Master Thesis
Computer Science
Thesis no: MCS-2010-28
May 2010

Performance Tradeoffs in Software Transactional Memory

Gulfam Abbas
Naveed Asif

School of Computing
Blekinge Institute of Technology
Sweden

Contact Information:
Author(s):

Gulfam Abbas
Address: Älgbacken 4:081, 372 34 Ronneby, Sweden.
E-mail: rjgulfam@hotmail.com

Naveed Asif
Address: c/o Gulfam Abbas, Älgbacken 4:081, 372 34 Ronneby, Sweden .
E-mail: naveed_asif77@hotmail.com

University advisor(s):
Professor Dr. Håkan Grahn
School of Computing
Blekinge Institute of Technology, Sweden

School of Computing
Blekinge Institute of Technology
SE – 371 79 Karlskrona
Sweden

Internet	: www.bth.se/com
Phone	: +46 455 38 50 00
Fax	: +46 455 38 50 53

ABSTRACT

Transactional memory (TM), a new programming paradigm, is one of the latest approaches to write programs for next generation multicore and multiprocessor systems. TM is an alternative to lock-based programming. It is a promising solution to a hefty and mounting problem that programmers are facing in developing programs for Chip Multi-Processor (CMP) architectures by simplifying synchronization to shared data structures in a way that is scalable and compos-able. Software Transactional Memory (STM) a full software approach of TM systems can be defined as non-blocking synchronization mechanism where sequential objects are automatically converted into concurrent objects.

In this thesis, we present performance comparison of four different STM implementations – RSTM of V. J. Marathe, et al., TL2 of D. Dice, et al., TinySTM of P. Felber, et al. and SwissTM of A. Dragojevic, et al. It helps us in deep understanding of potential tradeoffs involved. It further helps us in assessing, what are the design choices and configuration parameters that may provide better ways to build better and efficient STMs. In particular, suitability of an STM is analyzed against another STM. A literature study is carried out to sort out STM implementations for experimentation. An experiment is performed to measure performance tradeoffs between these STM implementations.

The empirical evaluations done as part of this thesis conclude that SwissTM has significantly higher throughput than state-of-the-art STM implementations, namely RSTM, TL2, and TinySTM, as it outperforms consistently well while measuring execution time and aborts per commit parameters on STAMP benchmarks. The results taken in transaction retry rate measurements show that the performance of TL2 is better than RSTM, TinySTM and SwissTM.

Keywords: Multiprocessor, Concurrent Programming, Synchronization, Software Transactional Memory, Performance

ACKNOWLEDGEMENTS

In the Name of Allah who is The Most Merciful and Beneficent

Prophet Mohammad (*Peace Be Upon Him*) said:

"Seek knowledge from the cradle to the grave"

We would like to congratulate and extend our gratitude for Professor Dr. Håkan Grahn on effectively guiding us in the achievement of this critical milestone. The successful completion of this research work was not likelihood without Professor Dr. Håkan Grahn consistent and valuable support.

The unconditional love, prayers, and sacrifices that our parents always gifted us are worthwhile mentioning. It would not have been possible for us to achieve this big success without their pure support.

In general we also would like to pay are special thanks to all our friends for their direct and indirect support which motivated us during all the times.

At last but not least we dedicate our degree to the great nation of ISLAMIC REPUBLIC OF PAKISTAN, the land which gave us identity, prestige, honor and the will to learn.

CONTENTS

LIST OF ACRONYMS

ApC	Aborts per Commit
CMP	Symmetric Multiprocessor
GV	Global Version
HTM	Hardware Transactional Memory
HyTM	Hybrid Transactional Memory
NIDS	Network Intrusion Detection System
OREC	Ownership Records
RSTM	Rochester Software Transactional Memory
SMP	Symmetric Multiprocessor
SSCA2	Scalable Synthetic Compact Applications 2
STAMP	Stanford Transactional Applications for Multi-Processing
STM	Software Transactional Memory
TL2	Transactional Locking II
TM	Transactional Memory
YADA	Yet Another Delaunay Application

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

Today we are living in an age of multicore and multiprocessor systems where the world is moving from single processor architectures towards multicore processors. There is an ever growing enhancement and development in processing power of CPUs and parallel applications that are being built to give end users more processing capabilities to accomplish complex and protracted jobs easily and rapidly. High performing and flexible parallel programming is the only means of utilizing the full power of multicore processors. Parallel programming has proven to be far more difficult than sequential programming.

Parallel programming poses many new challenges to mainstream parallel software developers, one of which is synchronizing simultaneous accesses to shared memory by multiple threads. Composing scalable parallel software using the conventional lock-based approaches is complicated and full of drawbacks [1]. Locks are either error prone (if fine-grained) or not scalable (if coarse-grained) and undergo a variety of problems like deadlocks, convoying, priority inversion and inefficient fault tolerance. One solution for such kind of problems is a lock-free parallel processing system which supports scalability and robustness [2, 3].

For decades in the database community, transactions offer a proven abstraction mechanism of dealing with concurrent computations [3]. Transactions do not suffer from locking drawbacks and take a concrete step towards making parallel programming easier [1]. Incorporating transactions into the parallel programming model builds a new concurrency control paradigm for future multicore systems named Transactional Memory (TM). A TM system executes code sequences atomically which allows application threads to operate on shared memory through transactions.

Transactions are a sequence of memory operations that either executes completely (commits) or has no effect (aborts). TM tries to simplify the development of parallel applications as compared to traditional lock-based programming techniques. TM is of three kinds [3], Hardware Transactional Memory (HTM), Software Transactional Memory (STM) and Hybrid Transactional Memory (HyTM). The first HTM idea was introduced in 1993 [4] and then in 1995 [5] STM was proposed to extend this idea. HyTM [6] is a combination of both hardware and software transactional memory. These pioneering works have paved the way for the development of many different versatile versions and extensions of hardware, software and hybrid TM implementations.

We focus here on STM which is a software system that implements nondurable transactions along with ACI (failure atomicity, consistency, and isolation) properties for threads manipulating shared data. The performance of recent STM systems has reached up to a level where these systems have gained an acme that makes them a reasonable vehicle for experimentation and prototyping. However, it is not clear how minimal the overhead of STM can reach without hardware support [3]. In trying to understand the performance tradeoffs of an STM in our thesis project we consider the key design aspects of four different STM implementations. They are

- RSTM [7] – a non-blocking (obstruction-freedom) STM,
- TL2 [8] – a lock-based STM with global version-clock validation,
- TinySTM [9] – a lock-based STM, and
- SwissTM [10] – a lock-based STM for mixed workloads.

Design and implementation differences in TM systems can affect a system's programming model and performance [3]. To compare the performance of these state-of-the-art STM implementations we use the STAMP benchmark suite [11], a collection of realistic medium-scale workloads. It currently consists of eight different applications and ten workloads as they inherently exploit the level of concurrency of the underlying STM implementation. Nearly all benchmarks measure the effectiveness of an STM as CPU time by varying contention and scalability parameters.

Thesis Outline

In this section we present the structure of the thesis. A brief introduction of each chapter is discussed here.

Chapter 1 (Problem Definition) provides us more detail about the problem and objectives of the study. It comprises of Problem Definition, Problem Focus, Aims and objective, and Research Questions.

Chapter 2 (Background) presents the background material and related works in the areas of parallel programming and transactional memory. This chapter discusses briefly Single Chip Parallel Computers, Database System and Transactions vs. Locks. This chapter also describes Transactional Memory and its different types. Finally, this chapter introduces the design alternatives and tradeoffs made by designers of software transactional memory systems.

Chapter 3 (Methodology) covers the Research Methodology that we adopted to exert with this thesis. It explains the research techniques, methods and components used during the findings of the study.

Chapter 4 (Theoretical Work) includes detailed description of all four STM systems – RSTM, TL2, TinySTM, and SwissTM. Finally, it summarizes the research which is related to our work.

Chapter 5 (Empirical Study) gives detail about our experimental platform. Along with it the STAMP benchmark's design and applications are also discussed.

Chapter 6 (Empirical Results) presents our experimental results and describes the observation and finding of our empirical study. It discusses the analysis result of the conducted empirical study in detail.

Chapter 7 (Discussion and Related Work) generalizes the results and relates them to the available literature in this regard.

Finally, we conclude the dissertation and suggest areas for further research.

CHAPTER 1: PROBLEM DEFINITION

Initially there was not much emphasis on building and developing applications for parallel architectures. Most of the applications were being developed for sequential architectures, and sequential architectures performed well for many years. With the advent of the technology, computers became popular in all spheres of life. It was a dire need of the time to build such systems which can fulfill our present and future needs in the theme of more processing capable systems. Further increase in the processor speed was not possible due to some design limitations in sequential architectures. One solution to this problem was parallel computing. In parallel computing we have more than one processor to accomplish a task. The parallel computing can give us power of doing complex and lengthy tasks in a trivial time as compared to sequential computing. Parallel computers share their resources like memory, hard disk and processors to process complex and lengthy instructions easily, and in a timely manner.

No doubt there are many advantages of parallel computing, but we are lacking in standards on which parallel applications can be built. It is hard for a programmer to code, build, debug and test applications for parallel architectures [1]. Working with concurrent programs is difficult but database community has been using concurrent architectures successfully for decades, both on sequential and parallel architectures. The basic entity of a database is the transaction, which constitutes a set of instructions that is completed in an atomic way. The similar transaction mechanism was taken from database for parallel architectures, where instructions in memory are transactionally processed.

As described above in introduction, the transactional memory is of three kinds HTM, STM and HyTM. We focus on STM implementations of transactional memory. A lot of research is going on in this area to build the most efficient STM implementation. This research study focuses on performance comparison of four different STM implementations. The STM implementations that we choose for experimental purposes are:

- Rochester Software Transactional Memory System (RSTM) [7]
- Transactional Locking II (TL2) [8]
- TinySTM [9]
- SwissTM [10]

1.1 Problem Focus

There are a number of STM systems available, each one having advantages and drawbacks. Some are good with heavy workloads while others deal well with tiny workload. Some work fine in high contention environment, while others in low contention environment. Some are good in both managed and unmanaged environment while on the other hand some only work in a managed environment.

1.2 Aims and objectives

The aim of this study is to investigate and compare different approaches of STMs which helps in a deeper understanding of various design choices and potential performance tradeoffs. Later on, this study analyzes the suitability of an STM with respect to another STM. It also presents transactional execution metrics commonly used to characterize TM applications.

To achieve this aim we have set the following objectives:

- Finding problems with traditional lock-based approaches
- Identifying the design alternatives in STM systems
- Comparing performance of STMs on the basis of transactional execution metrics

1.3 Research Questions

In order to understand the performance tradeoffs of different implementations of STM a comprehensive comparative study is required. Although some comparison studies [12-14] have been carried out in the past but those were very focused in their scope and covered only a few STM implementations. That's why our proposed study and experiment is important enough to warrant a study. This master thesis will primarily address the following research questions (RQ):

- RQ1: Which approaches exist to support software transactional memory?
- RQ2: What are the performance tradeoffs between the various approaches?

CHAPTER 2: BACKGROUND

Computer technology has gone through profound changes since its invention. Every decade added new attributes to it and better mechanisms were replaced with substandard ones. Computers have aided humans competently in different areas of life including engineering, medical, automobile industry, space, defense etc since its very beginning, making human life easy in several aspects. Computers became a consumer product and got popularity with the advent of personal computers PCs. The development of microprocessors turned into PCs affordable for general public as they were low in cost as compared to mainframes. The mainframes were large, costly computers owned by large corporations, government and educational institutes, and similar size organizations, but they were not affordable by general public.

The advent of microprocessor served well for several years due to the fact that quantity of transistors was being increased exponentially after every two years according to Moore's law [15]. Increasing clock speed to get better performance is also not feasible due to power consumption and cooling issues. This bound is an inflection point for the replacement of conventional uniprocessor systems as Intel's founder, Andrew Grove says "time in the life of a business when its fundamentals are about to change" [16].

2.1 Single Chip Parallel Computers

To fill this gap of halted performance in processing capacity of single microprocessor, single chip parallel computers were introduced, known as chip multiprocessors or multicore processors. The theme of this type of architecture is to put two or more processors onto a single chip. The architecture defined in single chip processors is similar to shared memory multiprocessors. The number of processors that can be fixed on a chip can be increased and the number of instructions that can be processed in a second will also keep on increasing according to Moore's law [15], even without increasing clock speed. If we want higher performance, we can add more processors according to this architecture.

Although people have worked with parallel computing structures for more than 40 years, there are not many well appreciated programs written for this architecture. It's a tough job to write programs for parallel architectures as compared to sequential architecture. Coding, debugging, testing parallel programs is tough, because there are not well defined standards to debug and test parallel programs [3].

2.2 Database Systems and Transactions

In [17], Herb Sutter and James Larus pointed out, The concurrency revolution is primarily a software revolution. The problem is not building multicore hardware, but programming it in a way that lets mainstream applications benefit from the continued exponential growth in CPU performance. To write programs for parallel systems has been a difficult task, but on the other side database community has been using concurrency for decades.

Databases are working successfully on parallel and sequential architectures. All the concurrency control mechanisms are handled implicitly by the database. At the core of database systems are transactions. A transaction is a set of instructions executed as a whole and used to access and modify concurrent objects. A transaction can be successful when it changes some state in underlying database, and non-successful or aborted when there is no change in the state of a database. Transactions are implemented through a database system or by a transaction monitor, which hides complex details from a user and provides a simple interface to communicate with [3].

As databases were dealing well with transactions on both sequential and parallel architectures, and providing satisfactory results. Therefore, using transactions in the memory was considered a good idea to employ with parallel architectures, so the basic idea of transactional memory is taken from database transactions. Database transactions and transactional memory differ in some aspects because of their basic design needs. Database transactions are saved on disks which can afford to take more time to execute a transaction as compared to transactional memory works in memory which have trivial time to complete a transaction. A database transaction constitutes of a set of instructions that are indivisible and instantaneous. Database transactions have four basic properties Atomicity, Consistency, Isolation and Durability collectively called ACID.

