASTM and its two variants are clear winners with respect to DSTM in read-dominated workloads.

Herlihy et al. [10] recently published a revised implementation of DSTM called DSTM2. The best feature of DSTM2 is that it lets the user write his/her own synchronization and recovery mechanisms in the form of transactional factories that transform stylized sequential interfaces into transactionally synchronized data structures. DSTM2 is the first ever STM intended to be used like a transactional memory library with the scope for user customization. The

authors have provided sample obstruction-free and shadow-factory implementations of DSTM2. The obstruction-free factory algorithm is identical to that of DSTM but additionally provides a implementation of the `clone()` method that creates a shallow copy of the transaction object. In the shadow-factory implementation, each field in the object has a shadow field that stores the tentative update or the old value. In the obstruction-free version, an object is represented in three levels – a start cell that holds reference to the object locator (object metadata), the object locator that has references to the status of the last transaction that opened the object, the old and new versions of the object. The object's old state is restored when the transaction aborts, and the current state is made durable when the transaction commits. List and skip list benchmarks are used to compare the performance of the obstruction-free factory and shadow-factory implementations. The list benchmark consists of a sorted list on to which numbers are randomly inserted, searched and removed. The same operations are carried out on a skip list for the skip-list benchmark. The shadow factory has a higher throughput than the obstruction-free factory with both the benchmarks. The effect is pronounced when the percentage of updates decreases. This is attributed to the overhead of reading an object in the obstruction-free implementation. Two levels of indirection are needed to read the data object from the encapsulated transaction object.

Ennals in [7] argued that even obstruction freedom is unnecessary work for an STM. For a simpler design, he implemented a lock-based STM known as Ennals' STM. The object data and object metadata are stored next to each other, and no indirection is required to access the object data. This design is not non-blocking and lets the transactions lock an object when it is opened for writing (encounter-time locking). In the event of a conflict, the current transaction waits to acquire a lock on the object (and begin writing) until the conflicting transaction finishes with the concurrent object. All locks are released only during commit or at abort time. To reduce the frequency of contention, the number of active transactions should be strictly less than or equal to the number of available processing cores. An elaborate discussion of Ennals' STM follows later in chapter 3 of this thesis. Red-black trees and skip lists are the micro-benchmarks employed to compare Ennals' STM with FSTM [5] and DSTM [8]. Scalability (up to 90 processors) under low contention, and performance under

varying contention of the three STMs are evaluated. Ennals' experiments showed that Ennals' STM is the clear winner in all the cases.

Following on the lines of Ennals' STM, Dice et.al [6] designed and implemented TL2, a lock-based STM. Unlike other lock-based STMs, TL2 avoids most periods of unsafe executions (and therefore aborts fewer transactions). Unsafe executions result from invalid transactions that have seen inconsistent data. It also fits seamlessly with any systems memory lifecycle without the need for special execution environments to sandbox the side-effects of unsafe executions. Ennals' STM and FSTM abort invalid transactions and convert them into transaction retries. Unlike Ennals' STM the objects opened for writing with TL2 are locked only at commit time (commit-time locking) to reduce the window of conflict with other transactions. A global version-clock is used for detecting inconsistent transactions that have seen dirty data. A detailed discussion on TL2 follows in chapter 3 of this thesis. For empirical performance evaluations, the throughput of small and large red-black trees is measured with varying proportions of puts and deletes. The key range [100,200] generates a small red-black tree, and the key range of [1000,2000] generates a large red-black tree. TL2 gives a better throughput than Ennals' STM and FSTM on all four cases [6]. This is attributed to the reduced window of contention during commit-time locking.

A tabular listing of the micro-benchmarks used by all the above STMs is given in the Appendix.

Chapter 3

OVERVIEW OF SAMPLE STM SYSTEMS

The three STMs; FSTM, Ennals' STM, and TL2 employed for performance comparisons in this thesis are described in greater detail in this chapter. All three STMs support transactions that are unbounded (no limit on the number of memory locations accessed by the transaction) and dynamic (the set of memory locations accessed by the transaction is determined at run time).

3.1. FSTM

Keir Fraser [5] developed a lock-free object-based dynamic STM as part of his PhD thesis. FSTM employs recursive helping and an enforced global total order for transactions to ensure that at least one process progresses despite contention.

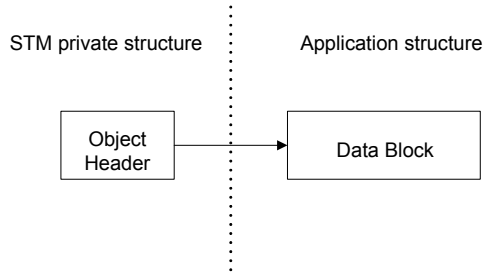

Figure 3.1. Basic unit of concurrency of FSTM: An object (data block) encapsulated in an object header. FSTM has only one level of indirection.

The basic unit of concurrency in the FSTM is an object—which is essentially a contiguous block of memory. These objects are encapsulated in opaque object headers for transactional purposes (Figure 3.1.). Objects are directly accessible for updates or reads only when the object headers are opened as part of a transaction. The transactions and STM objects are

created in a specialized transactional memory area in the main memory. The current contents of an FSTM object are stored within a *data block*. A transaction open call returns a pointer to the data block encapsulated in the object header (the object header is just a pointer to the data block). Multiple object headers can be opened as part of a transaction. Each transaction has a transaction-descriptor structure to keep track of the object headers opened by it. The transaction descriptor has a *status* field which indicates the current status of the transaction—UNDECIDED, READ-CHECKING, COMMITTED or ABORTED. The descriptor maintains a link to a read-only list for objects opened exclusively for reading, and a link to a read-write list for objects opened for writing. Both lists are composed of *object handles*. An *object handle* in the read-write list contains a pointer to the concurrent object's header (*object ref*), a pointer to the concurrent object (*old data*), a pointer to the shadow copy of the concurrent object (*new data*) and a pointer to the next object handle. The object handles in the read-only list have a similar structure except that there is no shadow copy created for the concurrent object. Fig 3.2. illustrates such a transaction descriptor.

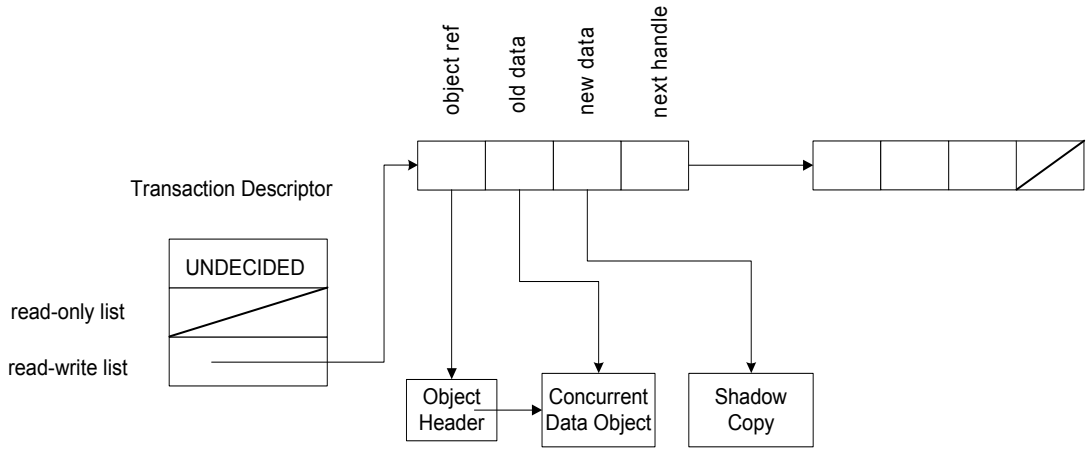

Figure 3.2. Transaction descriptor structure of an active transaction with an empty read-only list.

A transaction accesses a concurrent object by executing an open call on the object's header. The open call adds the object to the read-only list of the transaction if the concurrent object is opened for reading. If the concurrent object is opened for writing it is added to the read-write list. Opening an object does not make the changes made by the transaction visible to other transactions. The commit operation does this. Conflicts with respect to other transactions are detected only at commit time. Depending on the outcome of conflict detection, the transaction either aborts, commits or recursively helps the conflicting transaction. If the transaction successfully commits, its effects are made visible to other transactions.

The commit operation is essentially a multi-word *compare-and-swap* (CAS) operation. The three stages in the commit operation are the acquire phase, decision point and release phase. In the *acquire phase*, the concurrent objects are acquired by the transaction in some global total order by replacing the data-block pointer with a pointer to the transaction descriptor. During the *decision point*, the *status* field of the transaction descriptor is updated to the final outcome indicating the success or failure of the transaction. A transaction is successful only if it is valid and it successfully updates all the shared memory objects in its read-write list. During the *release phase*, the transaction relinquishes all the objects it has acquired in the acquire phase.

