ByteSTM: Java Software Transactional Memory at the Virtual Machine Level

Mohamed Mohamedin

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Binoy Ravindran, Chair
Leyla Nazhandali
Mohamed Rizk
Paul E. Plassmann

February 8, 2012
Blacksburg, Virginia

Keywords: Software Transactional Memory, Virtual Machine, Java, Multiprocessor, Concurrency, Synchronization

ByteSTM: Java Software Transactional Memory at the Virtual Machine Level

Mohamed Mohamedin

(ABSTRACT)

As chip vendors are increasingly manufacturing a new generation of multi-processor chips called multicores, improving software performance requires exposing greater concurrency in software. Since code that must be run sequentially is often due to the need for synchronization, the synchronization abstraction has a significant effect on program performance. Lock-based synchronization – the most widely used synchronization method – suffers from programability, scalability, and composability challenges.

Transactional memory (TM) is an emerging synchronization abstraction that promises to alleviate the difficulties with lock-based synchronization. With TM, code that read/write shared memory objects is organized as transactions, which speculatively execute. When two transactions conflict (e.g., read/write, write/write), one of them is aborted, while the other commits, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling-back changes made to objects. In addition to a simple programming model, TM provides performance comparable to lock-based synchronization. Software transactional memory (STM) implements TM entirely in software, without any special hardware support, and is usually implemented as a library, or supported by a compiler or by a virtual machine.

In this thesis, we present *ByteSTM*, a virtual machine-level Java STM implementation. ByteSTM implements two STM algorithms, TL2 and RingSTM, and transparently supports implicit transactions. Program bytecode is automatically modified to support transactions: memory load/store bytecode instructions automatically switch to transactional mode when a transaction starts, and switch back to normal mode when the transaction successfully commits. Being implemented at the VM-level, it accesses memory directly and uses absolute memory addresses to uniformly handle memory. Moreover, it avoids Java garbage collection (which has a negative impact on STM performance), by manually allocating and recycling memory for transactional metadata. ByteSTM uses field-based granularity, and uses the thread header to store transactional metadata, instead of the slower Java ThreadLocal abstraction.

We conducted experimental studies comparing ByteSTM with other state-of-the-art Java STMs including Deuce, ObjectFabric, Multiverse, DSTM2, and JVSTM on a set of micro-benchmarks and macro-benchmarks. Our results reveal that, ByteSTM's transactional throughput improvement over competitors ranges from 20% to 75% on micro-benchmarks and from 36% to 100% on macro-benchmarks.

Dedication

To my wife.

Without her love, care and support, I could not have completed this thesis

To my parents.

Without their love, trust and continued support, I could not have made their dream true

Acknowledgments

First, I would like to thank my advisor, Dr. Binoy Ravindran, for his continues support, help, guidance, encouragement, and trust. He believed in me more that I believed in myself. I really appreciate all his endless efforts.

I would also like to thank Dr. Leyla Nazhandali, Dr. Mohamed Rizk, and Dr. Paul E. Plassmann for serving on my committee and providing me with their valuable suggestions and feedback.

Second, I would like to thank Dr. Sedki Riad and VT-MENA program administration for their support.

Finally, I would like to thank my wife, my parents, my brother, and all my family and friends. Without their love and support, I could not have completed this thesis

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer architecture is undergoing a paradigm shift. Chip manufacturers are increasingly investing in, and manufacturing a new generation of multi-processor chips called multicores, as it is becoming increasingly difficult to enhance clock speeds by packing more transistors in the same chip without increasing power consumption and overheating. Consequently, application software performance can no longer be improved by relying on increased clock speeds of single processors; software must explicitly be written to exploit the hardware parallelism of multiprocessors.

Amdahl's law [6] specifies the maximum possible speedup that can be obtained when a sequential program is parallelized. Informally, the law states that, when a sequential program is parallelized, the relationship between the speedup reduction and the sequential part (i.e., sequential execution time) of the parallel program is non-linear. Formally, if p is the fraction of the program that runs in parallel, and n is the number of processors, then the speedup upper bound is given by:

$$Speedup = \frac{1}{(1-p) + \frac{p}{n}}$$

Thus, on an 8-processor machine, if 90% of the program runs in parallel, and 10% runs sequentially, the overall speedup is 4.7× and not 8× (i.e., the utilization of the machine is cut almost in half). If 95% of the program runs in parallel, and 5% runs sequentially, the overall speedup is still 5.9×. Now, if 99% of the program runs in parallel, and 1% runs sequentially, then overall speedup increases to 7.47×. Thus, Amdahl's law indicates that, the sequential fraction of the (parallelized) program has a significant impact on overall performance.