Atomicity: Atomicity means that either all actions are completed successfully or none of them starts its execution. If a transaction fails partially due to some reasons then it fails completely and needs to be restarted.

Consistency: Consistency means transition from one stable state to another stable state. If there is some transaction taken place in a database or memory, then that database or memory will only be considered in a consistent state when the transaction is executed or aborted successfully.

Isolation: Isolation means a transaction is producing results correctly without intervention of other transactions. Running parallel transactions will have no effect on the working of other transactions. A successful transaction will not have any effect and interference on concurrent transactions during its execution.

Durability: The final property of database systems is durability and it is only specific to database transactions. Durability means if some changes have taken place in a database then these are made permanent. This proper is only needed in databases, because memory transactions become obsolete as soon as the system shuts down.

2.3 Transactions vs. Locks

The following figure 1 taken from [1] gives us an idea of how the performance improves as we compare transactions with locking (coarse-grained and fine-grained).

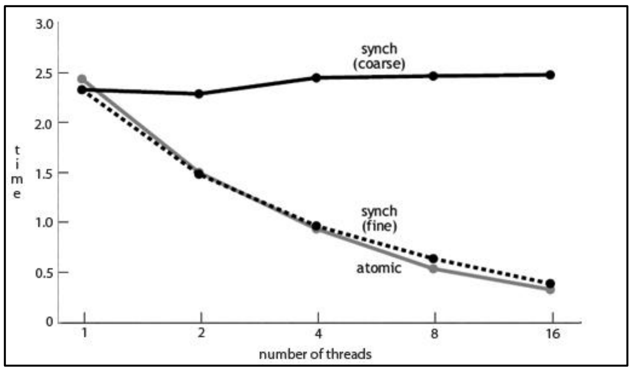


Figure 1: Performance of Transactions vs. Locks [1]

In this figure three different versions of Hash Map are compared. It compares time different versions take to complete a fixed set of insert, update and delete operations on a 16-way Symmetric Multiprocessor (SMP) machine. As the figure shows, increasing number of processors has no effect on coarse-grain while fine-grain and transactions give better performance. Thus coarse-grain locking is not scalable whereas fine-grain locking and transactions are scalable. That's why according to Adl-Tabatabai in [1] transactions give us same results as lock-grain with less programming effort.

It is hard and time consuming to select an appropriate locking strategy for any given problem, and it becomes even more difficult by following additional challenges of the lock-based programming language as presented in [18]. Due to the below mentioned problems and drawbacks, lock-based parallel programming is not a suitable paradigm for an average programmer.

- ***Deadlock*** primarily occurs when two or more threads acquire locks which are required by some other threads in order to proceed, and it causes a state known as circular dependence which is hard to satisfy. As the entire threads wait until lock is released by the other thread, none of threads can make any sort of progress which results in application hang. Deadlock may easily arise if fine-grained locking is used and no strict method of lock acquisitions is enforced. If such methods are not sufficient, resolution schemes and deadlock detection can provide backup in this regard. It is worth mentioning that these schemes are quite difficult to implement and are also vulnerable to live locks, specifically where threads frequently interfere with each other and as a result demising the progress.

- ***Convoying*** occurs upon de-scheduling of a thread holding a lock. During sleep, all other threads execute until and unless they require a lock, due to which many threads had to wait for the acquisition of same lock. As the lock releases, all the threads in a wait contend for this lock which thus causing excessive context switching. However, unlike deadlocks, application continues to progress but at relatively slower pace.

- ***Priority inversion*** occurs when a thread of lower priority holds a lock which is required by some other thread having high priority. In such a scenario, high priority threads will have to discontinue its execution until the lock is released by lower priority thread causing its effective and temporary demotion to the priority level of other thread. In other scenario, if a thread having medium priority is present it may further delay both high and low priority threads and cause inversion of medium and high priorities. There is a problem in priority inversion when discussing it for real time systems, because a thread having high priority may be blocked thus breaching time and response guarantees. However, for general purpose computing, these high priority threads are quite often used to accommodate user interaction tasks not the critical ones. Priority reduction may affect the performance of an application.

- Lock based code cannot be considered as ***composable***. It means that combining lock protected atomic operations into operations having large magnitude and still remain atomic is quite impossible.

- Finally, lock-based code is quite susceptible to faults and failures which are known as ***fault tolerance***. In a case when a single thread holding a lock fails, all of the other threads requiring that particular lock will eventually stop making progress. Failures are likely to increase as the numbers of processors are growing in parallel machines.

2.4 Transactional Memory

Transactional memory is a lock free synchronization mechanism defined for multiprocessor architectures. It is an alternative to lock-based programming. It provides programmers the ease of using read-modify-write operations on independently chosen words of memory. Transactional memory makes parallel programming easy by allowing programmers to enclose multiple statements accessing shared memory into *transactions*. Isolation is the primary task of transactions however, failure atomicity and consistency are also important. In a TM system a failure atomicity provides automatic recovery on errors. But if a transaction is in an inconsistent state then it will not be possible for a written program to produce consistent and correct results. If a transaction fails it can leave results in an inconsistent state. There should be some proper mechanism to revert changes to a previous consistent state.

Many proposed TM systems exist, ranging from full hardware solution (HTM) [19-22] to full software approach (STM) [23-26]. Hybrid TM (HyTM) [6, 27-29] is an additional loom which combines the best of both hardware and software i.e. the performance of HTM and the virtualization, cost, and flexibility of STM.

2.4.1 Hardware Transactional Memory

The idea of hardware transactional memory was introduced by Herlihy and Moss [4] in 1993. Hardware transactional memory (HTM) was first presented as a cache and cache-coherency mechanism to ease lock-free synchronization [30]. The HTM system must provide atomicity and isolation properties for application threads to operate on shared data without sacrificing concurrency. It supports atomicity through architectural means [19], and proposes strong isolation. It also provides an outstanding performance with a little overhead. However, it is often not efficient in generality. It bounds TM implementations to hardware to keep the speculative updated state and as such is fast but suffer from resource limitations [31].

Modern HTMs are divided into different categories that support unbounded transactions and those that support large but bounded transactions. Most of them concentrate on a mechanism to enlarge the buffering for transactions seamlessly [3]. Bounded HTMs enforce limits on transaction working set size, ensuring that transactions following this set size will be able to commit. Best-effort HTMs implement limits by leverage available memory already present in L1 and L2 caches. Unbounded HTMs have been proposed recently that is contrary to bounded HTMs, it allows a transaction to survive context switch events [3]. However to implement these systems is complex and costly. It is most likely that HTMs will prevail as STMs are particularly gaining a lot of attention these days. These HTMs can effectively be utilized with the existing hardware products and also provide an early prospect of gaining experience by utilizing actual TM systems and programming models.

2.4.2 Software Transactional Memory

The idea of software transactional memory was introduced in 1995 by N. Shavit and D. Touitou [5]. Software transactional memory (STM) implements TM mechanisms in software without imposing any hardware requirements. Since all TM mechanisms are implemented entirely in software without having any particular hardware requirements, STMs offers a better flexibility and generality as all mechanisms are implemented in the entire software. The STM can be defined as non-blocking synchronization mechanism where sequential objects are automatically converted into concurrent objects. In STM, a transaction is a finite sequence of instructions which atomically modifies a set of concurrent objects [32]. The STM system supports atomicity through languages, compilers, and libraries [19].

The recent research in STM systems has focused mainly on realistic dynamic transactions. The latest work in STM systems has made them a perfect tool for experimenting and prototyping. As software is more flexible than hardware, it is possible to implement and experiment it on new algorithms. It supports different features like garbage collection that are already available in different languages [3]. In addition, STMs are based on a very critical component being used by number of hybrid TMs, which provide leverage to HTM hardware. Because of this perspective STMs provide a basic foundation to build more efficient HTMs. As a matter of fact, primarily STM systems are considered for this thesis.

STM vs. database transactions: We believe an STM needs not to preserve its transactions to survive the crash as databases do. Concurrency analysis by Felber et al. [33] is a sensitive and crucial issue which needs full attention of the programmer.

- ***Durability*** is a challenge for database transactions. An STM system does not need to preserve its transactions to survive the crash. Therefore in STM transactions, we do not need durability as we need it in databases.

- In terms of ***programming languages***, the database transactions run as SQL statements where each statement runs as a single transaction, different transactions cooperate with each other in order to accomplish a task. While in memory transactions, it is the responsibility of the programmer to define a block of code that runs atomically.

- In terms of ***Semantics***, databases use *serializability* to protect its data from expected behavior. Serializability means each individual transaction is marked non-overlapping in time if it produces the same results as it would have been executing serially. Concurrency analysis is a sensitive and crucial issue which needs full attention of the programmer. As accessing data from transactional and non transactional code, any shortfall in concurrency analysis may lead towards totally inconsistent and devastating results. However concurrent STM transactions may lead to read-write conflicts, producing non-serialized results. STM runtime should implement *recoverability* theory to avoid this problem. Another problem is to handle conflicts caused by transaction reading between two updates of concurrent transactions overwriting each other.

- ***Transformation of transactional code*** is also a challenge in STM. In databases non transactional code runs inherently as a transaction. In STM this is done by either separating transactional and non transactional code or dynamically categorizing their access to shared objects. Monitoring of read access and write access is very crucial in the implementation of STM transactions. It is a challenging task to differentiate between these accesses, as even the use of encapsulation is not sufficient for their separation. To gain optimization and boost in performance, STM transactions are designed to run on multi-core systems, in contradiction to database transactions. To achieve this level of optimal performance is yet another challenging task in STM.

2.4.3 Hybrid Transactional Memory

Hybrid Transactional Memory (HyTM) was introduced in 2006 by P. Damron et al [6]. They worked on a new approach by which transactional memory can work on already existing systems. It has both the flavors of HTM and STM. HyTM can give the best performance and is scalable as well. The HyTM can utilize HTM properties to get better performance for transactions that do not exceed hardware limitations and can obviously execute transactions in STM. When STMs are combined with HTMs like in HyTM, they provide support for unbounded transactions without requiring any complex hardware. In HyTM small transactions are processed on lower overhead of HTMs, while larger transactions fall back onto unbounded STMs. This model of transaction handling is quite appealing in TM as it gives flexibility of adding new hardware with lower development and testing cost and decreased risk [27].

2.5 **STM Design Alternatives**

Design differences in STM systems can affect a system's programming model and performance [3]. In this section we review some distinctive STM design differences that already have been explored in the literature. Our purpose is to be able to identify the impact of these design differences on system performance.

2.5.1 Transaction Granularity

The basic unit of storage over which an STM system detects conflicts is called transaction granularity [3]. Word-based and object-based are two classes of transaction granularity in STMs. Detecting conflicts at word level gives the highest accuracy but it also has higher communication and bookkeeping cost. In object-based STMs, resources are managed at the object level granularity. This implies that the STM uses an object oriented language approach which is more understandable, easy to implement, and less expensive.

2.5.2 Update Policy

A transaction normally updates an object and modifies its contents. When a transaction completes its execution successfully it updates the object's original values with updated values. Based on the update strategy, *direct update* and *deferred update* are two alternative methods described in [3].

Direct update: In direct update, a transaction directly updates the value of an object. Direct update requires a mechanism to record the original value of an updated object, so that it can be reversed in case if a transaction aborts.

Deferred update: In deferred update, a transaction updates the value of an object in a location that is private to the transaction. The transaction ensures that it has read the updated value from this location. The value of this object is updated when a transaction commits. The transaction is updated by copying values from the private copy. In case the transaction aborts, the private copy is discarded.

2.5.3 Write Policy

Whenever a transaction is executed it can make some changes in shared resources. Atomically the transaction either modifies all or nothing. Committing or aborting a transaction is not always successful. That is why, in STM systems, a mechanism is provided to handle both successful commits and aborts. The two approaches that are used to handle this problem are *Write-through or undo* and *Buffered write* described in [34].

Write-through or undo: In this approach changes are directly written to the shared memory. For safe side, each transaction keeps an undo list and reverts their updates in case they need to abort. The write-through approach is really fast as changes are made directly to the shared memory. But aborting a transaction can be very expensive as all the made changes need to be undone.

Buffered write: In this approach, writes are buffered and changes are only made upon successful commit to the shared memory. Here in buffered write approach, aborting a transaction is simple as no changes are made to the shared memory. To commit a transaction values are copied from the buffer to the shared memory.

2.5.4 Acquire Policy

Accessing the shared memory exclusively is called acquiring it. There are two strategies of acquiring shared memory are *Eager* and *Lazy acquire* described in [34].

Eager acquire: If a transaction acquires shared resources and modifies them as well then, it is called eager acquire. Using eager acquire transactions has advantages, because they know as soon as possible that the shared resource is being accessed by some other transaction. Eager acquire has drawbacks in case of long transactions, because the current long transaction will not allow any other transaction to access the shared resources until it completes its working.

Lazy acquire: The lazy acquire strategy works best with buffered writes as the memory is modified only at the commit time. Using this approach ensures that as all the computations are completed, all the changes can be written back to shared memory without any intervention. Using the lazy acquire with the write-through and undo does not suit as it will not do any work at the commit time, therefore it is the wastage of resources.

2.5.5 Read Policy

There are two kinds of read policies [34] invisible and visible reads. In invisible reads multiple transactions can read the same shared resources without any conflict, so most STM systems make shared resources invisible. In invisible reads each transaction validates its read set before commit. In visible reads, STM systems acquire locks or offer a list of readers for each read set on shared objects. In this policy when a transaction wants to modify a shared resource, it checks if there are any readers on that shared resource. If other reader found then it must wait until the resources get free.