It is unnecessary to acquire the concurrent objects opened for read-only access. An example substantiating this claim is the tree-search structure. Any change to the leaf or inner nodes necessitates acquiring the root of the tree. This is a potential bottleneck. Hence only concurrent objects in the read-write list are acquired in the acquire phase. The acquire phase is followed by a read phase. During the read phase, all the concurrent objects in the read-only list of the transaction are checked for consistency to see if they have been changed since they were last read. If a conflict is detected with a transaction that has just COMMITTED, the current transaction has to abort. If a conflict is detected with an UNDECIDED or ABORTED transaction, the current transaction traverses the read-write lists of the conflicting transactions to ascertain whether the object has been changed. If the object has changed, the current transaction aborts. During the *read phase*, the current transaction does not recursively help

the conflicting transaction to complete. Introduction of such a read phase causes transactions to see inconsistent intermediate states. This causes non-serializable transactions to be seen, whose updates do not appear to be atomic.

Fraser [5] gives an example of such a scenario. Transaction T1 opens x_i for writing and y_j for reading. Transaction T2 opens y_j for writing and x_i for reading. According to the FSTM implementation, T1 and T2 will pass their respective read phases and commit successfully. This violates the correctness of the FSTM implementation because T2 should see the update made by T1, and vice versa. To fix this problem a READ-CHECKING state is introduced.

After a transaction finishes the *acquire phase*, it atomically switches to a READ_CHECKING state. The *read phase* begins when the transaction transitions out of the READ_CHECKING state. During the read phase, the transaction walks through its descriptor's read-only list and checks the consistency of each object in that list. If any object in the read-only list of the transaction has changed since the transaction has read it, and this conflicts with a transaction in the UNDECIDED state, the current transaction has to abort. Note that here a current transaction in the READ-CHECKING state does not help a transaction in the UNDECIDED state. So the READ-CHECKING transaction appears to occur before the UNDECIDED transaction. This results in a serializable transaction schedule. If the current transaction detects a conflict with another transaction currently in READ-CHECKING state, depending on the global total order of transactions, the current transaction either helps the conflicting transaction or aborts it. The global order of the transactions is determined based on the machine address of each transaction's descriptor (i.e., in temporal order). If current transaction precedes the conflicting transaction in the global order and both of them are in their respective read phases, then the current transaction can abort the conflicting transaction. Otherwise the conflicting transaction is helped by the current transaction. Once the transaction reaches its decision point, it atomically commits or aborts depending on the decision. The global total ordering of transactions is introduced to ensure that there is no cyclical helping of transactions; i.e., where there is a cycle of transactions in their respective read phases trying to help other transactions. Since the

progress of at least one process is ensured despite contention, the FSTM algorithm is consequently lock-free.

3.2. Ennals' STM

Robert Ennals [7] argued that lock freedom or the more relaxed obstruction freedom should be discarded to make STMs faster. According to Ennals, going to great lengths to make an STM non-blocking is not worth the effort, as blocking behavior is acceptable and simpler to implement. These ideas are incorporated in his lock-based STM referred to as Ennals' STM.

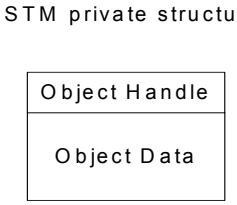

Figure 3.3. The basic unit of concurrency of Ennals' STM: Object data with an STM specific object handle.

The basic unit of concurrency in Ennals' STM is an object (Figure 3.3.). An object handle is added to each object to prevent direct accesses to concurrent objects. Here the object data is stored in place, and there are no levels of indirection to access object data from the object metadata. Non-blocking STMs cannot have object data stored in place [7] . To see why, consider the case when a transaction currently accessing an object is switched out (by a context switch). In the meantime, another transaction tries to access the same object. If the second transaction waits for the first transaction to finish with the object, the first transaction would be blocking the second transaction. If the second transaction overwrites the object it violates correctness. If the second transaction aborts the first, the probability of livelock increases. Consequently Ennals' STM is a blocking lock-based implementation that can take advantage of the cache locality of object data and object metadata (which resides in the

handle). Compared to other STMs this would result in a smaller number of cache lines loaded per read and write.

Similarly to FSTM, the transactions and STM objects are created in a designated transactional memory area in the main memory. Unlike other STMs, this memory is divided in to public and private regions. The public memory contains only objects and is accessible to all transactions. Each transaction has its own private memory that only it can access. The contents of the transaction's private memory are freed only when the transaction commits or aborts. Bookkeeping information is stored here, in the form of a transaction descriptor with pointers to read descriptors and write descriptors. Each transaction maintains separate read descriptors and write descriptors for each object opened for reading and writing respectively. The transaction's write descriptor has a *last version* field that stores the current version of the object being written to. The write descriptor also points to the concurrent object's handle and creates a working copy of the object's data. The read descriptor records the version of the object read and points to the object's handle. Figure 3.4. presents such a transaction descriptor.

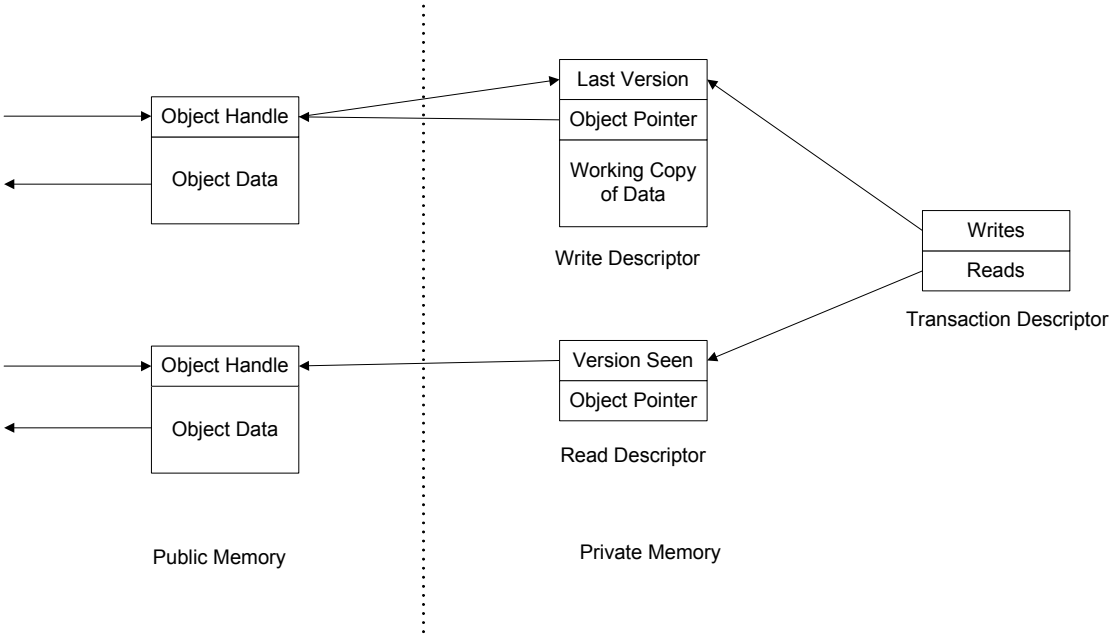

Figure 3.4. Memory Layout for Ennals' STM: Illustrates the transaction descriptor structure of a transaction with one object open for writing and another object open for reading.

To write to an object, a transaction must obtain an exclusive lock on the object. The transaction should also create a working copy of the object's data to which it can write. To obtain an exclusive lock, the transaction should create a new write descriptor, add the object's handle to the write descriptor and make the object's handle point to the transaction's write descriptor. When the current transaction tries to obtain an exclusive lock, the object's handle could either contain a version number or pointer to the write descriptor of another transaction. If the object's handle contains a version number, then the current transaction uses an atomic compare-and-swap operation to replace the version number with a pointer to a new write descriptor. The write descriptor has its last version field set to the version number in the object's handle, and its working copy is initialized to the current version of the object data. If the object's handle is a pointer to a write descriptor, the current transaction will wait until the owning transaction is finished with the object and the object's handle is updated to the current version number of the object.[1] This leads to a blocking implementation were the current transaction has to wait until the owning transaction finishes with the concurrent object before writing to the object. The implementation presented here is also lock-based. To read from an object, the transaction waits until the object's handle contains a version number and then logs the version number in a read descriptor. Unlike commit-time locking, the transaction locks the object when it is opened for writing, and the locks are released only at commit or abort time. This implies that the transaction uses a revocable two-phase locking scheme for writes and optimistic concurrency control for reads. Hence it is unnecessary to store the status of a transaction in the descriptor.