Code that must be run sequentially in a parallel program is often due to the need for coordination and synchronization (e.g., shared data structures that must be executed mutual exclusively to avoid race conditions). Per Amdahl's law, this implies that synchronization abstractions have a significant effect on performance.

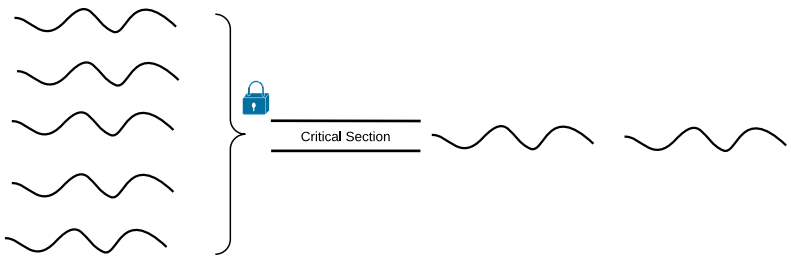


Figure 1.1: Locks serialize access to critical sections.

Lock-based synchronization is the most widely used synchronization abstraction. Coarse-grained locking (e.g., a single lock guarding a critical section) is simple to use, but results in significant sequential execution time: the lock simply forces parallel threads to execute the critical section sequentially, in a one-at-a-time order (see Figure 1.1). With fine-grained locking, a single critical section now becomes multiple shorter critical sections. This reduces the probability that all threads will need the same critical section at the same time, permitting greater concurrency. However, this has low programmability: programmers must acquire only necessary and sufficient locks to obtain maximum concurrency without compromising safety, and must avoid deadlocks when acquiring multiple locks. Moreover, locks can lead to livelocks, lock-convoying, and priority inversion. Perhaps, the most significant limitation of lock-based code is their non-composability. For example, atomically moving an element from one hash table to another using those tables' (lock-based) atomic methods is not possible in a straightforward manner: if the methods internally use locks, a thread cannot simultaneously acquire and hold the locks of the methods (of the two tables); if the methods were to export their locks, that will compromise safety.

Lock-free synchronization [46], which uses atomic hardware synchronization primitives (e.g., Compare And Swap [50, 51], Load-Linked/Store-Conditional [87]), also permits greater concurrency, but has even lower programmability: lock-free algorithms must be custom-designed for each situation (e.g., a data structure [67, 61, 29, 47, 15]), and reasoning about their correctness is significantly difficult [46].

1.1 The Case for Transactional Memory

Transactional Memory (TM) is an alternative synchronization abstraction that promises to alleviate the difficulties with lock-based synchronization. With TM, code that read/write shared memory objects is organized as *memory transactions*, which optimistically execute, while logging changes made to objects–e.g., using an undo-log or a write-buffer. Two transactions conflict if they access the same object and one access is a write. When that happens,

a contention manager resolves the conflict by aborting one and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling-back the changes–e.g., undoing object changes using the undo-log (eager), or discarding the write buffers (lazy). In addition to a simple programming model (locks are excluded from the programming interface), TM provides performance comparable to lock-based synchronization [83] and is composable.

TM's first implementation was proposed in hardware, called hardware transactional memory (or HTM) [45]. A typical technique used in HTMs is to modify the hardware cache coherence protocol to support transactions. The idea is to use the cache as a write-set, which buffers the transaction's changes. If a location is modified by another processor's transaction, which causes a conflict, then the cache coherence protocol will detect the conflict and the transaction aborts itself. When a transaction commits successfully, it marks the associated cache line as committed. This ensures that the cache line will be written back to the memory when replaced. If a transaction is aborted, then all its changes are discarded by marking the modified cache lines as invalid. HTM has the lowest overhead, but HTM transactions are usually limited in space and time. A transaction is limited to the size of the cache, which has the HTM logic, and to the scheduler time slice. Examples of HTMs include TCC [39], UTM [7], Oklahoma [92], ASF [21], and Bulk [19].