2.5.6 Conflict Detection

An important task of STM is to detect conflicts. A conflict occurs when two or more transactions try to acquire and operate on the same object. Most of the STMs employ single-write multiple-read strategy. They also distinguish between RW (Read-Write) and WW (Write-Write) conflicts. The Conflict can be detected at different phases of a running transaction [3]. Detecting conflict before commit falls into the category of *early conflict detection* which reduces the amount of computation by the aborted transaction. Detecting conflict on commit is known as *late conflict detection* which maximizes the amount of computation discarded when a transaction aborts.

2.5.7 Concurrency Control

An STM system that executes more than one transaction concurrently requires synchronization among the transactions to arbitrate simultaneous accesses to an object [3]. This is necessary both in direct update and deferred update systems. These three events (conflict occurrence, detection, and resolution) can occur at different times but not in different order. In general, there are two alternative approaches to concurrency control.

Pessimistic concurrency control: With *pessimistic concurrency control*, all three events happen at the same time in execution. As a transaction tries to access a location, the system detects conflict and resolves it. In this type of concurrency control, a system claims exclusive access to a shared resource and prevents other transactions from accessing it.

Optimistic concurrency control: With *optimistic concurrency control*, detection and resolution of conflicts can happen after conflicts occur. In this type of concurrency control multiple transactions can access an object concurrently. It detects and resolves conflicts before a transaction commits.

Another feature of concurrency control is that its *forward progress guarantees* with two approaches blocking synchronization (lock-based) and non-blocking synchronization (wait-free, lock-free, and obstruction-free).

Lock-based: This STM does not provide any guarantee of progress because locks are used in the implementation in order to ensure mutual and exclusive access to the shared resources.

Wait-free: A wait-free STM guarantees greater progress. In wait free, progress is made by all the threads in a finite number of steps keeping an entire system in context. It is quite difficult to achieve such high progress as it requires that a thread which may not even get CPU time should be assigned by the scheduler in order to make progress in a finite number of system steps. In such an STM, all the threads require to workout with each other in order to make sure that every thread is making progress.

Lock-free: The main difference between the lock-free and wait-free is that a STM which guarantees lock-free only makes sure that the progress is made by at least one thread in a finite number of steps keeping an entire system in context. This minor difference has a significant impact on the STM implementation. However threads still may need to work out with each other only in the scenario when a conflict is raised. If there is not conflict amongst the thread than each of the thread can run without any hitches.

Obstruction-free: An obstruction-free STM guarantees even further less progress. A precise fact of obstruction-free is that it ensures the progress of at least one thread in a finite number of steps in the absence of disputation. Furthermore, in obstruction-free STM if one thread is making some sort of progress than other threads, it will be aborted in order to resolve any conflict. However, one interesting fact of obstruction-free STM is that it surpassed lock- and wait-free STM implementations. This highlights that an increase in the performance by minimizing the guarantees is much larger than the overheads which may be introduced by any amount of additional aborts.

2.5.8 Memory Management

Classen described in [34], the memory management is referred to as allocation and de-allocation of memory. If a transaction allocates memory and is not successful, it should be possible to free the allocated memory; otherwise it can result into memory leakage. On the other side if a transaction de-allocates memory and is not successful or it aborts, then this memory should still be available to restore into previous state. The allocation and de-allocation of memory can be viewed as another form of write operations.

2.5.9 Contention Management

According to Classen [34], an STM needs a contention manager. The role of contention manager is to resolve conflicts. There is an attacker and a victim during a conflict among different transactions. Upon a conflict between two transactions, the contention manager can abort the victim, or abort the attacker, or force the attacker to retry after some period. The contention manager can use different techniques to avoid future conflicts. Following are different management schemes for conflict resolution:

- The simplest Timid [35] – always aborts a transaction whenever a conflict occurs.

- Polka [23] – backs off for different intervals equal to the difference in priorities between the transaction and its enemy.

- Greedy [36] – The greedy contention manager guarantees that each transaction commits within a finite or bounded time.

- Serializer [37] – The serializer works like greedy contention manager with an exception that on aborting a transaction , each transaction gets a new priority.

Regardless of the management schemes a contention manager implements, it must select one of the following options whenever a conflict occurs:

Wait: A simple way of resolving a conflict is to wait for some time until the resolution of an issue on its own. This may seem to be a naive way but it has the tendency to work in many scenarios.

Abort self: In some cases, it is not possible for a transaction to carry on its work due to the fact that another transaction may be holding the shared resource which is required by this transaction. A way to resolve this situation is that this transaction aborts itself and restarts again. This option can be considered as another simpler way to implement because all STMs must have a mechanism of aborting a transaction.

Abort other: One of the last options is to abort the transaction which is holding the lock of required shared resources by the in progress transaction. This option can be considered as quite practical if transactions are priority based. A transaction having high priority aborts those transactions having low priority. This option is quite difficult to implement as compared to the above two options.

CHAPTER 3: METHODOLOGY

This chapter addresses the methodology chosen to answer the presented research questions to achieve the main goals of this research. We are using a mixed methodology to explore deeply our study area. According to C. B. Seaman, a mixed methodology is such a methodology that covers both qualitative and quantitative areas of a research [38]. The motivation behind selecting mixed approach was to first get better and extensive understanding of the problem by conducting literature review. In the second phase, an experiment was conducted in order to address and solve this problem.

3.1 Qualitative Research Methodology

The qualitative part of research is composed of exploration of any activity [38]. The qualitative part of our research is used to answer RQ1 and it also partially answers RQ2. We are using the qualitative research methodology in the following way.

3.1.1 Literature Review

First, a literature study is carried out to collect the material related to both STM performance issues and the techniques developed to solve these issues. The literature review is a qualitative approach [39] that helps in collecting a wide range of information. It is used to increase our knowledge on the topic by analyzing the viewpoints of different researchers. The research papers close to our research area are sorted out by identifying the significant material that will aid us in fully understanding STMs and their performance. This study provided us sound ground knowledge generally about transactional memory and especially about understanding of different software transactional memory systems. The digital libraries and online databases which were utilized in this regard are as follows:

- IEEE Xplore
- ACM Digital library
- Springer Link
- Google Scholar (scholar.google.com)
- Transactional Memory Bibliography (cs.wisc.edu/trans-memory/biblio/index.html)

3.1.2 Background Study

Background study is presented basic understanding of different factors that influence different STMs and affect their performance and prepared the ground about the thesis. It endows with basic concepts required to understand this study. This research starts with identifying the different research articles and books related to our research work which help us in better comprehension of different STM techniques.

3.1.3 Selection and Suitability of STM systems

There exist many implementations of STM i.e. WSTM [40], OSTM [41], DSTM2 [42], SXM [43], McRT-STM [25], DracoSTM [44], and STM-Haskell [24]. In this study, we chose four STM systems for experimental purposes which are briefly described in chapter 4. All systems cover different design properties of software transactional memory. These systems give different results in different environments, with different workloads, different contention management schemes, and deal differently with applied overhead. We have chosen four STM implementations due to the following reasons:

- They are the state-of-the-art STM systems, well appreciated in the research community and all of them are publicly available. Furthermore, they support the manual instrumentation of concurrent applications with transactional accesses. Definitely our objective is to evaluate the performance of the core STM algorithm, not the detail of measuring the efficiency of the higher layers such as STM compilers.

- They characterize an extensive diversity of known STM design choices such as obstruction-free vs. lock-based implementation, invisible vs. visible reads, eager vs. lazy updates, and word-level vs. object level access granularity at which they perform logging. Lock-based STM systems, first proposed in [45], implement some variant of the two-phase locking protocol [46]. Obstruction-free STM systems [47] do not use any blocking mechanisms, and guarantee progress even when some of the transactions are delayed.

- These systems also allow for experiments with different contention management approaches, from simply aborting a transaction on a conflict, through exponential back off, up to advanced contention managers like Greedy [36], Serializer [37], or Polka [23].

3.1.4 Selection and Suitability of Benchmarks

We chose STAMP – Stanford Transactional Applications for Multi-Processing [11] – benchmark suite to compare the performance of our STMs because it offers variety of workloads and has been extensively used to evaluate TM implementations [48]. It is portable across a whole range of transactional memory implementations including: hardware transactional memory, software transactional memory and hybrid transactional memory. It covers a wide range of transactional behaviors. It consists of eight applications including bayes, genome, intruder, kmeans, labyrinth, ssca2, vacation and yada. It is publicly available at http://stamp.stanford.edu.

3.2 **Quantitative Research Methodology**

A quantitative study is presented where we performed performance measurements on real hardware. The quantitative research methodology is used to answer RQ2. We are using the quantitative research methodology in the following way.

3.2.1 Selection and Suitability of STM Performance Metrics

For analyzing, the transactional behaviors of a set of complex realistic TM applications, following metrics are commonly used:

- Commit Phase Time and Abort Phase Time
- Commit Reads and Abort Reads
- Commit Writes and Abort Writes
- Execution time
- Aborts per Commit
- Transaction Retry Rate

The metrics that we measured during this study are Execution time, Aborts per Commit and Transaction Retry Rate. The inspiration behind selecting these metrics is that they help in determining the transactional scalability of the applications. The execution time shows the transactional effectiveness of application scale with respect to the increasing number of threads. Since transactional memory is a scheme that imposes the committing or aborting of transaction sequences, the important issue while trying to monitor the transactional management is naturally the ratio of aborted transactions to committed transactions. Finally, the last metric is used to exploit the inherent concurrency of the underlying STM implementation.

3.2.2 Experimentation

Experiments are considered the cornerstone of the empirical study which is performed on a subject when we have control over the environment. The experiments are used to test the behavior of this subject directly, precisely and systematically. Experiments are performed more than once in order to validate the subject's outcome.

3.2.3 Analysis of Gathered Results

After we performed experiments on the chosen metrics we got some data. This data was scrutinized and on the basis of this data we were able to fetch some results. These results are further discussed in order to attain a final conclusion.

CHAPTER 4: THEORETICAL WORK

This chapter clearly describes the four STMs i.e. RSTM [7], TL2 [8], TinySTM [9], and SwissTM [10] in detail employed for performance evaluation in this thesis. These systems represent a wide spectrum of design choices.

4.1 RSTM

4.1.1 RSTM Overview

The RSTM [7] system was designed by Marathe et al. at the University of Rochester to improve the performance of an obstruction-free deferred-update STM. It was written as a fast STM library for C++ multithreaded programs but an equivalent library could also be implemented for C language, though it would not be more convenient. Obstruction freedom [12] is the weakest guarantee of non-blocking synchronization that simplifies implementation by guaranteeing progress only in the absence of conflict. To make this guarantee, RSTM employs Polka [23] as a contention manager. Contention manager decides what to do on conflict either abort a transaction or spin-wait and which transaction to abort if there is any conflict between transactions.

The basic unit of concurrency over which RSTM detects conflicts is an object. Inside a transaction, objects may be opened for read-only or read-write access. Objects that are opened for read-write are replicas, and those for read-only are not. A transaction that wishes to update any object must first acquire it before committing. Acquiring an object is getting exclusive access to that object. It can be done in eager or lazy fashion. Eager systems acquires an object as soon as it's opened while lazy systems acquires it some time prior to committing the transaction. In some existing STM systems (e.g. DSTM [47], SXM [43], WSTM [40] and McRT [25]) writers acquire objects and perform conflict detection eagerly, whereas some others (e.g. OSTM [41], STM Haskell [24]) do it lazily. Eager acquire aborts doomed transactions immediately, but causes more conflicts. However lazy acquire enables readers to run together with a writer that is not committing yet. A thread that opens an object for reading may become a visible or invisible reader. In either case eager or lazy conflict detection, writers are visible to readers and writers but readers may or may not be visible to writers. RSTM currently supports both eager and lazy acquire and both visible and invisible readers.

The information about acquired objects is maintained in a transactional metadata. RSTM adopts a novel organization for transaction metadata with only a single level of indirection to access an object rather than the two levels used by other systems like DSTM or ASTM [49]. This cache-optimized metadata organization reduces the expected number of cache misses. To further reduce overhead, RSTM is considered for non-garbage-collected languages by maintaining its own epoch-based collector. This lightweight memory collector avoids dynamic allocation for its data structure (i.e. Object Headers, Transaction Descriptors, private read and write lists), except for cloning data objects. The garbage-collected languages increase the cost of tracing and reclamation. RSTM avoids tracing altogether for transactional metadata by a simpler solution to mark superseded objects as retired.

4.1.2 Design Features

One of RSTM's prominent features is a visible reader list which avoids the aggregate quadratic cost of validating a transaction's invisible read list each time it opens an object. An Object Header reserves a fixed-size room for a modest number of pointers to visible reader

Transaction Descriptors. When a transaction acquires the object for write, it immediately aborts each transaction in the visible reader list. A transaction on a visible reader list does not need to validate reads, since a conflicting write will abort the transaction. This implicitly gives writers precedence over readers because there is no chance that a visible reader will escape a writer’s notice. However, RSTM arranges for each transaction to maintain its private read list and validate it. Even so, visible readers can reduce the size of this read list and the cost to validate it.

According to [3], the authors of RSTM strongly argue that STM should be implemented with non-blocking synchronization, because blocking synchronization is vulnerable to a number of problems like thread failure, priority inversion, preemption, convoying, and page faults.

4.1.3 Implementation

In RSTM, every shared object is accessed through an Object Header, which holds the bitmap of the visible readers and the New Data field that identifies the current version of the object as shown in the figure 2. RSTM limits the number of visible readers. The New Data field is a single word that holds a pointer to the Data Object and a dirty bit. In RSTM this lower bit of the New Data field is used as a flag which tells whether Data Object is a clean object or a write-made replica. If the flag is set to zero, then the New Data pointer refers to the current copy of the Data Object. It saves dereference in the common case of non-conflicting access. Otherwise, if the flag is set to one, then a transaction has the object open for writing whenever that object’s Owner pointer points to Transaction Descriptor.