During commit time the transaction checks whether it is valid and reaches a decision on whether it should commit or abort. It is during commit time that the transaction makes its changes visible to other transactions. A transaction deems itself valid if all the objects it has read from are consistent—i. e., they have not been modified after the current transaction read them. The consistency of reads can be checked by verifying the version numbers in the read

[1] The current version number of an object is held in the object handle.

descriptor and the object's handle the descriptor points to. A transaction commits if it is valid, and makes its writes visible by copying across its working copy of the data to the written object's data. The transaction also sets the object's handle to a new version number. The transaction aborts if it is invalid (because it has seen inconsistent data). As an optimization, the runtime periodically checks for invalid transactions and aborts them. Otherwise, an infinite loop could result when a transaction is unable to commit due to having seen inconsistent data. In the event of a deadlock, one transaction will abort, releasing all its locks and reverting all its updates.

Ennals also places a restriction on the number of transactions that can be run at a given time. The number of transactions should not exceed the number of available cores. Consider the case where *N* transactions are running on *N* cores. If a new transaction needs to be started, it has to wait until a core is free. In this way the number of transactions executing can be minimized and all the available cores can be utilized fully. If the number of transactions exceeds the number of cores, a blocking behavior results, as is common in a lock-based setting. Hence this cannot occur in STMs that claim to be obstruction-free or non-blocking. This optimization argues that having fewer transactions will reduce the frequency of conflict between them. This in turn will provide a performance boost.

3.3. TL2

A *closed* TM system is one where memory that is used transactionally cannot be used non-transactionally and vice versa. Ennals' STM [7] and FSTM [5] are closed transactional memory systems. Non-blocking STMs, like FSTM, require a special transactional memory area separate from the non-transactional memory. A lock-based STM, like Ennals' STM, requires specialized `malloc()` and `free()` operations for the transactional memory. In both FSTM and Ennals' STM, inconsistent states and invalid transactions are contained in specialized managed runtime environments. These special requirements make it difficult to do "mechanical" code transformation from sequential or lock-based code into concurrent code. TL2 [6], the STM by Dice, Shalev and Shavit, attempts to address this problem by

employing *commit-time* locking (locks are acquired only when attempting to commit) and a global version-clock based validation scheme.

TL2 is a lock-based STM that employs commit-time locking. The TL2 algorithm requires a global version-clock. This global version-clock is incremented once by each transaction that writes to memory and read by all transactions. In TL2, a special versioned write-lock is associated with every transacted memory location. The versioned-write lock, put simply, is a single word spin-lock. The lock is acquired and released using atomic CAS (Compare & Swap) operations. One bit of the single-word spin-lock is used to indicate whether the lock is set or not, and the rest of the bits hold a version number. This version number is read from the global version-clock and incremented on each successful lock-release. A collection of versioned write-locks is allocated for data structures such as linked lists. As per the TL2 algorithm, only read-write transactions need to collect read-sets and write-sets; read-only transactions do not. Each read-write transaction thread has a private read-set and write-set. These are implemented as linked lists. Each read-set entry contains the address of the lock that covers the variable being read. Each write-set entry contains the address of the variable, the value to be written to the variable, and the address of its associated lock.

To perform a write operation to the shared memory, the transaction should load the current value of the global version-clock and store it in a thread-local variable called the read-version number (*rv*). Then the transaction code can begin executing speculatively in its private memory. A read-set of all the addresses read and a write-set of all the addresses to be written is maintained locally. To enable aborting or retrying of a transaction a logging functionality is implemented. This is accomplished by augmenting the loads with instructions that record the read address, and replacing stores with code recording the address and value to be written. The transactional load first checks the local write-set to see if the value is locally changed. If so, the updated value is loaded. A load instruction has to check whether the location's versioned write-lock is free at the time of the load. The load instruction checks to be sure that the lock bit is clear, and that the version number of the lock is $\leq rv$. If the lock's version number is greater than *rv*, it suggests that the location was modified after *rv* was read

by the current thread. In such a case the current thread has to abort because its read of the location should have occurred before the location was modified by another thread.

After the speculative execution, the transaction attempts to acquire all the locks it requires. Acquiring a lock involves setting the write-lock bit of the location. Failure to acquire the locks forces the transaction to abort. Once all the locks are acquired, the global version-clock is incremented. The incremented global version-clock value is recorded in a local write-version number variable *wv*. The read-set is validated next. It is valid if the write-lock associated with each location in the read-set is $\leq rv$. Also, for the transaction to see a consistent state, the lock bits of all the locations in the read-set must be in the cleared state.

If the validation fails, the transaction aborts. In the special case when $wv = rv + 1$, it is not required to validate the read-set, as only the current transaction would have modified the locations in the read-set. To commit the transaction, the new values for all the locations in the write-set are updated in the shared memory. The locks on these locations are released by setting the version value to *wv* and clearing the write-lock bit.

Read-only transactions need not even keep a local read-set. A read-only transaction samples the global version-clock and stores its value in a local read-version variable (*rv*) like a write transaction. The read-only transaction then proceeds with the speculative execution. If any of the locations read has the write-lock bit set or the lock's version field is $> rv$, the current read-only transaction aborts. If the transaction succeeds in reading all the locations it requires, it commits.

To prevent the global version-clock from being a contention bottleneck, the authors suggest augmenting the version number with the thread ID of the thread that last modified it. In this scenario, the thread increments the global version-clock only if it was the last one that modified it. In the other cases, it updates its thread ID but leaves the version-number field unchanged.

Chapter 4

PERFORMANCE METRICS

One goal of this thesis is to identify metrics that can effectively compare different transactional memory implementations. All existing STM implementations provide experimental results showing how fast their STM runs with respect to others by varying resource contention and scalability parameters. Execution speed, however, is just one dimension, one metric. The execution speed of an STM is typically measured in time taken for a CPU operation per μs for a processor in a multiprocessor/multicore setting. A typical case in point is TL2 [6] by Dice et.al. A red-black tree micro-benchmark is employed to substantiate the claim that "TL2 is ten-fold faster than a single lock." Although comparing STMs based on execution speed helps in assessing the usability of an STM, parameters such as memory usage and ease of programming are equally relevant. One of the major motivations behind transactional memory research is the difficulty in lock programming. It is essential for STMs to be easily programmable to achieve better user acceptance. This chapter introduces performance metrics employed for qualitatively and quantitatively comparing various software transactional memory implementations.

4.1. Experimental Setup

To compare the performance of the three STMs – FSTM, Ennals' STM and TL2 — tests were conducted on a number of test beds. A graphical representation of the test results is given later in this chapter. This section discusses the two experimental setups: IBM Blade Center Linux Cluster (*Henry2*) and Sun V880 NC BioGrid Node (*Henry8*). Both test beds are part of the high performance and grid computing initiative at North Carolina State University [11].

- IBM Blade Center Linux Cluster (Henry2)

The Henry2 cluster consists of more than 175 dual Xeon compute nodes with 2.8–3.2 GHz Intel Xeon processors (nominally 5.6–6.4 GFlops per processor), 4GB per node distributed memory, 36GB or 40GB of disk space per node plus shared storage (> 7TB total), and dual gigabit Ethernet interconnects. Henry2 runs GNU/Linux with a Linux 2.6.9-42.Elsmp kernel.

- Sun V880 NC BioGrid Node (Henry8)

Henry8 is a Sun Fire™ V880 shared-memory multiprocessor with 8 UltraSPARC-IIi® processors operating at 750MHz. It has 1 TB disk space and a gigabit Ethernet network connection. Henry8, which is in the North Carolina State University "realm," is part of the NC BioGrid. Henry8 runs Solaris™ 8.

All the performance experiments were conducted using a red-black tree micro-benchmark. Red-black trees are inherently non-blocking data structures, which makes them especially suited for lock-free programming [5]. The red-black tree micro-benchmark is a lock-free implementation of a red-black tree with early release. With early release, the transaction releases an object after writing it, despite pending updates on other objects by the transaction. A single sentinel node replaces all the NULL child pointers in the red-black tree implementation. (The sentinel node is essentially a node with a special key value other than NULL). This eliminates the need to keep track of multiple NULL child pointers. Although the sentinel node is a probable performance bottleneck, this problem can be avoided by releasing it early prior to committing.

4.2. Ease of Programming

The recent advances in technology have been focused on delivering multiple processing cores per chip (CMP – Chip Multi-Threading) rather than improving the performance for a single thread. With the advent of such technologies, programmers are forced to write programs that effectively exploit parallelism. A typical way to write parallel programs is by

using locks. Coarse-grained locking offers limited parallelism, whereas fine-grained lock programs are notoriously hard to write. The idea of transactional memory stems from this dilemma. From a user perspective, transactional memory is easier to program than locks and provide fine-grained parallelism [5].