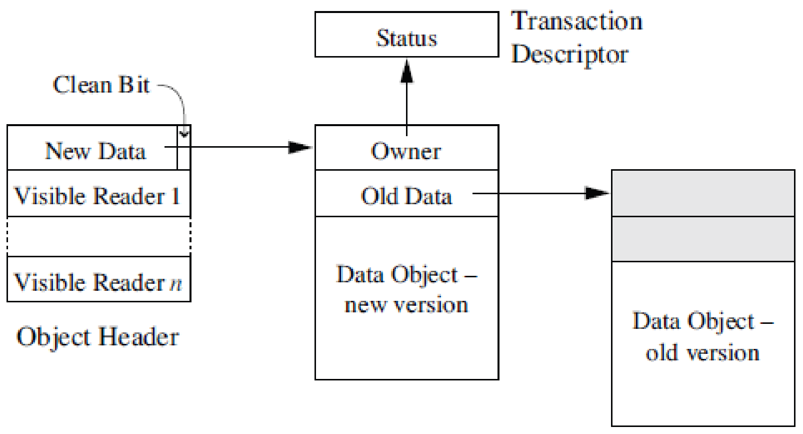


Figure 2: RSTM Transactional Metadata [7]

The Transaction Descriptor determines the transaction’s state, which holds the lists of opened objects (i.e. visible or invisible reads and eager or lazy writes) and the Status that can be ACTIVE, COMMITTED, ABORTED. If the Status is COMMITTED, then Data Object is the current version of the object. If the Status is ABORTED, then Data Object’s Old Data pointer is the current version. If the Status is ACTIVE, no other transaction can read or write the object without first aborting the transaction. To avoid dynamic allocation, each thread has a static Transaction Descriptor that is used for all transactions of this thread.

A transaction opens an object before accessing it. In order to open the object for write, the transaction must first acquire it. To affect an acquire, the transaction reads the Object Header’s New Data pointer to identifying the current Data Object and makes sure no other transaction owns it. If it is owned, the contention manager is invoked to tune performance. Then allocation of a new Data Object and copying of data from object’s current version to the new and initialization of the Owner and Old Data fields in the new object are done. After this step, the transaction uses a CAS to atomically swap the header’s New Data pointer to point to the newly allocated Data Object. At the end, the transaction adds the object to its private write list, so the header can be cleaned up on abort. If the object is open for a read, the transaction adds the object to its visible reader list for post-transaction cleanup. Otherwise if the list is full, it adds the object to the transaction’s private read list for incremental validation.

4.2 **TL2**

4.2.1 TL2 Overview

TL2 [8] is mainly interested in mechanical methods of code transformation from sequential or coarse-grained to concurrent. Mechanical means the transformation of code is done either by hand or preprocessor or compiler. TL2 works fine with any system’s memory life cycle including a support for malloc/free methods used to allocate and free the memory. The user code is guaranteed to work in a consistent state by efficiently consuming execution time.

TL2 provides solution to two potential threats that STM implementations are facing. First threat is Closed Memory Systems and second one is Specialized Runtime Environments. A closed memory system or closed TM is where memory can either be used transactionally or non-transactionally. This implementation is easy to adopt in languages that support garbage collection like java, but it is difficult to handle in languages like C/C++, where user has to code manually to handle memory allocation and free operations. The unmanaged environments give room for execution of Zombie transactions. A transaction is a zombie transaction when it founds an inconsistent read set, but it has not yet aborted the transaction. Efficient STM implementations need special runtime environments that can handle irregular effects of inconsistent states in unmanaged environments. The efficient runtime environments use traps to find problems in transactions and use retry statements to execute a transaction again in hope that it will succeed.

The algorithm provided in TL2 offers a solution to both of the above mention problems i.e. closed memory systems and specialized runtime environment. TL2 by Dice, Shalev and Shavit used open memory system that provides solution to this problem by employing global version clock and commit time locking [8].

4.2.2 Global Version Clock

The global version clock is incremented each time a transaction writes to memory and is read by all other transactions. Transactions recorded in databases use time stamping. Database time stamping is used for large database transactions. But we need such a mechanism that can work efficiently with small memory transactions. To overcome this problem the global clock version used in TL2 is used, which is different from database because it supports working with small memory transactions efficiently. The global clock was also used by Reigel et al. in [50]. Reigel et al. global clock supports time stamping for non-blocking STMs and it is costly. The global clock version used in TL2 is lock based and simple.

In TL2 all memory locations are augmented by a lock which contains version number. Extending or augmenting all memory locations with version number can give a consistent memory state to a transaction at a very little cost. The transactions that need to write or update memory need to know the read and write set before committing. Once read and write sets are available, locks are applied i.e. transactions acquire locks so that no other transaction can change the current state of acquired read or write sets. The transaction will try to commit its new values by incrementing global version clock and checking validity of read sets. Upon successful completion, the transaction will release locks on read and write sets, update memory locations with new value of global version clock [8].

4.2.3 TL2 – Algorithm

The TL2 algorithm uses commit-time locking instead of encounter time locking mechanism to update transactions. The global version clock is read incremented by a write transaction and is ready by all other transactions. To implement data structure in memory we use per object (PO) and per stripe (PS) locks. A PO is assigned to a single object, while PS is assigned to a large number of objects and memory is divided into different portions using some hash function. As PS is defined in TL2 algorithm, almost the same way PO can also be defined. TL2 is defined for write transactions and low cost read only transactions [8].

Write Transaction: There is a sequence of operations that is followed before any write transaction successfully completes. The current global version clock's value is stored in a local variable read-version (rv). Transactional code is executed. Read and write set addresses are locally maintained. Locks are acquired on write set. If locks cannot be acquired due to some reason, the transaction fails. When a lock is successfully acquired on a write set, a value is incremented in global version clock and is written back in local write version (wv). Finally before updating the memory locations with write set, it is confirmed that the read set is consistent. It has not been changed after locks were acquired on write set. Every location is written or updated with write set. The lock is released and the memory locations are made available to be used by other operations

Low Cost Read-Only Transactions: Another task that this algorithm supports is the efficient execution of read only transactions. The value of global version clock is loaded into a local variable read version (rv). There are different ways to implement global version. It can easily become a bottleneck if it is not implemented intelligently. It may introduce contention and cache sharing problems. The solution to this problem is to divide global version clock into version number and thread ID. By using this scheme a thread would not have to change its version number if it is different from last one it used.

Mixed Transactional and Non-Transactional Memory Management: TL2 implementation either divides memory into transactional and non transactional areas thus mix transactional and non transactional operations are not possible. The memory used by a transactional operation should not be allowed to be used by a non transactional operation. But it is a dire need of an STM implementation that memory used transactionally may later be used by non-transactionally. Languages like java support such operations implicitly. But in languages like C/C++, one has to code explicitly to handle memory. Malloc () function is used to allocate memory and free () function is used to free memory space used by a transaction.

Mechanical Transformation of Sequential Code: TL2 supports mechanical code transformation. Code written for sequential programs can automatically be transferred to concurrent code. But manually written code has always edge over mechanically transformed code.

4.2.4 TL2 – Variants

The open-source TL2 library contains several variants which differ in how they manage the global version number. As originally described, TL2 increments the global version number at commit-time for each updated transaction. All transactions must fetch the global version number variable as its timestamp. Thus every transaction has a unique timestamp, and TL2 refers this form of clock management as “GV1”. TL2 later developed and more refined clock management schemes are known as GV4, GV5, and GV6 [51].

GV4: In its default mode, called GV4, TL2 tries to minimize the risk of contention on the global version number by using a pass-on-failure strategy: if a transaction fails to increment the global version number, it does not retry to increment it, but uses the new value of the global version number as its timestamp. This is safe because the global version number is incremented after a transaction has locked all the ownership records “orecs” associated with its writes, and validated its reads. So, transactions that try to increment the global version number do not conflict with each other and can commit at the same time using the same timestamp.

GV5: Another variant of TL2 is GV5, which further minimizes the contention for the global version number on the basis that every transaction is not required to make an attempt to increment the global version number. As an alternative, a transaction reads and increments global version number locally and uses the resulting value as its timestamp but does not write the incremented value back to the global version number. Though GV5 further minimizes contention on the global version number but it increases unnecessary aborts.

GV6: It is an adaptive hybrid of GV4 and GV5 to perform the best possible results. This strategy tries to avoid unnecessary aborts as in GV5 and avoid bottlenecks on advancing global number less frequently than GV4.

4.3 TinySTM

4.3.1 TinySTM Overview

The performance of a STM system depends on different design choices and configuration parameters. We have to select from different design choices, either current STM system is going to be word based or object based, lock based or non-blocking, write through or write back, encounter time locking or commit time locking. Similarly regarding STM implementations, we do make choices among different configuration parameters like the number of locks used to handle concurrent access to shared data or to map locks to memory addresses. These factors are invariably used on different system architectures due to their CPU or cache line size [9].

The work load of an STM implementation plays a major role in selecting right design choice and configuration parameters. The ratios to update read only transactions, the size of read and write sets and contention on shared memory; all these factors make it difficult achieving a STM that is perfectly suitable in all situations. Some STMs are good for one kind of environment but they do not perform well in other type of environment. For example, time based TMs are excellent in read only transactions, but they are not that good in an environment where transactions are updated frequently, validation read sets in larger transactions is also an issue [9].

4.3.2 TinySTM – Algorithm

Locks and Versions: TinySTM uses shared array of locks to manage concurrent access. Addresses are mapped on per stripe which is locked using a hash function. Lock represents size of address in memory. Locks least significant bit shows whether lock is owned or not. If least significant bit of lock is owned, remaining bits of address store owner transaction in case of write-through method or write set owner transaction in case of write-back. And if least significant bit of lock is not owned, a version number is stored in remaining set bits that are based on commit timestamp of the latest transaction (see Figure 5).

Reads and Writes: When a transaction reads a memory location, it first checks whether the read set is currently being read or written by any other transaction. If no other transaction holds lock on the current read set, and the value of read set is not changed then the read set is consistent. When a transaction writes to a memory location, it reads lock entry from selected memory addresses. If it finds lock bit is set then it verifies either the current transaction is the owner or not. If current transaction is the owner, then it simply writes new value to memory location. If it is not owner of the current transaction then the current transaction waits for some time to get resources free or abort immediately. TinySTM uses abort immediate option.

Write-through vs. Write-back: In write-through design changes are directly updated in memory and these changes are saved in an undo log buffer to revert previous values in case of abort. The write-through design has lesser commit time. In write-back design changes are held in a write log until the commit time. The write-back design has lesser abort overhead.

Memory Management: Dynamic memory management can be complex in unmanaged languages and environments. TinySTM has provided memory management functions that help in handling dynamic memory efficiently. Every transaction keeps a record of allocated and freed memory. Memory management functions properly dispose off the allocated memory and freed memory is upheld until commit.

Clock Management: TinySTM is based on shared counter clock which is efficiently working in SMP architectures. If due to some reason this shared counter clock mechanism is creating problem in large systems, a more scalable time based clock or multiple synchronized physical clocks can be used.[52] TinySTM can work on 32 bit and 64 bit architectures. The maximum value of clock on 32 bit architecture is 2^{31} and maximum value for 64 bit architecture is 2^{63}. Frequent commit statements can easily exploit maximum value on 32 bit architectures. TinySTM provides its solution through roll-over mechanism [9]. When a transaction detects that it is facing maximum value, it waits for a little time during which all pending transactions are completed or aborted, the clock counter is restarted.

4.3.3 Implementation

TinySTM is a light weight highly efficient lock based, word based STM implementation. Using word based design choice supports to adopt the direct mapping of memory to any subsystem. Word based STMs allow memory access at word granularity and can be used in unmanaged environments. TinySTM is based on encounter time locking. There are two reasons to use encounter time locking:

- Detecting contention as they occur helps in increasing transaction’s throughput, because transactions do not have to work extra. However, commit time locking may give some advantage of read write conflicts, but conflicts detected late cannot be solved without aborting one of the transactions.

- Reads-after-writes are efficiently handled using encounter time locking and is especially good with large write sets.

Along with encounter time locking, two other strategies are used in TinySTM to access memory: write-through and write-back access. In write-through access transactions immediately write to memory and undo updates if aborted. In write-back access transactions do not update until the commit time. In TinySTM such a procedure is followed that any transaction does not need to access another transaction’s memory [9].

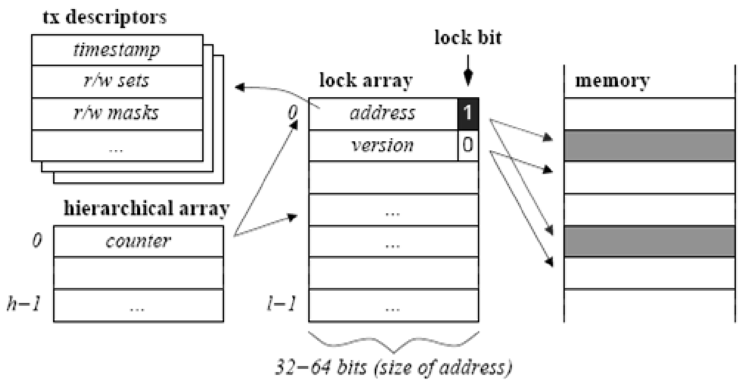


Figure 3: Data structures in TinySTM [9]

4.3.4 Hierarchical Locking

LSA algorithm [31] guarantees that there is a consistent read set for read only transactions, therefore read only transactions do not need to validate their read sets. Update transactions do need to validate their read sets before they update any value of the memory locations. Large read sets validation may be costly. To validate read sets fast, number of locks can be reduced. Reducing number of locks can increase abort rate. The solution of this problem was the introduction of hierarchical locking as proposed by the authors of TinySTM [9]. Hash function is used to map memory addresses to counters which are consistent to lock arrays. Memory locations that are mapped to same lock are also mapped to same counter. Every transaction keeps two private data structures. A read mask and write mask of hierarchical *h* bits. Read sets are divided into h parts.

4.3.5 Dynamic Tuning

TinySTM uses dynamic tuning parameters that affect its performance in achieving higher transaction throughput. First one is using a hash function to map locks to memory locations. TinySTM uses right shifts that provide control over how many contiguous memory locations can be mapped using same lock. Second is number of entries/addresses in lock array. The smaller value will be able to map more locks on a single lock, which decreases the size of read set. Third is the array size used for hierarchical locking. Higher value of hierarchical locking increases atomic operations but it decreases validation overhead and contention.

4.4 **SwissTM**

4.4.1 SwissTM Overview

SwissTM [10] is a deferred, lock based STM system that uses invisible reads and relies on a time-based scheme to reduce the cost of transaction validation and speed up read-set validation, like TL2 [8] and TinySTM [9]. It is a C++ implementation of LSA with dynamic snapshot extensions. It uses a variant of two-phase locking for concurrency control that causes no overhead on all short read-write and read-only transactions while favoring the progress of transactions that have performed a significant number of updates. Two-phase contention manager with random linear back-off embedded in SwissTM is bimodal, distinguishing between small and large transactions. The random linear back-off also increases scalability.

SwissTM uses mixed invalidation [53] conflict detection scheme in which write-write conflicts are noticed early, but conflicts detected in read-write case are not acted upon until the commit time. By using early write-write conflict detection, it avoids wasted work in transactions that is almost certain to abort, since at most one the conflicting transactions can ever commit. By detecting read-write conflicts late, it reduces the number of unnecessary aborts due to false read-write conflicts, in this case both may commit if the reader does so first. Thus SwissTM takes the best of both worlds to support mixed workloads consisting of both short and long transactions, as well as simple and possibly complex data structure. This combined strategy is beneficial for large transactions and complex objects as it introduces no significant overheads on short transactions and simple data structures [10].

The contention management scheme underlying SwissTM gets invoke only on write-write conflicts. To provide good performance across a wide range of mixed workloads, the contention manager aborts conflicting transaction that performed less work by using a shared counter to establish a total order among transactions, similarly to Greedy [36], but it avoids updates to the shared counter for short transactions, resulting in a two-phase contention manager [48]. The two-phase contention management scheme is a variation of the Greedy contention manager. Though Greedy performs poorly on short transactions, two-phase contention manager improves both performance and scalability by overcoming this issue completely. This is because it allows all read-only and short transactions to commit without incrementing the shared counter yet it provides the strong progress guarantees of Greedy. The more complex transactions switch dynamically to the Greedy mechanism that involves more overhead but favors these transactions, preventing starvation [10].

4.4.2 Design philosophy

All transactions share a global commit counter *commit-ts,* incremented by every commit. Every transaction has a transaction descriptor, *tx*, containing the value of *commit-ts,* and the transaction's read and write logs. The value of *Commit-ts* is read upon the transaction beginning or updated by every subsequent validation. Each memory word m is globally mapped to a pair of lock entries for r-lock and w-lock. Lock *w-lock* is eagerly acquired by a writer T to prevent other transactions from writing to m. Lock *r-lock* is lazily acquired by T to prevent other transactions from reading word m. In addition when r-lock is released it contains the version number of m, as a result, observing inconsistent states of words written by T. Every 4 consecutive memory words share a lock [10].

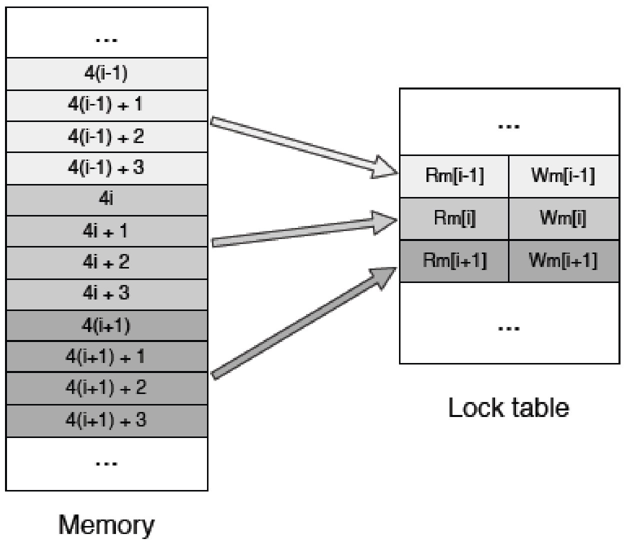


Figure 4: Mapping of memory words to global lock table entries.[10]

4.4.3 Locking granularity

The design of SwissTM is a result of trial-and-error, various choices that might have seemed natural, revealed inappropriate. An important implementation choice underlying SwissTM is its lock table granularity, in particular the size of memory that gets mapped to the same lock. Increasing the size of memory stripes increases abort rates due to false conflicts but reduces locking and validation time due to data access locality. The optimal value for this parameter is application specific. For the experiments described in [10], the value of 4 words was the best. It's interesting to note that picking a different value for the lock granularity may affect the performance but it is not preventing scalability.

4.4.4 Contention Manager – Algorithm

```
cm-start(tx)
if not-restart(tx) then tx.cm-ts ← ∞;
cm-on-write (tx)
if tx.cm-ts = ∞ and size(tx.write-log)=$W_n$ then
tx.cm-ts ← increment&get(greedy-ts);
cm-should-abort(tx,w-lock)
if tx.cm-ts = ∞ then return true;
lock-owner = owner(w-lock);
if lock-owner.cm-ts < tx.cm-ts then return true;
else abort(lock-owner); return false;
cm-on-rollback(tx)
wait-random(tx.succ-abort-count);
```

Figure 5: Pseudo-code representation of the two-phase contention manager [10]

The contention manager gets invoked by the main algorithm,

- At transaction start,
- On a write/write conflict,
- After a successful write, and
- After restart.

Conceptually, every transaction starts with an infinite value of a counter – tx.cm-ts. This is a variation of the Greedy counter greed-ts. This counter gets updated after W_nth writes of the transactions. Upon a conflict, the transaction with higher value of cm-ts is aborted. After restarting, transactions are delayed using a randomized back-off scheme.

4.5 STM Feature Comparison

Table 1 illustrates a summary of the design aspects and outlines the most significant difference among the four state-of-the-art STM systems – RSTM, TL2, TinySTM, and SwissTM – employed for performance evaluation in this study.

Table 1: Feature comparison of the four state-of-the-art STM systems

System Name	Granularity	Update Policy	Write Policy	Acquire Policy	Read Policy	Conflict Detection	Concurrency Control	Progress Guarantee
RSTM	Object-based	Deferred	Buffered	Both	Both[1]	Both[2]	Optimistic	Obstruction-free
TL2	Both	Deferred	Buffered	Lazy	Invisible	Both	Optimistic	Lock-based
TinySTM	Word-based	Both	Both	Both	Invisible[3]	Early[4]	Optimistic	Lock-based
SwissTM	Word-based	Deferred	Buffered	Both	Invisible	Mixed invalidation[5]	Both	Lock-based

[1] visible for up to 32 threads
[2] early or late (selectable)
[3] by default invisible, but can switch to visible to help with contention management
[4] early, but depends on acquire policy (if acquire is commit-time, conflict detection is obviously late)
[5] A conflict detection scheme in which write-write conflicts are noticed early, but read-write conflicts are detected late.

CHAPTER 5: EMPIRICAL STUDY

In the preceding chapter we sketched the key design aspects of the most recent STM implementations. One goal of this thesis is to identify performance tradeoffs of these STM design alternatives. In order to evaluate these systems, the applications of the STAMP benchmark suite [11] have been ported to these systems. In this chapter we introduce our evaluation framework and cover STAMP in detail.

5.1 Experimental Platform

To compare the performance of all the four STMs – RSTM, TL2, TinySTM and SwissTM – we conducted different tests on a computer with 2 quad-core Intel(R) Xeon(R) CPU E5335 @ 2.00GHz processors (in total 8 cores). This machine has 16 GB of RAM running with Linux operating system. All observations were gathered by running each STM system more than once. In order to analyze the outputs and to obtain fully qualified results, averaged results were taken from multiple runs. All the performance tests were performed using the STAMP Benchmark. The acquired results are graphically presented in this chapter.

We used RSTM (version 5), the TL2-x86 0.9.6 implementation, TinySTM (version 0.9.9), and SwissTM (release dated: 2009-09-10). All these STMs were tested using STAMP Benchmark suit (version 0.9.10). These STM implementations are available from their respective websites. We performed tests on these implementations on their default configurations: RSTM was configured to use (eager conflict detection, invisible reads with commit counter heuristic, and the polka contention manager), TL2 is configured to use (lazy conflict detection and Global Version 4 (GV4)), TinySTM is configured to use (encounter time locking and timid contention manager), and SwissTM is configured on (mixed validation: Optimistic (commit-time) conflict detection for read/write conflicts and pessimistic (encounter-time) conflict detection for write/write conflicts, and a new two-phase contention manager).

5.2 STAMP Benchmark

Benchmarks for STM implementations are still very few [54]. Recently, some complex benchmarks for elongating STM implementations have emerged. STAMP [11] is a new comprehensive benchmark suite designed for TM research which currently consists of eight different applications representative of real-world medium-scale workloads. In table 2 we specify a brief description of these applications. They provide runtime transactional characteristics like varying transaction lengths, frequent or rare use of transactions, time spent in transactions and the average number of retries per transaction.

Table 2: The eight applications in the STAMP suite [11]

Application	Domain	Description
Bayes	Machine learning	Bayesian network structure learning
Genome	Bioinformatics	Performs gene sequencing
Intruder	Security	Network intrusions detection
Kmeans	Data mining	Implements partition-based clustering
Labyrinth	Engineering	Routes shortest-distance in maze
ssca2	Scientific	Efficient graph construction
Vacation	Online transaction processing	Client/server travel reservation system
Yada	Scientific	Refines a Delaunay mesh

We choose STAMP to compare the performance of our STMs because it offers a variety of workloads and has been extensively used to evaluate TM implementations [48]. Moreover, STAMP is portable across many types of TM implementations, including hardware, software, and hybrid implementations and publicly available at http://stamp.stanford.edu.

5.2.1 STAMP – Design

The design of the STAMP benchmark suite offers a comprehensive breadth and depth analysis and is portable to many kinds of TMs (HW, SW, and Hybrid) to make it an effective tool for evaluating TM systems.

Breadth: STAMP consists of eight different applications covering different domains and algorithms. TM simplified development of each ones that are not trivially parallelizable as they can get benefit from TM's optimistic concurrency.

Depth: STAMP covers a wide range of important transactional behaviors. STAMP is also facilitated by multiple input data sets and different configuration settings per application. STAMP applications spend a significant portion of their execution time within transactions.

Portability: STAMP can easily work with HTM, STM, and HyTM systems. The code for all benchmarks is written in C with macro-based transaction annotations to indicate both transaction boundaries and memory accesses that require instrumentation for STMs and HyTM designs. The same annotations are used by TM versions of the code. C macros make these annotations easy to replace, remove, or port to different systems.

5.2.2 STAMP – Applications

The STAMP applications include a Bayesian structure learning network "Bayes", a gene sequencing program "Genome", a network intrusion detection algorithm "Intruder", a k-means clustering algorithm "KMeans", a maze routing algorithm "Labyrinth", a set of graph kernels "SSCA2", a client-server travel reservation system simulating SPECjbb2000 "Vacation" and finally a Delaunay mesh refinement algorithm "Yada".

Bayes: This application implements an algorithm for learning the structure of Bayesian networks, which is an important part of machine learning. Often, Bayesian networks are learned from observed data. Conceptually, a Bayesian network is represented as a directed acyclic graph, where a node represents a variable and an edge represents a conditional dependence between variables. The graph tries to represent the relation between variables in a data set. This particular algorithm is based on the hill climbing strategy that combines local and global search, similar to the technique described in [55]. For efficient estimates of probability distributions, the ad-tree data structure from [56] is used. All operations on the acyclic graph occur within transactions. Overall, this application has a high amount of contention as the sub-graphs change frequently.

Genome: This application implements a gene sequencing program that processes a list of DNA segments and matches them to reconstruct the original larger genome. The algorithm uses transactions for removing duplicate segments by using hash-set to create a set of unique segments and paring them with existing segments using a Rabin-Karp string matching algorithm [57]. Conflicts occur when threads try to use the same segment during the matching phase [54]. In general, the application is highly parallel and almost contention free. Additionally, the transactions are of moderate length and have moderate sizes of read and write sets.

Intruder: Signature-based Network Intrusion Detection Systems (NIDS) scan network packets for matches against a known set of intrusion signatures. Haagdorens et al. in [58] presents various techniques for implementing network-intrusion detection. This application implements "Design5" of the NIDS described by Haagdorens et al. which splits the algorithm into three stages (capture, reassembly, and detection) to exploit pipelined parallelism of network packets. Transactions are used to protect the FIFO queue in stage one (capture) and in stage two (reassembly) transactions use the dictionary that contains lists of packets. Overall, since two of the three stages are spent in transactions, this application has a moderate amount of total transactional execution time.

K-means: This application was taken from MineBench [59]. The K-means algorithm is a partition-based method [60] and is commonly used clustering technique where a number of objects with numerous attributes are partitioned into a number of clusters. K-means represents a cluster which is the mean value of all objects contained in it. This algorithm is essentially data parallel. Conflicts occur when two threads attempt to insert objects into the same partition. Varying the number of partitions affects the amount of contention among threads but transactions in K-means are used to update the cluster centers, for which there is very little contention.