It is beneficial to analyze the programming effort and complexity associated with different transactional memory systems. An STM that is easier to program and understand has an edge over another that is difficult to program. Quantifying the readability and understandability of programs has long been a challenge for software engineering practitioners. Börstler et.al [12] have proposed a comprehensive set of metrics for measuring the understandability of sample code. These understandability metrics give a good indication of ease of programming. Even though these metrics have been proposed for an object-oriented language like Java™, most of them are applicable to non-object-oriented languages as well. The three STMs used for performance comparison in this thesis are coded in C. Table 4.1. lists some of the metrics that can be used to measure program understandability.

Cyclomatic Complexity (CC) provides a quantitative measure of the logical complexity of the program [13]. This metric counts the number of independent paths in the basis set of a program and provides an upper bound on the number of tests that must be conducted to ensure that all statements in the program have been executed at least once. The *LoC* (Lines of Code) metric is computed by counting only the number of source code lines excluding comment lines and blank lines. A Code Counter tool [14] is used for this purpose. The cyclomatic complexity is calculated from a flow graph (a graph that depicts the logical control flow in a program, with each graph node representing a structured program construct) using the formula, $CC = P + 1$, where P is the number of predicate nodes contained in the flow graph. A sample CC calculation is shown in Figure A.1. A high value for *WMC* (Weighted Method Count) indicates poor program understandability.

Table 4.1. The metrics employed to measure program understandability.

Acronym	Description
LoC	Total lines of code.
$CC_{max}(m)$	Maximum cyclomatic complexity of methods (*m*).
CC / LoC	Average CC per LoC. Here CC is the sum of the cyclomatic complexities of all the methods.
LoC / m	Average LoC per method. Here *m* is the total number of methods in the program.
m / c	Average number of methods per class. In a non object-oriented language like C, the number of classes (*c*) is 1.
WMC	Weighted Method Count. The product of the three previous measures.

Each of the three STMs, FSTM, Ennals' STM, & TL2 have shipped the sample C implementation of a red-black tree benchmark with the STM source code. This program serves as a benchmark and as an example program for prospective STM users. The values for the metrics in Table 4.1. for the red-black tree benchmark of the three STMs are shown in Table 4.2.

Table 4.2. Ease of programming metrics for FSTM, Ennals' STM, and TL2.

Measure	FSTM	Ennals' STM	TL2
LoC	409	409	510
$CC_{max}(m)$	18	18	16
CC / LoC	0.132	0.132	0.303
LoC / m	51.125	51.125	18.888
m / c	8	8	27
WMC	54	54	155

FSTM and Ennals' STM post the same values for all the metrics. This is because Ennals has used the same red-black tree benchmark code written by Fraser [7] . From the results, TL2 has the highest *WMC* value and hence poor ease of programming.

4.3. Object-Acquire Semantics

This comparison criterion analyzes when a concurrent object is acquired for modification by a read-write transaction in an STM system.

In the lock-free FSTM, the object is acquired at commit time rather than at encounter time when the object is opened. This results in the transaction being visible to the concurrent system only at commit time. This approach of delaying object acquire until commit time is termed *lazy acquire*. Acquiring the object at encounter time is the *eager-acquire* strategy [15]. With eager acquire the conflicts with other transactions are detected early, and hence such conflicts can be resolved early. By contrast, lazy acquire has a small window of contention with other transactions. If the conflicting transaction were to be aborted anyway, due to late conflict detection, lazy acquire lets the current transaction run till completion. This can save the effort of aborting the current transaction first and retrying it later with eager acquire. From the empirical experiments conducted by the authors of FSTM [5], the lazy-

acquire strategy exhibits better performance over eager-acquire strategy. Comparing STMs using lazy and eager-acquire approaches is beyond the scope of this thesis because TL2 and Ennals' STM are lock-based. As explained in section 3.1, the acquire object operation is a multi-word CAS operation.

In TL2 and Ennals' STM, an object is acquired by locking it. Hence these are lock-based STMs. Ennals' STM employs encounter-time locking where the object is locked when it is opened for writing. The strategy used by TL2 is commit-time locking where the locking of the object is deferred until commit time. With encounter-time locking conflicts can be detected early. The current transaction or the conflicting transaction can abort depending on which one is of lower priority. It is very unlikely for a transaction to know, during encounter time, the entire set of objects it needs to access. In such a scenario, a transaction may have acquired all but one object and be forced to abort. With encounter-time locking the window of contention is longer than with commit-time locking. With commit-time locking the locks are acquired only after a speculative execution of the transaction. At this point the transaction knows the entire set of locations it needs to lock. Having a smaller window of contention implies a smaller number of transaction aborts and a smaller amount of sandboxing ill effects of invalid transactions. In this case the transaction needs to keep track of all the objects it needs to modify. This step is not required in encounter-time locking, as locks are acquired when the objects are encountered. Table 4.3 compares the two locking schemes.

Table 4.3. Comparison of encounter-time locking vs. commit-time locking.

Encounter-time locking	Commit-time locking
The objects are acquired when opened for writing	The objects are acquired only during transaction commit
No speculative execution necessary	Speculative execution needed to determine all the objects that need to be acquired at commit time.
Locks held for a longer time	Locks held for a shorter time
Larger window of contention	Smaller window of contention

Consider transaction *A*, which needs to write to *N* objects for successful completion. Transaction *B* needs to access *M* objects, of which one object is also used by *A* (let that object be *x*). Transaction *B* has a higher priority than transaction *A*. Consider an encounter-time locking scenario. Transaction *A* starts and begins acquiring locks on objects. In the meantime, transaction *B* starts and acquires a lock on object *x*. Even after acquiring locks on the required *N*–1 objects, transaction *A* would still have to abort, since transaction *B* is of higher priority. On the other hand, with commit-time locking, transaction *B* would have time to finish its execution and release its lock on object *x* before transaction *A* even attempts to acquire a lock on object *x*. This scenario clearly favors commit-time locking although this scenario does not happen very frequently.

The execution speedup obtained with commit-time locking [6] is at the expense of using more memory to store per-transaction information. This approach could be advantageous as long as the transaction sizes are small.

4.4. Metadata Organization

The storage location of concurrent object metadata is an important design decision for an STM implementation. This determines whether the commit operation is lightweight or not.

In FSTM, the object metadata (object header) and the data block is placed inside the transaction descriptor of the transaction that owns the object (per transaction metadata). This is the case for read-write as well as read-only transactions. Such a scenario is illustrated in Figure 3.2. While opening an object for reading or writing there is a level of indirection from the object header to the data block to access the object data (Figure 3.1). This indirection will adversely affect the execution time of transactions—especially read-only transactions—if the object header and data block cannot be loaded in the same cache line. This results in multiple cache lines being loaded for a single object load. During commit time, both the object header and the data block of all the acquired objects need to be modified. The commit operation in FSTM is a multi-word CAS operation. If the transaction has acquired N objects for modification, $2N+1$ CAS operations are required for a successful transaction commit – N for updating object headers, N for updating the data blocks with their new values and 1 for changing the status of the transaction from ACTIVE to COMMITTED.

In Ennals' STM object metadata (object header) and object data are stored next to each other. Hence when the object is opened for reading or writing, no level of indirection is required to access the object data from the metadata (Figure 3.3). The object metadata and data is loaded in the same cache line. Ennals' STM also employs per-transaction metadata storage. The transaction acquires an object by obtaining an exclusive lock on the object when it is opened for reading or writing. For each acquired object, the transaction stores the object data and a pointer to the object header. During commit time, the object data should be updated with the new value and the object header should be updated with the new version number to indicate the release of the lock. For a transaction that has acquired N objects, the commit operation will take $2N+1$ CAS operations to complete — N for modifying object data, N for modifying object header and 1 for changing the transaction state to COMMITTED.

In TL2, as in Ennals' STM, the metadata is stored next to the object data. Hence both the object data and metadata are loaded in the same cache line. For conflict detection the TL2 algorithm uses a global version-clock. To write to a concurrent object, the transaction acquires an exclusive lock on the object during commit time. For a successful commit

operation, the transaction should increment the global version-clock value, update the object data with the new value, release the lock by updating the write-version number of the object and clear the lock bit. For a transaction that has acquired *N* objects, the commit operation takes 2*N*+1 CAS operations – *N* for updating the object data with the new value, *N* for updating the write-version number of the object and 1 for incrementing the global version clock.

Thus, all the three STMs use the same number of CAS operations for committing, if the number of acquired objects are the same.

4.5. Transaction Validation

This section discusses the strategy employed by STMs to detect invalid transactions. Invalid transactions are transactions that have seen inconsistent data and hence should be aborted. Invalid transactions are also referred to as *zombie* transactions ("zombies").

In FSTM, every transaction validates during commit time to check whether the values it read have been modified by any other transaction. If any value read by the current transaction was modified another transaction, the current transaction has to abort since it is reading stale data. Here, checking for validity of a transaction is deferred until commit time. This results in longer-running zombie transactions that may abort other valid transactions, ultimately leading to a livelock. When the zombie transaction is detected it is aborted by the STM system without user intervention.