Labyrinth: This application implements a variant of Lee's algorithm [61] in which the maze is represented as a three-dimensional uniform grid, where each grid point can contain connections to adjacent, non-diagonal grid points. This algorithm is guaranteed to find the shortest path between the start and end points of a connection. Overall, almost all of labyrinth's execution time is taken by the calculation of the path. This operation also reads and writes an amount of data proportional to the number of total maze grid points. While creating the transactional version of this program, the techniques described in [62] were used to reduce the chance of conflicts. Transactions are beneficial for implementing this program as deadlock avoidance techniques would be required in a lock-based approach. The amount of contention is very high in it because of the large number of transactional accesses to memory.

Ssca2: The Scalable Synthetic Compact Applications 2 (SSCA2) application [63] is comprised of four kernels that operate on a large, directed, and weighted multi-graph. STAMP focuses on Kernel 1, which constructs an efficient graph data structure using adjacency arrays and auxiliary arrays. The transactional version of SSCA2 has threads which add nodes to the graph in parallel, and uses these transactions to protect accesses to data to adjacent arrays. Due to the fact that this operation is small, transactions do not take much time to execute. In addition, transaction's length and their read and write sets size is also small. Due to this the amount of contention in this application is also low.

Vacation: This application implements an enterprise travel reservation system powered by a non-distributed database. The 3-tier design of vacation shown in figure 6 is similar in design to SPECjbb2000 [64]. The system consists of several client threads interact with an in-memory database via the system's transaction manager that implements the tables as Red-Black trees. In particular, client threads acting as customers try to reserve, cancel, and update their records, while performing actions such as, booking hotel rooms, flights and renting cars. Coarse-grain transactions are used during each of these client sessions to ensure validity of the database. In conclusion, these transactions greatly simplified the parallelization for all the data structures in vacation.

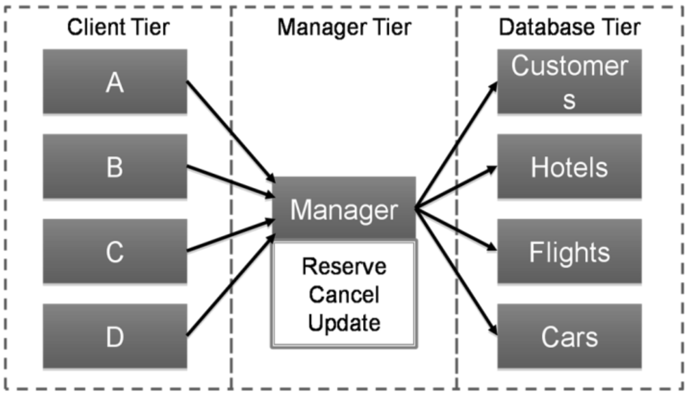


Figure 6: Vacation's 3-tier design

Yada: The Yet Another Delaunay Application (YADA) implements the Delaunay mesh refinement [65]. There are primarily two data structures (1) a set that contains the mesh segments, and (2) a task queue that stores the generated mesh triangles that need to be refined. Transactions protect accesses to the work queue. The usage of transactions in yada is similar in design to one presented in [66]. The operations on the task queue are complex and involve large reads and write sets which leads to a moderate amount of contention.

5.3 Application TM Characteristics

STAMP applications can be configured with different parameters to classify different workloads in order to represent several application domains and exercise a wide spectrum of transactional behaviors such as short or long transactions, different sizes of read and write sets, and varying degrees of contention. In all the experiments, we executed STAMP applications using the parameters suggested in the guidance notes supplied with the STAMP benchmark suite 0.9.10 distribution. Table 3 specifies those recommended workloads and highlights their transactional characteristics.

Table 3: STAMP workloads and their qualitative transactional characteristics [11]

Workload	Parameters	Tx Length	R/W Set	Contention
bayes	-v32 -r4096 -n10 -p40 -i2 -e8 -s1	Long	Large	High
genome	-g16384 -s64 -n16777216	Medium	Medium	Low
intruder	-a10 -l128 -n262144 -s1	Short	Medium	High
kmeans	-m40 -n40 -t0.00001 -i inputs/random-n65536-d32-c16.txt	Short	Small	Low
labyrinth	-i inputs/random-x512-y512-z7-n512.txt	Long	Large	High
ssca2	-s20 -i1.0 -u1.0 -l3 -p3	Short	Small	Low
vacation	-n2 -q90 -u98 -r1048576 -t4194304	Medium	Medium	Medium
yada	-a15 -i inputs/ttimeu1000000.2	Long	Large	Medium

Ten realistic complex benchmark configurations are executed for each application as shown in table 4. Two applications kmeans and vacation are executed with high and low data contention configurations. Each complex benchmark configuration is repeated five times and the averaged results are presented for our performance experimentation. From here onwards, experimentation is referred to by its configuration names introduced in table 4.

Table 4: Application configurations used in the evaluation [11]

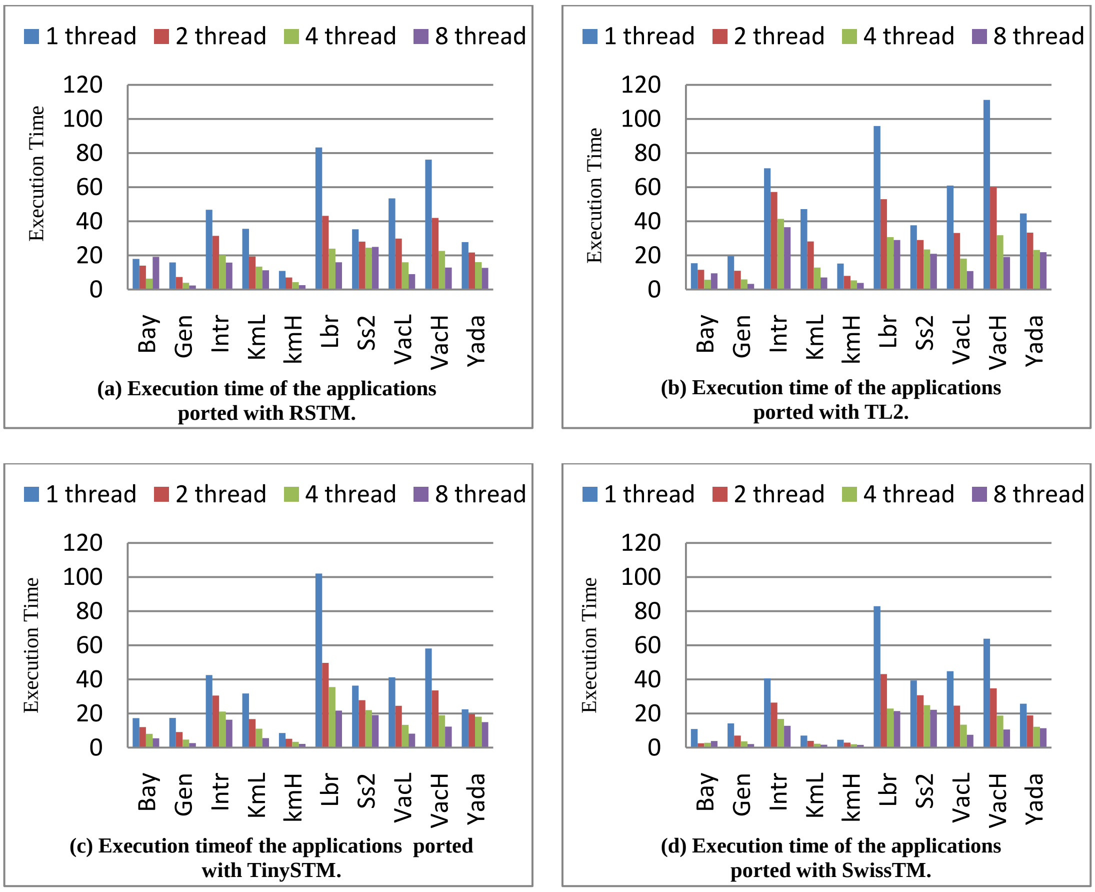
Configuration Name	Application	Configuration
Bay	Bayes	max_edges_learned_per_variable:8, edge_insert_penalty:2, max_number_of_parents:10, percent_chance_of_parent:40, number_of_records:4096, random_seed:1, number_of_variables:32
Gen	Genome	gene_length:16384, segment_length:64, number_of_segments:16777216
Intr	Intruder	percent_of_attacks:10, max_number_of_packets_per_stream:128, total_number_of_streams:262144, random_seed:1
KmL	Kmeans low contention	max_clusters:40, min_clusters:40, threshold:0.00001, input_file_name:inputs/random-n65536-d32-c16.txt
KmH	Kmeans high contention	max_clusters:15, min_clusters:15, threshold:0.00001, input_file_name:inputs/random-n65536-d32-c16.txt
Lbr	Labyrinth	input_file:inputs/random-x512-y512-z7-n512.txt
Ss2	Ssca2	probability_of_inter_clique:1.0, max_path_length:3, max_number_of_parallel_edges:3, problem_scale:20, probability_unidirectional:1.0
VacL	Vacation low contention	number_of_queries_per_task:2, %_of_relations_queried:90, number_possible_relations:1048576, %_of_user_tasks:98, number_of_tasks:4194304
VacH	Vacation high contention	number_of_queries_per_task:4, %_of_relations_queried:60, number_possible_relations:1048576, %_of_user_tasks:90, number_of_tasks:4194304
Yada	Yada	angle_constraint:15, file_prefix:inputs/ttimeu1000000.2,

CHAPTER 6: EMPIRICAL RESULTS

Our empirical results of all the four state-of-the-art STMs – RSTM[7], TL2[8], TinySTM[9], and SwissTM[10] – were measured on a computer with 2 quad-core processors (section 5.1). This chapter strives to make a quantitative comparison between the design peculiarities of these STM implementations. In this chapter, we highlight the performance tradeoffs embodied by these four different STM designs and present commonly used metrics to characterize TM applications. To capture execution data from the execution of the applications, we instrumented all these designs. The graphs of the experimental evaluation and analysis of the results is also presented in this chapter.

6.1 A First-Order Runtime Analysis

It is quite important to have an idea about the execution time of the STAMP's applications before starting to examine other TM behaviors. The time of execution is measured from an instant where multiple threads start executing transactions to an instant where they terminate executing transactions. Hence any setup and shut down time is excluded [13]. To allow direct comparison on our hardware platform, we measure the performance of ten variants of the STAMP applications by presenting the execution time. The time of execution is presented in order to show wellness of application scale with respect to the increasing number of threads. It also presents the measure with respect to effectiveness of the transactional execution of the applications. The execution time primarily depends on the characteristics of both i.e. the application and the TM implementation [67].



(a) Execution time of the applications ported with RSTM.

(b) Execution time of the applications ported with TL2.

(c) Execution timeof the applications ported with TinySTM.

(d) Execution time of the applications ported with SwissTM.

Figure 7: Performance comparison of the different STM systems for each application

Figure 7 presents how long, in seconds, the execution time of each of the target applications ported to execute under our four STM systems. Each application was run for 1, 2, 4 and 8 threads to provide experimental results. Its purpose was to check how fast an STM runs with respect to others by varying resource contention and scalability parameters. The execution time bar charts normalized to sequential execution with code that does not have an extra overhead. Thus, comparisons among the scalability of different applications are straightforwardly visible. Most of the applications have smaller runtimes as the number of threads rises. Figure 7 also shows that the Bay application does not scale well beyond four cores, when we ported it with RSTM, TL2, and SwissTM, since for larger number of processors, it degrades its performance.

Scalability [68] is the standard metric used to demonstrate how well applications execute with more processing resources. It is the ratio of total time to the number of threads used. It provides an important evaluation of the application's parallelism, and runtime system's efficiency of an STM. This scalability motivates the study of the metrics described below.

6.2 Analyzed Metrics

For this evaluation, we record characteristics in tabular forms and visualization charts to understand the behavior of an application that uses transactional memory because many transactional workloads contain heterogeneous transactions and different synchronization patterns throughout runtime execution. Since transactional memory is a scheme that imposes the committing or aborting of transaction sequences, the first issue while trying to monitor the transactional management is naturally the ratio of aborted transactions to committed transactions. Later in this work, other values are observed, such as the transaction retry rate in all four STMs.

6.2.1 Aborts per Commit (ApC)

In order to get better understanding of the transactional behavior of STAMP's applications we found ApC the most indicative metric. This criterion of performance comparison depicts the ratio of aborted transactions to committed transactions. It is a measure of wasted execution as granularity of a transaction corresponds directly to the time frame during which transactions maintains read, write possession along with the amount of work lost on a particular abort. This metric indicates the efficiency with which computing resources have been utilized as amount of work committed is assessed by the workload inputs. Architectural decisions can have vast influence and variations on the functions related to the transaction abortion.

Often transactions conflict with each other. In order to deal with conflicts, there exists a common approach which is to abort one of the conflicting transactions. A contention manager is resorted by the transactional framework for the resolution of conflicts. This conflict resolution depends on the policies of the contention manager as some transactions are more likely to be getting aborted than others e.g. preference may be given to smaller or larger transactions or to the transactions sharing smaller or higher resources etc.

The purpose of investigating it is that a good contention management should be able to reduce ApC in an application that exhibit repeat conflicts. For instance, if a contention manager aborts a long transaction and favors short transactions, this indicates that there may be a poor contention management. Examining and studying the application may correspond to the enhanced contention management policies. However it is very important as it assists in quantifying the effects of other associated characteristics on the whole execution of the application. For example, a workload containing those transactions which are highly contentious but in fact they are only in execution for short intervals. It may expose quite less actual contention than that of a workload which is comprised of less contentious transactions but occurs frequently.

In tables 5-14 we are presenting the TM behavior of ten variants of the STAMP applications. They include the number of transactions (Txns) and the number of aborts per commit (ApC). Each application was run for 2, 4 and 8 threads to provide ApC results. As expected, with a single thread ApC is zero. All the implementations show that a significant amount of work has been wasted by the aborted transactions.

Results also showed that the most suitable STM is SwissTm for Bay, Gen, Intr, KmL, KmH, VacL, VacH, and Yada as shown in table 5, 6, 7, 8, 9, 10, 13, and 14 respectively. Furthermore, it was observed that SwissTM is also suitable for K-means and Vacation even when ApC was set at high and low contention levels. However, there exists some difference in the calculated values of ApC for high and low contention as shown in the table 8, 9,10, and 13.The pattern in the difference suggested that for low contention levels the ApC levels were also low and as the contention levels were increased values of ApCs also tend to increase. In addition, for Ss2 application it was observed that TL2 STM is most suitable as presented in the table 11. For Lbr application, it was observed that by varying the scalability, suitability of STMs also vary i.e. for 2, 4 and 8 threads the most suitable STM are RSTM, TinySTM, SwissTM respectively as shown in the table 10. In general, it was observed that when we scale each application by 8 threads the most suitable STM tends to be SwissTM apart from Ss2 application. These results were inferred on the basis of calculating minimum ApC values for each application, as minimum ApC value corresponds to most suitable STM.

Results showed that the TinySTM was not suitable for application Gen, Intr, KmL, KmH, and Yada as shown in the table 6, 7, 8, 9, and 14. Furthermore for VacH and Ss2 the STMs which are not suitable are TL2 and RSTM respectively. These results were inferred on the basis of calculating maximum ApC values for each application, as maximum ApC values corresponds to least suitable STM.

Table 5: Transactional behaviors of Bay

Procs	RSTM		TL2		TinySTM		SwissTM	
	Txns	ApC	Txns	ApC	Txns	ApC	Txns	ApC
2	2411	0.006221	2621	0.006124	2520	0.006746	925	0.005405
4	2691	0.017466	2851	0.013197	2201	0.012267	913	0.009858
8	2252	0.059059	2973	0.044845	2417	0.056268	885	0.019209

Table 6: Transactional behaviors of Gen

Procs	RSTM		TL2		TinySTM		SwissTM	
	Txns	ApC	Txns	ApC	Txns	ApC	Txns	ApC
2	2489218	0.001	2494027	0.002	2489218	0.005793	2489220	0.000423
4	2489220	0.003	2500861	0.005	2489220	0.006575	2489220	0.002138
8	2489220	0.01	2514437	0.01	2489220	0.059450	2489220	0.003999

Table 7: Transactional behaviors of Intr

Procs	RSTM		TL2		TinySTM		SwissTM	
	Txns	ApC	Txns	ApC	Txns	ApC	Txns	ApC
2	23428127	0.05	33438023	0.30	23428127	0.67	23428127	0.02
4	23428129	0.29	47910526	0.51	23428129	2.48	23428129	0.06
8	23428133	1.74	59110273	0.60	23428133	5.68	23428133	0.24

Table 8: Transactional behaviors of KmL

Procs	RSTM		TL2		TinySTM		SwissTM	
	Txns	ApC	Txns	ApC	Txns	ApC	Txns	ApC
2	23428127	0.09	33438023	0.05	23428127	0.23	23428127	0.02
4	23428129	0.40	47910526	0.17	23428129	0.73	23428129	0.06
8	23428133	1.38	59110273	0.32	23428133	1.39	23428133	0.12

Table 9: Transactional behaviors of KmH

Procs	RSTM		TL2		TinySTM		SwissTM	
	Txns	ApC	Txns	ApC	Txns	ApC	Txns	ApC
2	23428127	0.42	33438023	0.18	23428127	0.82	23428127	0.06
4	23428129	1.62	47910526	0.39	23428129	2.33	23428129	0.16
8	23428133	4.47	59110273	0.58	23428133	6.53	23428133	0.40

Table 10: Transactional behaviors of Lbr

Procs	RSTM		TL2		TinySTM		SwissTM	
	Txns	ApC	Txns	ApC	Txns	ApC	Txns	ApC
2	1022	0.018591	1062	0.037665	1022	0.019569	1022	0.019569
4	1026	0.087719	1105	0.071493	1026	0.038986	1026	0.039961
8	1034	0.101547	1228	0.15798	1034	0.095745	1034	0.088975

Table 11: Transactional behaviors of Ss2

Procs	RSTM		TL2		TinySTM		SwissTM	
	Txns	ApC	Txns	ApC	Txns	ApC	Txns	ApC
2	22362283	5E-06	22362291	4E-07	22362283	5E-06	22362283	2E-06
4	22362285	2E-05	22362335	2E-06	22362285	1E-05	22362285	4E-06
8	22362295	8E-05	22362444	7E-06	22362293	3E-05	22362293	9E-06

Table 12: Transactional behaviors of VacL

Procs	RSTM		TL2		TinySTM		SwissTM	
	Txns	ApC	Txns	ApC	Txns	ApC	Txns	ApC
2	4194304	0.0005	4197789	0.00083	4194304	0.0006	4194304	4E-05
4	4194304	0.001	4204698	0.002472	4194304	0.0021	4194304	0.0001
8	4194304	0.004	4221569	0.006458	4194304	0.0071	4194304	0.0003

Table 13: Transactional behaviors of VacH

Procs	RSTM		TL2		TinySTM		SwissTM	
	Txns	ApC	Txns	ApC	Txns	ApC	Txns	ApC
2	4194304	0.001	4255578	0.014399	4194304	0.002712	4194304	1E-04
4	4194304	0.003	4382163	0.042869	4194304	0.01135	4194304	0.0003
8	4194304	0.01	4766462	0.120038	4194304	0.040623	4194304	0.0009

Table 14: Transactional behaviors of Yada

Procs	RSTM		TL2		TinySTM		SwissTM	
	Txns	ApC	Txns	ApC	Txns	ApC	Txns	ApC
2	23428127	0.25	33438023	0.424933	23428127	14.14	23428127	0.11
4	23428129	0.33	47910526	0.529542	23428129	29.70	23428129	0.22
8	23428133	0.36	59110273	0.517905	23428133	58.88	23428133	0.35

6.2.2 Transaction Retry Rate

STM systems normally support condition synchronization by the use of a retry mechanism [24]. In this mechanism, a transaction self aborts explicitly and in the process it also reschedules itself after detecting its precondition for the operation, which may not hold any longer. The set of location which have been read by retryer are then tracked by the runtime, and at the same time it also refrain these set of locations from getting rescheduled, until at least any one of the location in the set is modified by any other transaction.

Classifying the total number of retries can give input to an STM regarding the retry policies. Furthermore, it also can give input to the programmers on data layouts of application that may be the reason of conflicts related to synchronization. In Figure 8, we could easily find that the number of retries grow significantly with the number of threads, e.g. for TinySTM, when we are running 8 threads, there are 7 times of the number of retries than that of 2 threads. Retries in large numbers indicate that the application lacks inherent concurrency, or may be certain design choices of STM are causing unrequited conflicts. In order to differentiate between these stated cases, numbers are collected by implementing different STMs which highlight about three concerns that are related to the scalability of TM:

- If an application is not scaled even by having a low retry rate, it indicates the cause is not likely associated to synchronization but may be with other causes like cache performance or data layout etc.

- The designer of TM runtime should pay extra attention to the runtime behavior specifically for transaction aborts. Even though the proportion of retries has not much significance now, it may have a high growth rate if it is run on more cores by utilizing numerous threads.

- The programmers should also get aware of the fact that the prospect of retries is rapidly increasing when there are more threads. For this reason programmers should incorporate more efforts in decreasing the overhead of abort and thus memory accesses in the transactions will also be decreased.

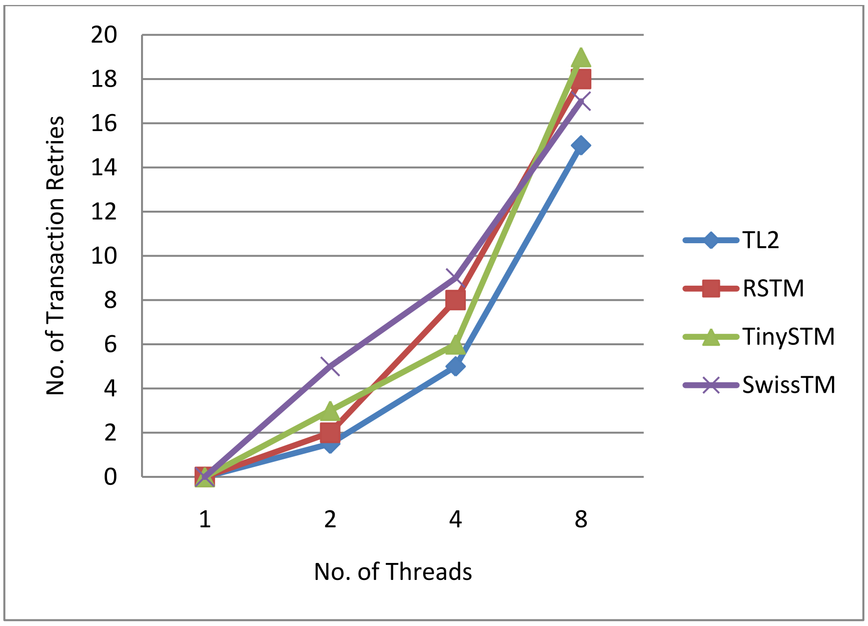


Figure 8: No. of retries in Bayes for different STM implementations. Lower is better.

6.3 **Validity Threats**

There are some potential facts that limit the outcomes of this study. The following sections explain the validity threats related to the experimentation.

6.3.1 Internal Validity

There are many types of STM systems, benchmarks and metrics. We have subjectively chosen four STM systems and three metrics for the experimental purposes. Similarly, benchmarks are also subjectively chosen to exploit the level of concurrency of the underlying STM implementation. Hence threat of not using other systems, benchmarks and metrics cannot be ruled out for this study.

We performed our experimentation up to 8 cores for the reason that our chosen STM implementation and STAMP applications did not scale well beyond 8 cores. We analyzed this scalability problem could be due to the following reasons:

- Lack of concurrency in the application – we use STAMP applications
- The implementation of an STM – we use RSTM, TL2, TinySTM, and SwissTM
- The system architecture – we use a computer with 2 quad-core processors in total 8 cores.

6.3.2 External Validity

Since, all the experiments were run on one machine due to limited resources. The overall results of the experiment cannot be generalized for other machines hence results of the current study are only generalizable to limited extent for 2 quad-core processors machine.

CHAPTER 7: DISCUSSION AND RELATED WORK

In the current study, the non-trivial TM applications such as STAMP [11] have been analyzed keeping the transactional behavior in context. Each application was used to show the scalability of four STM implementations, and the metrics which were presented are commonly used to characterize TM applications. Scalability is one of the classic metrics which is utilized to present how well an application executes with more processing resources. Scalability is the ratio of total time to total number of threads used. It is a key measure of the application's parallelism, and an STM runtime system's efficiency. Thus, comparisons among the scalability of different applications showed most of the applications had smaller runtimes as the number of threads raised.

One of the key features of applications is that ApC can assist in determining the transactional scalability of an application. Specifically in a scenario when distinct threads are involved in constantly updating the same variables, there exists no such method of making critical sections parallel, hence institutively it is anticipated that it will have large number of aborts. In the case, where transactions may abort several times, there may be a need of dynamically tuning the runtime system to restrain from excessive aborts. All of the four implementations showed that there was a significant amount of work which was wasted by the aborted transactions. Another useful metric which we chose in order to evaluate the performance of STM was transaction retry overhead. Retries in large numbers indicate that an application lacks inherent concurrency. Our work showed that retries rapidly increased as the number of threads were raised.

As transactional memory is an emerging research area, little work has been done to understand the performance tradeoffs of different implementations of STM.

Perfumo et al. in [13] performed execution characterizations of Haskell TM benchmarks; however the metrics presented were different as discussed in this thesis. Furthermore, non-trivial TM applications were also not considered in this paper. However this thesis provides a detailed study of non-trivial TM applications.

Ansari in [67] profiled the execution of applications against DSTM2. This profiling and its relation to the performance was applied to several popular non-trivial TM applications, such as STAMP [2] applications i.e. Genome, KMeans, and Vacation. The aim of this profiling was to get better understanding of the factors which may have any impact on the overall performance. Statistical data presented in this thesis had similar goals as of Ansari's work. In this thesis 8 non-trivial applications have been ported across 4 STM implementations to generate execution characteristics of applications.

Chung et al. in [69] presented a comprehensive study which looked into 35 distinct TM applications in order to identify some key features that are related to transactional behavior. Wealth of data was provided by evaluating the performance with respect to nested transaction depth, transactions size, sizes of readset and writeset etc. They did not evaluated non-trivial application TM applications which have been studied in this thesis, and they also did not generate any execution characteristics as presented here.

CONCLUSION

In this thesis an attempt was made to study real world and complex TM applications i.e. STAMP benchmark suite in order to characterize their transactional behaviors with STM systems. In this report we have reported some figures associated with the performance in conjunction with execution characteristics for the studied TM applications. The study has been performed on four state-of-the-art STM implementations – RSTM, TL2, TinySTM, SwissTM – to obtain execution data, and metrics to examine the execution characteristics of these TM applications up to maximum 8 threads. These characteristics associated with the execution provide key insights into the design of efficient STM systems for both synchronization and parallelization. This thesis study has navigated through the metrics to understand the observed scalability.

This dissertation makes the following observations:

- Transactional Memory is discussed in general with main focus on Software Transactional Memory. TM’s potential advantages are defined along with its different design alternatives.

- An overview of four different STM implementations encompassing RSTM, TL2, TinySTM and SwissTM is presented.

- Performance comparison of four STM implementations is conducted on Execution Time, Aborts per Commit and Transaction Retry Rate parameters.

- The SwissTM performs better than RSTM, TL2 and TinySTM while measuring execution time and Aborts per Commit parameters.

- The TL2 performs better than RSTM, TinySTM and SwissTM while measuring Transaction Retry Rate parameter.

By the comparison study done as part of this master thesis it can be summarized that,

- Additional cores and atomic safety tend to provide application with much more increased and promising performance. However the overhead related to the TM management may also be increased.