The transaction checks if it is valid prior to committing in Ennals' STM. This is done by validating that no value it read has been written to by any other transaction. The author of Ennals' STM recognized the problem with long-running zombie transactions. In addition to validating transactions at commit time, the STM system periodically checks for invalid transactions. In the current implementation of Ennals' STM takes only fixed integer values for validity-check intervals. The programmer who has access to the Ennals' STM source code can set these intervals. Having an interval that is small is an overkill as it might increase

the execution time of the STM. Again, letting the zombie transaction run longer might result in starvation of a transaction. Starvation occurs when the transaction has read all the locations it needs to read, has read some inconsistent data, and cannot write to any location because it has read inconsistent data.

In TL2, the validity of the values read by the transaction (read set) is checked each time the transaction opens an object for reading or writing. In addition to this, the validity of the values read by the transaction is checked during commit time. This combination of validity checking at open-time and commit-time helps detect zombie transactions early. TL2 employs commit-time locking. Due to validating the transaction's read set before the commit phase starts, the transaction can be aborted if it is invalid, before it even starts acquiring any locks. This eliminates the need to release all the locks owned by the transaction during its abort. At the same time validating the transaction for each object open and commit can increase the transaction execution time. Due to the small window of contention during commit time, this might not have a pronounced effect on the execution time.

4.6. Contention Management Strategy

Contention management strategy refers to the policy employed to resolve conflicts between transactions over the concurrent object.

FSTM is a lock-free STM. Recursive helping is the contention management strategy used by FSTM. If the current transaction detects a conflict with a COMMITTED transaction, it has to abort. If it detects a conflict with an UNDECIDED or ABORTED transaction it traverses the read-write list of the conflicting transaction to determine the current version of the object. If the conflicting transaction's object version differs from the current transaction, either the current transaction aborts or recursively helps the conflicting transaction. If the conflicting transaction precedes the current transaction in the global order of transactions (section 3.1), the current transaction recursively helps the conflicting transaction. If the current transaction precedes the conflicting transaction, the conflicting transaction is aborted by the current transaction. For example, if an operation in transaction A is obstructed by another operation

in transaction B and transaction B precedes transaction A in the global order of transactions, A will help B complete its work. This implemented by recursively reentering the operation and passing the invocation parameters specified by B's operation. B's operation in this scenario is responsible for making available sufficient information to let the conflicting processes determine its invocation parameters. When the recursive call is completed, the obstruction is removed and A's operation can continue its progress. According to the FSTM algorithm the conflicts are detected only at commit time.

Ennals' STM is a lock-based STM. Here the conflicts occur when trying to acquire locks when some other transaction has locked the object. The current transaction spins for a fixed number of times or till the owner lets go of the lock. In the event of a deadlock, the transaction with the lowest priority aborts. Determining transaction priorities was explained in section 3.2. If the transaction fails after the fixed number of attempts, it either aborts itself or the conflicting transaction depending on the transaction priorities. The objects are acquired at encounter time, and hence conflict detection does not wait till commit time.

TL2 employs the same conflict detection strategy as Ennals' except that it tries to acquire locks only during commit time. Deferring the conflict detection to commit time reduces the window of contention. Empirical evaluations in section 4.10 show the effects of contention management strategies on the STM's throughput.

4.7. Lock-Acquire Semantics

This STM comparison criterion is exclusive to lock-based STMs like TL2 and Ennals' STM. A comparison of the encounter-time locking scheme used by Ennals' STM and commit-time locking strategy used by TL2 is given in section 4.3.

4.8. Storage Reclamation

All three STMs – FSTM, Ennals' STM and TL2 – compared here are implemented in C. These implementations do not make any unreasonable assumptions about the underlying

hardware like the availability of specialized CAS instructions [5-7]. In STM systems, transaction aborts require some degree of automatic storage reclamation. For example, assume that transaction *A*, which owns object *x*, is explicitly aborted by transaction *B*. Before transaction *B* acquires object *x* it has to make sure that transaction *A* has aborted and relinquished object *x*. In languages like C, which permit explicit storage management, either STMs can provide specialized garbage-collection schemes or leave the job of freeing memory to the programmer.

FSTM and Ennals' STM use the same specialized garbage-collection technique implemented by Fraser [5]. FSTM has a very high rate of heap allocations and garbage creation due to allocating a new version of the object each time it is updated. This necessitates a customized garbage-collection scheme. Each FSTM transaction descriptor is an aggregate of embedded object handles that are sequentially allocated. The embedded objects are allocated sequentially within the aggregate and are not used or reclaimed except as part of the aggregate. A reference-counting scheme is used to garbage-collect the FSTM transaction descriptor. The FSTM descriptor contains a reference count that indicates how many processes hold a reference to it. Each process that operates on the descriptor increments the reference count once. A process is responsible for all decrementing all the shared reference counts introduced as part of the operation. Even though reference counting introduces the overhead of incrementing and decrementing reference counts, the descriptor can be reused as soon as the last reference to it is removed. Since the descriptors are ephemeral, it is highly unlikely that many processes will ever access them. Hence there is never a need for a large number of reference-count manipulations. The reference-counting scheme is integrated with the FSTM source code.

For all the other FSTM objects like object headers and data blocks, an epoch-based reclamation scheme is used. There is a separate garbage-collection module in FSTM that does this. In this scheme, each object, when not referenced from the shared heap, is explicitly added to the garbage list. Once an object becomes garbage, no new shared references can be created to it. The author introduces a global epoch count to determine when no stale

references exist to an object. Each process, when it accesses shared memory, observes the current epoch. Whenever a process starts a shared-memory operation, it additionally checks the process list to see if all the processes currently executing in the shared memory region have observed the current epoch. In that case, the process frees the contents of the garbage list that was created two epochs back and increments the epoch count. At a given time, only three garbage lists need to be maintained – for the current epoch (e), the epoch (e–1) and the epoch (e–2).

For its per-object implementation, TL2 relies on the programmer to free the memory. It does not use any special garbage-collection schemes [6]. The authors admit that the act of acquiring a lock cannot be guaranteed to take place when the object is alive. They also suggest repeatedly validating the entire transaction before updating each location in the write-set to address the problem of inaccessible locked objects.

4.9. Search Overhead

This STM comparison criterion explores the search overhead incurred in locating the current version of the concurrent object when a conflict is detected between two transactions over that object.

In FSTM, if the current transaction detects a conflict with an ABORTED or UNDECIDED transaction, the current transaction has to traverse the read-write list of the conflicting transaction to detect the current version of the object. If the conflict is detected with a COMMITTED transaction, the current transaction has to abort. The time spent on searching the read-write list can impact performance depending on the size of the list. Read-write lists are implemented as linked lists in FSTM.

In Ennals' STM and TL2 this search overhead is absent. When a conflict is detected with an active transaction, the current transaction waits for a random interval and tries again. The current transaction tries for a fixed number of times or till the owner relinquishes the lock to acquire the lock. If it fails, it either aborts itself or the conflicting transaction depending on

the transaction priorities. If the conflict is detected with a committed transaction, the current transaction aborts.

4.10. Execution Time

The execution time (throughput) is measured as CPU time per μsecond for each successful read/write operation on the red-black tree benchmark. The ratio of reads to writes is set to 3:1 for all the experiments conducted. Tests were conducted on the Henry2 (section 4.1.) cluster for FSTM and Ennals' STM. The TL2 STM is currently implemented to run only on the Solaris™ platform. The execution times of all the three STMs – TL2, FSTM and Ennals' STM – were measured on Henry8 (section 4.1). Experiments were conducted for low-contention, high-contention and varying-contention scenarios. The mean data set size of the red-black tree for the low-contention setting is 524288(2^{19}) and for the high-contention setting, it is 16 (2^4). For the varying-contention setting the mean data set size was varied from 16 (2^4) to 524288 (2^{19}). The graphs of the empirical evaluation and analysis of the results is presented in this section.

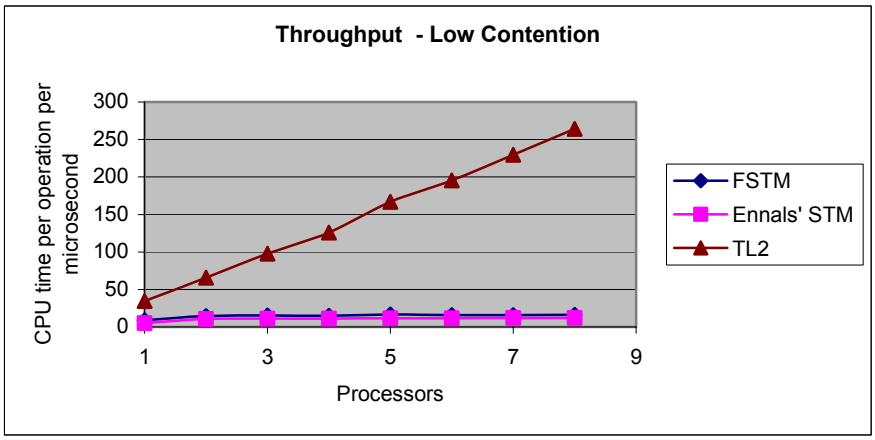

Figure 4.1. Execution times for the three STMs measured on Henry8 with a mean data size of 2^{19} for a red-black tree.

Figure 4.1 shows the throughput of the three STMs under low contention for a large red-black tree using 1 to 8 processors on Henry8. FSTM and Ennals' STM clock similar execution times, whereas TL2 exhibits a very poor throughput. For FSTM, less contention implies less recursive helping, which results in a faster execution time. With Ennals' STM the locks are acquired early, there is no speculative execution involved, and there is less contention. This results in better throughput. For TL2, there is an overhead of maintaining the global version-clock and performing speculative execution. Chances of more threads contending for locks are increased as all of them try to acquire locks at commit time. This explains the poor performance of TL2 relative to FSTM and Ennals' STM.

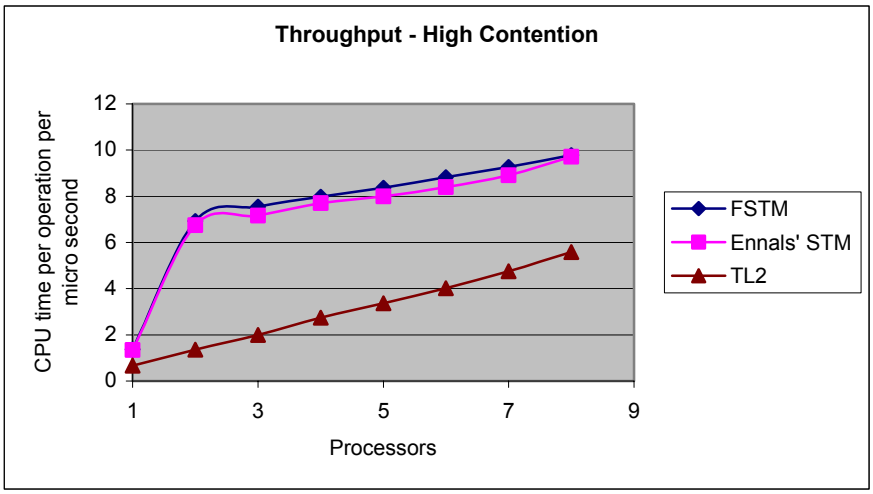

Figure 4.2. Execution times for the three STMs measured on Henry8 with a mean data size of 2^4 for a red-black tree.

Figure 4.2 shows the throughput of the three STMs under high contention for a small red-black tree using 1 to 8 processors on Henry8. TL2 clocks substantially better execution times than FSTM and Ennals' STM. Ennals' STM performs slightly better than FSTM. Due to high contention, FSTM resorts to more recursive helping to resolve conflicts. This increases the execution time. For Ennals' STM the window of contention is more due to encounter-time locking. A larger window of contention leads to more conflicts and hence more aborts and retries. In the high-contention setting, the commit-time locking strategy of TL2 is a winner as the window of contention is small.

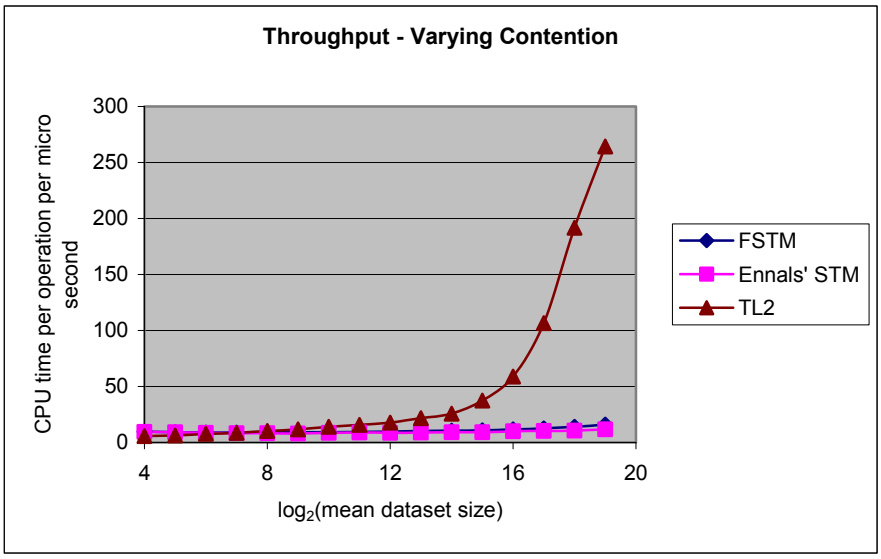

Figure 4.3. Execution times for the three STMs measured on Henry8 using 8 processors for a red-black tree.

Figure 4.3 shows the throughput of the three STMs under varying contention (mean dataset size from 2^4 to 2^{19}) for the red-black tree benchmark using 8 processors on Henry8. From the figure it can be inferred that TL2 performs better under high contention. Its throughput deteriorates as the amount of contention diminishes. Ennals' STM and FSTM perform consistently on high and low-contention settings. Figure 4.3 underlines the findings of Figure 4.1 and Figure 4.2.

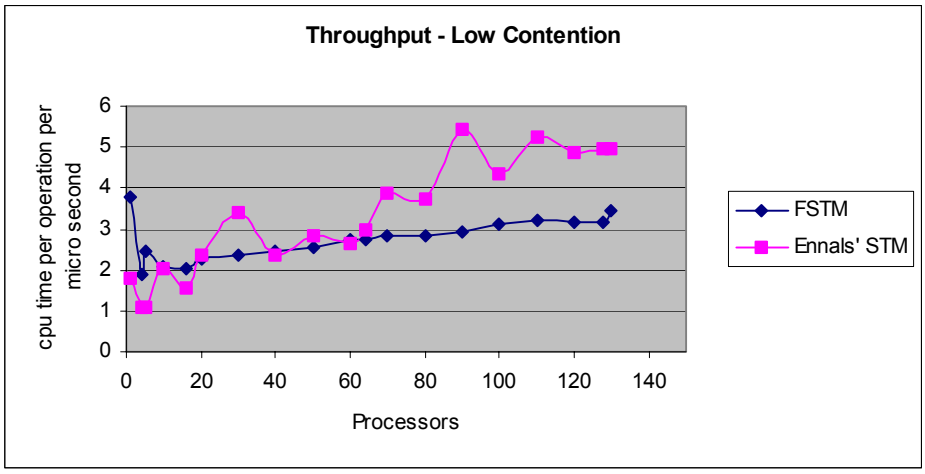

Figure 4.4. Execution times for FSTM and Ennals' STM measured on Henry2 with a mean data size of 2^4 for a red-black tree.

Figure 4.4 shows the throughput of FSTM and Ennals' STM under low contention for a large red-black tree using 1 to 130 processors on the Henry2 cluster. The red-black tree micro-benchmark employed here is a lock-free implementation. FSTM, being lock free, is not affected by the locking and cache-coherence protocols of a cluster. This explains the no-spike line for FSTM. Ennals' STM, being lock-based, on the other hand is affected by the timing dependencies on acquiring locks. Ennals' STM backs off (exponential or random) and spins till the object is available in the event of contention. The timing dependence of the contention effects explains the wavy graph, which is similar to the ones for spin-locks on clusters [16]. FSTM shows consistency and has better throughput when running on a loosely coupled system like a cluster.

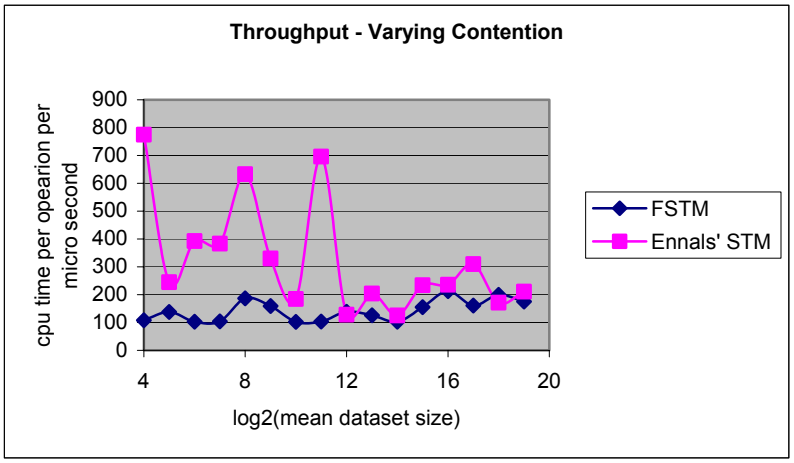

Figure 4.5. Execution times for FSTM and Ennals' STM measured on Henry2 using 64 processors for a red-black tree.