- It is hard to build an STM system that can perform consistently well in all kind of situations.

FUTURE WORK

Transactional Memory is a lock free data-structure specifically designed to run on parallel architectures. In this research work we emphasized on performance tradeoffs among different Software Transactional Memory implementations. We have discussed in detail all our chosen STM implementations, their potential strengths and weaknesses. Although we have studied these systems qualitatively and quantitatively, which has helped us in coming to a conclusion which STM implementation performs better than others, on the basis of our selected performance metrics. But there is always a room for more research and better understanding from a different view point, in a research work. As far as we consider, this study can further be extended and researched in the future in the following ways:

- We compared performance tradeoffs of four software transactional memory systems namely RSTM, TL2, TinySTM, and SwissTM. All these systems were implemented using C/C++ language. This research could further be extended by investigating some other software transactional memory systems that are built by using other languages like Java, Haskell, C# etc.

- The metric selection is critical while performing performance tradeoffs, we chose execution time, aborts per commit, and transaction retry rate. This could be interesting if someone chose other metrics and perform performance comparisons. By doing this, one can reach onto new conclusions that can find and provide help in understanding software transactional memory systems from a different perspective.

- We run all our selected software transactional memory systems on the STAMP Benchmark suit along with the chosen performance metrics. Many benchmarks have been created by researchers and industry practitioners in order to evaluate the parallel systems. Some of the benchmarks which have emerged are NBP OpenMP [70], and SPEComp [71]. TM systems and performance metrics can be evaluated by running on these benchmarks to see the strengths, and shortcomings of them.

- We conducted all our experiments up to 8 cores. This experiment can further be conducted on more than 8 cores, to see the behavior of these STM systems while executing on more cores.

- Finally, more future research work is required to cope with the issues such as compatibility of present system architecture with the STM to fulfill the needs of transactional management. It is mandatory to have such STMs which are applicable in many cores.

REFERENCES

[1] A.-R. Adl-Tabatabai, *et al.*, "Unlocking Concurrency," *Queue,* vol. 4, pp. 24-33, 2007.

[2] K. Fraser and T. Harris, "Concurrent programming without locks," *ACM Trans. Comput. Syst.*, vol. 25, p. 5, 2007.

[3] J. R. Larus and R. Rajwar, *Transactional memory*, 1st ed. [San Rafael, Calif.]: Morgan & Claypool, 2007.

[4] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," *SIGARCH Comput. Archit. News*, vol. 21, pp. 289-300, 1993.

[5] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, Ottowa, Ontario, Canada, 1995, pp. 204-213.

[6] P. Damron, *et al.*, "Hybrid transactional memory," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, San Jose, California, USA, 2006, pp. 336-346.

[7] V. J. Marathe, *et al.*, "Lowering the Overhead of Software Transactional Memory," in *ACM SIGPLAN Workshop on Transactional Computing*, Held in conjunction with PLDI 2006. Expanded version available as TR 893, Department of Computer Science, University of Rochester, March 2006.

[8] D. Dice, *et al.*, "Transactional Locking II," in *Proceedings of the 20th International Symposium on Distributed Computing*, Stockholm, Sweden, 2006, pp. 194-208.

[9] P. Felber, *et al.*, "Dynamic performance tuning of word-based software transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, Salt Lake City, UT, USA, 2008, pp. 237-246.

[10] A. Dragojevi, *et al.*, "Stretching transactional memory," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, Dublin, Ireland, 2009, pp. 155-165.

[11] M. Chi Cao, *et al.*, "STAMP: Stanford Transactional Applications for Multi-Processing," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, 2008, pp. 35-46.

[12] V. J. Marathe, *et al.*, "Design tradeoffs in modern software transactional memory systems," in *Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, Houston, Texas, 2004, pp. 1-7.

[13] C. Perfumo, *et al.*, "The limits of software transactional memory (STM): dissecting Haskell STM applications on a many-core environment," in *Proceedings of the 5th conference on Computing frontiers*, Ischia, Italy, 2008, pp. 67-78.

[14] J. Lourenco, *et al.*, "Understanding the behavior of transactional memory applications," in *Proceedings of the 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, Chicago, Illinois, 2009, pp. 1-9.

[15] G. E. Moore, "Cramming more components onto integrated circuits," in *Readings in computer architecture*, ed: Morgan Kaufmann Publishers Inc., 2000, pp. 56-59.

[16] A. S. Grove, *Only the paranoid survive! : the threat and promise of strategic inflection points*, 1st ed ed. New York: Currency Doubleday, 1996.

[17] H. Sutter and J. Larus, "Software and the Concurrency Revolution," *Queue,* vol. 3, pp. 54-62, 2005.

[18] M. Olszewski, "A Dynamic Instrumentation Approach to Software Transactional Memory " Master's Thesis, Department of Electrical and Computer Engineering, University of Toronto, October, 2007.

[19] C. S. Ananian, *et al.*, "Unbounded Transactional Memory," *IEEE Micro,* vol. 26, pp. 59-69, 2006.

[20] K. Olukotun and L. Hammond, "The Future of Microprocessors," *Queue*, vol. 3, pp. 26-29, 2005.

[21] L. Ceze, *et al.*, "Bulk Disambiguation of Speculative Threads in Multiprocessors," in *Proceedings of the 33rd annual international symposium on Computer Architecture*, 2006, pp. 227-238.

[22] R. Rajwar, *et al.*, "Virtualizing Transactional Memory," in *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005, pp. 494-505.

[23] I. William N. Scherer and M. L. Scott, "Advanced contention management for dynamic software transactional memory," in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, Las Vegas, NV, USA, 2005, pp. 240-248.

[24] T. Harris, *et al.*, "Composable memory transactions," *Commun. ACM*, vol. 51, pp. 91-100, 2008.

[25] B. Saha, *et al.*, "McRT-STM: a high performance software transactional memory system for a multi-core runtime," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, New York, USA, 2006, pp. 187-197.

[26] D. J. Scales, *et al.*, "Shasta: a low overhead, software-only approach for supporting fine-grain shared memory," in *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, Cambridge, Massachusetts, United States, 1996, pp. 174-185.

[27] C. C. Minh, *et al.*, "An effective hybrid transactional memory system with strong isolation guarantees," in *Proceedings of the 34th annual international symposium on Computer architecture*, San Diego, California, USA, 2007, pp. 69-80.

[28] A. Shriraman, *et al.*, "An integrated hardware-software approach to flexible transactional memory," in *Proceedings of the 34th annual international symposium on Computer architecture*, San Diego, California, USA, 2007, pp. 104-115.

[29] F. Tabba, *et al.*, "NZTM: nonblocking zero-indirection transactional memory," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, Calgary, AB, Canada, 2009, pp. 204-213.

[30] S. Lie, "Hardware Support for Unbounded Transactional Memory," Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 2004.

[31] S. Tomic, *et al.*, "Hardware Transactional Memory with Operating System Support, HTMOS," in *Parallel Processing*, ed, 2008, pp. 8-17.

[32] V. J. Marathe and M. L. Scott, "A qualitative survey of modern software transactional memory systems," Technical Report TR 839, Department of Computer Science, University of Rochester, June, 2004.

[33] P. Felber, *et al.*, "Transactions are back---but are they the same?," *SIGACT News*, vol. 39, pp. 48-58, 2008.

[34] S. Classen, "LibSTM: A fast and flexible STM Library," Master's Thesis, Laboratory for Software Technology, Swiss Federal Institute of Technology, ETH Zurich, Feb, 2008.

[35] W. N. S. III and M. L. Scott, "Contention Management in Dynamic Software Transactional Memory," in *Proceedings of the ACM PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, NL, Canada, July, 2004.

[36] R. Guerraoui, *et al.*, "Toward a theory of transactional contention managers," in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, Las Vegas, NV, USA, 2005, pp. 258-264.

[37] M. L. Scott. *Applications Included with RSTM Web Page:* http://www.cs.rochester.edu/research/synchronization/rstm/applications.shtml.

[38] C. B. Seaman, "Qualitative Methods in Empirical Studies of Software Engineering," *IEEE Trans. Softw. Eng.*, vol. 25, pp. 557-572, 1999.

[39] J. W. Creswell, *Research design : qualitative, quantitative, and mixed methods approaches*, 2. ed. Thousand Oaks: Sage, 2003.

[40] T. Harris and K. Fraser, "Language support for lightweight transactions," *SIGPLAN Not.*, vol. 38, pp. 388-402, 2003.

[41] K. Fraser, "Practical lock freedom," Ph. D. dissertation, Computer Laboratory, University of Cambridge, Feb, 2004.

[42] M. Herlihy, *et al.*, "A flexible framework for implementing software transactional memory," presented at the Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, Portland, Oregon, USA, 2006.

[43] R. Guerraoui, *et al.*, "Polymorphic Contention Management in SXM," in *DISC '05: Proceedings of the nineteenth International Symposium on Distributed Computing*, Cracow, Poland, Sep, 2005, pp. 303-323.

[44] J. E. Gottschlich and D. A. Connors, "DracoSTM: a practical C++ approach to software transactional memory," in *Proceedings of the 2007 Symposium on Library-Centric Software Design*, Montreal, Canada, 2007, pp. 52-66.

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

[46] K. P. Eswaran, *et al.*, "The notions of consistency and predicate locks in a database system," *Commun. ACM*, vol. 19, pp. 624-633, 1976.

[47] M. Herlihy, *et al.*, "Software transactional memory for dynamic-sized data structures," in *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, Boston, Massachusetts, 2003, pp. 92-101.

[48] A. Dragojevic, *et al.*, "Why STM can be more than a Research Toy," Technical Report # LPD-REPORT-2009-003, University of Neuchatel, Switzerland,2009.

[49] V. J. Marathe, *et al.*, "Adaptive Software Transactional Memory," in *Proceedings of the 19th International Symposium on Distributed Computing*, Cracow, Poland, Sep, 2005, pp. 354–368.

[50] T. Riegel, *et al.*, "A Lazy Snapshot Algorithm with Eager Validation," ed, 2006, pp. 284-298.

[51] L. Yossi , *et al.*, "Anatomy of a Scalable Software Transactional Memory," in *TRANSACT '09: 4th Workshop on Transactional Computing*, Feb, 2009.

[52] T. Riegel, *et al.*, "Time-based transactional memory with scalable time bases," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, San Diego, California, USA, 2007, pp. 221-228.

[53] M. F. Spear, *et al.*, "Conflict Detection and Validation Strategies for SoftwareTransactional Memory," in *Proceedings of the Twentieth International Symposium on Distributed Computing*, Sep, 2006, pp. 179 - 193.

[54] C. Kotselidis, *et al.*, "Investigating software Transactional Memory on clusters," in *IPDPS '08: IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1-6.

[55] D. M. Chickering, *et al.*, "A Bayesian approach to learning Bayesian networks with local structure," in *UAI '97: Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence*, pp. 80-89.

[56] A. Moore and M. S. Lee, "Cached Sufficient Statistics for Efficient Machine Learning with Large Datasets," *Journal of Artificial Intelligence Research*, vol. 8, pp. 67-91, March, 1998.

[57] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J. Res. Dev.*, vol. 31, pp. 249-260, 1987.

[58] B. Haagdorens, *et al.*, "Improving the Performance of Signature-Based Network Intrusion Detection Sensors by Multi-threading," in *Proceedings of 5th International Workshop on Information Security Applications*, Jeju Island, Korea, August, 2004, pp. 188-203.

[59] R. Narayanan, *et al.*, "MineBench: A Benchmark Suite for Data Mining Workloads," *IEEE International Symposium on Workload Characterization (IISWC)*, pp. 182-188, Oct, 2006.

[60] J. C. Bezdek, *Pattern Recognition with Fuzzy Objective Function Algorithms*: Kluwer Academic Publishers, 1981.

[61] C. Y. Lee, "An Algorithm for Path Connections and its Applications," *IRE Transactions on Electronic Computers*, vol. EC-10, pp. 346-365, September, 1961.

[62] I. Watson, *et al.*, "A Study of a Transactional Parallel Routing Algorithm," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, 2007, pp. 388-398.

[63] D. Bader and K. Madduri, "Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors," in *HIPC '05: Proceedings of 12th International Conference on High Performance Computing*, Goa, India, Dec, 2005, pp. 465-476.

[64] http://www.spec.org/jbb2000/index.html. *Standard Performance Evaluation Corporation, SPECjbb2000 Java Business Benchmark.*

[65] J. Ruppert, "A Delaunay refinement algorithm for quality 2-dimensional mesh generation," *J. Algorithms*, vol. 18, pp. 548-585, 1995.

[66] M. Kulkarni, *et al.*, "Using Transactions in Delaunay Mesh Generation," in *Proceedings of the Workshop on Transactional Memory Workloads*, Ottawa, Canada, June, 2006, pp. 23-31.

[67] M. Ansari, *et al.*, "Profiling Transactional Memory Applications," in *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Washington, DC, 2009, pp. 11-20.

[68] M. Ansari, *et al.*, "On the Characterisation of Complex Transactional Memory Applications," Preprint Series, CSPP-44, School of Computer Science, The University of Manchester, March, 2008.

[69] J. C. Chung, Hassan. Minh , Chi Cao. Mcdonald , Austen. Carlstrom , Brian D. Kozyrakis , Christos. Olukotun, Kunle, "The Common Case Transactional Behavior of Multithreaded Programs," in *Proceedings of the 12th International Conference on High-Performance Computer Architecture*, 2006.

[70] A. Marowka, "Performance of OpenMP Benchmarks on Multicore Processors," in *Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing*, Agia Napa, Cyprus, 2008, pp. 208-219.

[71] V. Aslot, *et al.*, "SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance," in *Proceedings of the International Workshop on OpenMP Applications and Tools: OpenMP Shared Memory Parallel Programming*, 2001, pp. 1-10.