Figure 4.5 shows the throughput of Ennals' STM and FSTM under varying contention (mean dataset size from 2^4 to 2^{19}) for the red-black tree benchmark using 64 processors on the Henry2 cluster. In a cluster-like system, numbers of processors that are powers to 2 yield better throughput than the ones that are not. For this reason number of processors is set to 64 for the varying-contention benchmark. As expected, FSTM, being lock free, is less affected by the underlying cluster architecture. In Ennals' STM, the peaks and valleys are the result of time delays to acquire locks [16]. High contention results in more waits for exclusive locks on objects. This explains the peaks being higher as the contention increases for Ennals' STM. The exact nature of the wavy graph is cluster environment dependent.

From the results presented in this section it can be seen that lock-free STM are more suitable for loosely coupled systems like clusters than lock-based ones. The results also show that with a shared-memory multiprocessor system like Sun Fire V880, lock-based STMs have better throughput.

4.11. Memory Usage

The performance criterion of memory usage measures the amount of virtual memory used for one successful run of the red-black tree benchmark. The high-memory watermark[2] quantifies the amount of virtual memory used for each run of the red-black tree micro-benchmark. This value is obtained using a processor-statistics reporting utility (`prstat` for Henry8). The virtual-memory usage statistics are collected every second, and the maximum value is used as the high-memory watermark. The ratio of reads to writes is set to 3:1 for all the experiments conducted. The execution times of all the three STMs – TL2, FSTM and Ennals' STM – were measured on Henry8 (section 4.1.). As in section 4.10, experiments were conducted for low-contention, high-contention and varying-contention scenarios. The graphs of the empirical evaluation and analysis of the results is presented in this section.

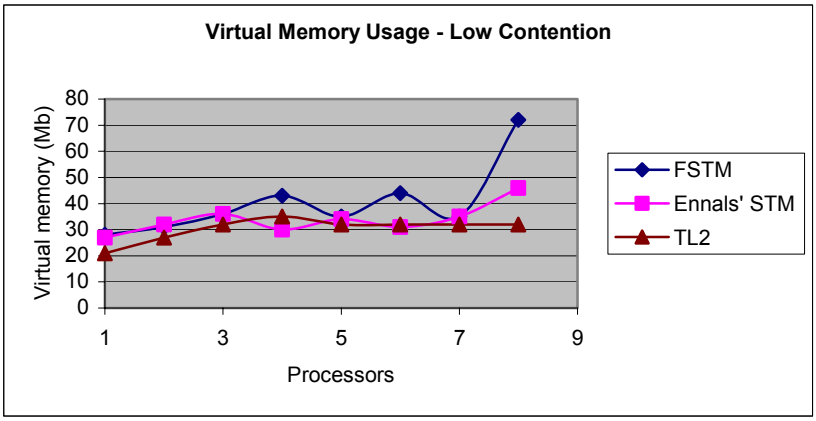

Figure 4.6. Virtual memory usage for the three STMs measured on Henry8 with a mean data size of 2^{19} for a red-black tree.

[2] High memory watermark refers to the maximum amount of memory used.

Figure 4.6 shows the amount of virtual memory used by each of the three STMs under low contention for a large red-black tree using 1 to 8 processors on Henry8. The lock-based STMs – TL2 and Ennals' STM – have better cache locality due to no indirection being required to access object data from the object header. FSTM, being lock free, requires at least one level of indirection to access object data from the object header. These pointers result in a larger memory usage compared to the other two. In the red-black tree benchmark, each thread creates red-black tree objects equal to the *floor* of the maximum key range divided by the number of threads. This means that in situations where the number of objects is not evenly divisible by the number of threads slightly fewer objects will be created. In all the experimental runs the number of processors equals number of threads and each thread is bound to a different processor. The additional number of tree objects created for evenly divisible number of processors explains the spikes in the FSTM line. The effect is observed when the data set sizes are large. TL2 and Ennals' STM report comparable memory usage due to their lock-based nature. The effect due to additional objects being created for even number of processors is not substantial in lock-based STMs. This is attributed to the smaller object sizes as a result of there being zero levels of indirection.

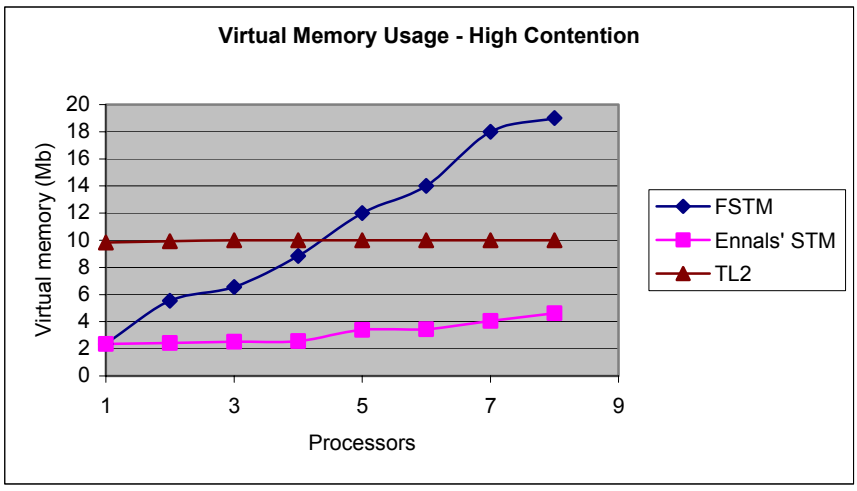

Figure 4.7. Virtual memory usage for the three STMs measured on Henry8 with a mean data size of 2^4 for a red-black tree.

Figure 4.7 shows the amount of virtual memory used by each of the three STMs under high contention for a large red-black tree using 1 to 8 processors on Henry8. As before, TL2 and Ennals' STM, being lock based, report a similar memory-usage pattern. FSTM, being lock based, requires maintaining more per thread information. When the data set size is small, the memory required to maintain per thread information is relevant. The nearly linear curve for FSTM is explained by the almost constant amount of extra memory used when creating a new thread and binding it to a processor. The per-thread information maintained by lock-based STMs is much smaller compared to the lock-free FSTM. This results in an almost constant memory usage when data-set sizes are small. TL2 uses a global version-clock and maintains data to synchronize it. This causes TL2 to use more memory than Ennals' STM.

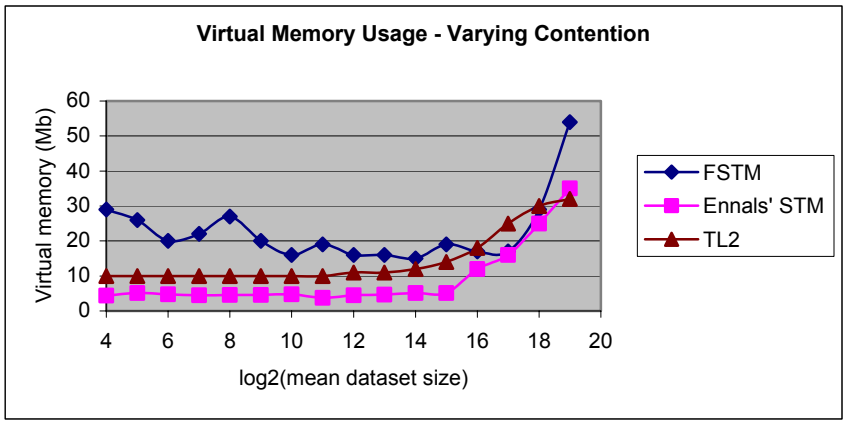

Figure 4.8. Virtual memory usage for the three STMs measured on Henry8 using 8 processors for a red-black tree.

Figure 4.8 shows the amount of virtual memory used by each of the three STMs under varying contention (mean dataset size from 2^4 to 2^{19}) for the red-black tree benchmark using 8 processors on Henry8. As the mean data-set size increases, the amount of virtual memory used also increases. Due to better cache locality the lock-based STMs – TL2 and Ennals' STM – use less memory than the lock-free FSTM. The spikes in the FSTM curve could be due to slightly more aborts and retries involved. The garbage list structures maintained for the specialized garbage collection scheme of FSTM also use more memory.

The empirical evaluations presented in this section conclude that lock-based STMs use less memory than lock-free STMs due to better cache locality.

Chapter 5

CONCLUSION

Software Transactional Memory (STM) is a promising alternative to lock-based mutual exclusion strategies. This thesis presents performance comparisons based on memory, indirection and compute overheads of three different STM implementations – a non-blocking STM due to Fraser (FSTM), a lock-based STM due to Ennals, and a lock-based STM (TL2) with global version-clock validation due to Dice et.al. A red-black tree micro-benchmark is employed for all the quantitative comparisons in this thesis. The empirical evaluations done as part of this thesis suggest that:

- ✓ Ennals' STM has an edge over TL2 and FSTM, as it performs consistently well on low and high contention settings. Ennals' STM being lock-based, does not resort to expensive strategies like recursive helping (FSTM) to resolve conflicts. In low-contention settings, the encounter-time locking strategy of Ennals' STM is a winner over the commit-time locking strategy of TL2.

- ✓ Lock-based STMs have better cache locality than lock-free STMs because they use less memory.

- ✓ Lock-free STMs are more suited for loosely coupled systems like clusters than lock-based ones as lock-based STMs are affected by timing dependencies to acquire locks. (See figure 4.4.)

- ✓ Choice of contention management strategies significantly affects STM throughput and memory usage. (See section 4.6.)

From this it can be concluded that it is hard to build a general-purpose STM that performs consistently well with varying contention and architecture. The environment where the STM

is to be deployed and the constraints imposed by it have to be analyzed critically during the STM design phase.

This thesis has begun the process of comparing STM implementations based on performance. This is because performance is important and in fact, it is the motivation for creating STMs in the first place. The evaluation of performance is an important task. We hope that this work will set the stage for more detailed performance comparisons to come.

LIST OF REFERENCES

[1] H. Garcia-Molina, J. D. Ullman and J. Widom, *Database Systems: The Complete Book.* Prentice Hall, 2002.

[2] M. Herlihy, J. Eliot and B. Moss, "Transactional Memory: Architectural Support For Lock-free Data Structures," *Proceedings of the 20th Annual International Symposium on Computer Architecture,* pp. 289-300, 1993.

[3] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing,* vol. 10, pp. 99-116, 1997.

[4] R. Rajwar and M. Hill, "Transactional Memory Online", http://www.cs.wisc.edu/trans-memory.

[5] K. Fraser, "Practical Lock-Freedom." , PhD Thesis, *University of Cambridge Computer Laboratory,* 2004.

[6] D. Dice, O. Shalev and N. Shavit, "Transactional locking II," *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*. pp. 204-213.

[7] R. Ennals, "Software transactional memory should not be obstruction-free," Technical Report Nr. IRC-TR-06-052. *Intel Research Cambridge Tech Report.,* 2006.

[8] M. Herlihy, V. Luchangco, M. Moir and W. N. Scherer III, "Software transactional memory for dynamic-sized data structures," *Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing,* pp. 92-101, 2003.

[9] V. J. Marathe, W. N. Scherer III and M. L. Scott, "Adaptive Software Transactional Memory," *Proceedings of the Nineteenth International Symposium on Distributed Computing,* pp. 354-368, 2005.

[10] M. Herlihy, V. Luchangco and M. Moir, "A flexible framework for implementing software transactional memory," *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications,* pp. 253-262, 2006.

[11] "High Performance and Grid Computing (N C State)", http://hpc.ncsu.edu/.

[12] J. Börstler, M. E. Caspersen and M. Nordström, "Beauty and the beast toward a measurement framework for example program quality," submitted to OOPSLA Educators' Symposium, 2007.

[13] R. S. Pressman, *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 2005.

[14] "Code Counter 1.32", http://www.geronesoft.com/.

[15] V. J. Marathe and M. L. Scott, "A Qualitative Survey of Modern Software Transactional Memory Systems", Technical Report Nr. TR 839, *University of Rochester Computer Science Dept.*, 2004.

[16] D. E. Culler and J. P. Singh, *Parallel Computer Architecture.* Morgan Kaufmann Publishers San Francisco, 1999, pp. 341 ff.

APPENDIX

Table A.1. Benchmarks used in Shavit and Touitou's STM

Counting	A single shared counter is incremented by multiple processes a fixed number of times.
Resource Allocation	A set of processes shares a common pool of resources. A process tries to acquire a fixed-size subset of the resources from time to time.
Priority Queue	Multiple processes enqueue random values and dequeue the greatest value from a sequential heap implementation of a fixed-size priority queue. The number of enqueue and dequeue operations are pre-determined.
Doubly-Linked Queue	A fixed-size queue implementation as a doubly linked list in an array. Items are enqueued at the tail and dequeued at the head. Each of a set of processes performs a fixed number of enqueues and dequeues.

Table A.2. Benchmarks used in Herlihy's DSTM

Simple Locking	A simple linked list synchronized with a single lock.
IntSetSimple	A simple transactional integer set.
IntSetRelease	A simple transactional integer set implemented with early release.
RBTree	A non-blocking red-black tree implementation where nodes are opened in read mode and upgraded to write mode as required.

Table A.3. Benchmarks used in Fraser's FSTM

Skip List	A lock-free implementation of the skip list data structure with sentinel nodes for minimal and maximal key values.
Red Black Tree	A lock-free red-black tree implementation where nodes are opened in read mode and upgraded to write mode as required. A sentinel node replaces all the NULL child pointers.

Table A.4. Benchmarks used in Marathe et al's ASTM

IntSet	A sorted list of integers ranging between 0 and 255 for increased contention.
LFUCache	A priority queue heap to simulation of the cache replacement in an HTTP web proxy the least-frequently used (LFU) algorithm.
IntSetRelease	A highly concurrent IntSet variant in which transactions open objects in read mode and release them, early, while moving down the list.
RBTree	A concurrent red-black tree implementation.
RandomGraphList	A random undirected graph represented as a sorted linked list in which each node points to a separate sorted neighbor list.

Table A.5. Benchmarks used in Herlihy et al.'s DSTM2

List	A simple sorted linked list implementation with provision for random inserts, removals and searches.
Skip List	A skip list data structure implementation with provision for random inserts, removals and searches.

Table A.6. Benchmarks used in Ennals' STM

Skip List	An implementation of the skip list data structure with sentinel nodes for minimal and maximal key values.
Red Black Tree	A red-black tree implementation where nodes are opened in read mode and upgraded to write mode as required. A sentinel node replaces all the NULL child pointers.

Table A.7. Benchmarks used in Dice et.al's TL2

Red Black Tree	A concurrent version of the java.util.TreeMap sequential red-black tree implementation.

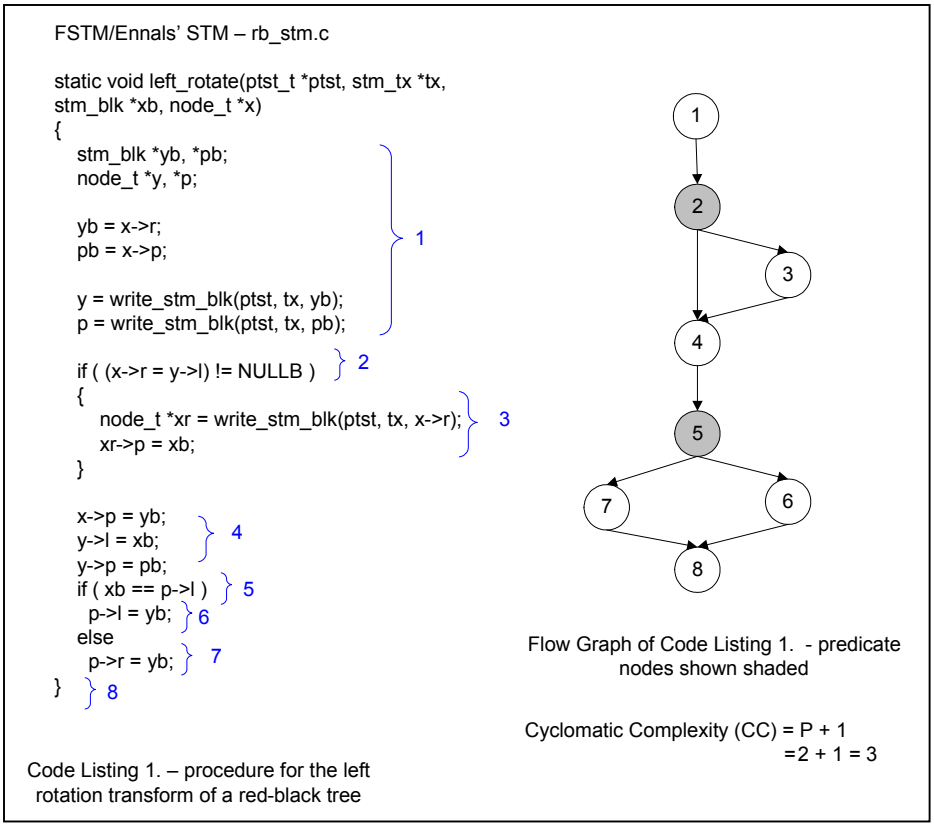

Figure A.1. Sample cyclomatic complexity calculation