TM implementation in software, called software transactional memory (or STM) was proposed later [86]. A typical technique used in STMs is to log all writes in a redo (memory) log, and track all reads in a read-set. Each read validates against other concurrent transactions by verifying that its read-set is still valid and other transactions did not write to a location after it was read. At commit time, the read-set is validated again. If the transaction is valid, the redo log entries are locked and the values are written-back to the memory. If the commit fails, then the transaction aborts, and the redo log is discarded. The transaction is subsequently restarted. STM transactions do not need any special hardware, are not limited in size or time, and are more flexible. However, STM has a higher overhead, and thus lower performance, than HTM. Examples of STMs include RSTM [98], TinySTM [80], Deuce [55], and AtomJava [49]. Figure 1.2 shows an example STM code. The example uses the `atomic` keyword, which implicitly creates a transaction for the enclosed code block.

Hybrid TM (or HyTM) was subsequently proposed in [58], which combines HTM with STM, and avoids their limitations. A typical technique used in HyTMs is to run a transaction using HTM. If the transaction fails due to HTM size and time limitations, then it is restarted as an STM transaction. The HTM system is usually modified so that it can detect conflicts with STM transactions. An STM's write replaces the original value with a flag value and buffers the write. An HTM read checks if the read value equals the flag. If so, the HTM transaction is aborted. An HTM write checks that no active STM transaction has read the location. A *reader pointer* is associated with every location read by an STM transaction. If this pointer is null, then no STM transaction had read it. Otherwise (i.e., an active STM transaction has read the location), then the HTM transaction is aborted. Examples of HyTMs include SpHT [57], VTM [76], HyTM [24], LogTM [68], and LogTM-SE [102].

Thread A

```
1 atomic{
2   for (int i=0; i<1000; i++)
3       counter++;
4 }
```

Thread B

```
1 atomic{
2   for (int i=0; i<1000; i++)
3       counter++;
4 }
```

Figure 1.2: Example of implicit transaction language support. If `counter` is initialized to zero, the final value will be 2000.

1.2 Limitations of STM Implementations

Given the hardware-independence of STM, which is a compelling advantage, we focus on STM and improving its performance.

STM can be classified into three categories: *library-based*, *compiler-based*, and *virtual machine-based*. Library-based STMs add transactional support without changing the underlying language. Library-based STMs can be further classified into two categories: those that use *explicit* transactions [43, 33, 72, 99] and those that use *implicit* transactions [16, 81, 55]. Explicit transactions are difficult to use and support only transactional objects. Thus, external libraries cannot work with explicit transactions. Implicit transactions, on the other hand, use modern language features (e.g., Java annotations [94]) to mark sections of the code as atomic (the example in Figure 1.2 illustrates a similar code). Instrumentation is used to add transactional code transparently to the atomic sections. Some implicit transactions work only with transactional objects [16, 81], while others work on any object and support external libraries [55].

Compiler-based STMs support implicit transactions transparently by adding new language constructs (e.g., `atomic`). Then the compiler generates the transactional code that calls the underlying STM library. Compiler-based STMs can optimize the generated code and do overall program optimization. On the other hand, compilers need the source code to support external libraries. With managed run-time languages (i.e., those supported by a VM), compilers alone do not have full control over the VM. Thus, the generated code will not be optimized and may contradict with some of the VM features like the garbage collector. Examples from this category include the Intel C++ STM compiler [52] and AtomJava [49].

Virtual machine-based STMs have not been heavily studied. The only works that we are aware of include [40, 18, 100, 2]. In [40], STM is implemented in C inside the JVM to get benefits of the VM-managed environment. This STM uses an algorithm that does not support the opacity [37] correctness property, which requires that 1) committed transactions must appear to execute sequentially in real-time order, 2) no transaction observes modifications to shared state done by aborted or live transactions, and 3) all transactions, including aborted and live ones, must observe a consistent state. Since opacity is not ensured, inconsistent reads will occur before a transaction is aborted. Thus, if this algorithm is used in

an unmanaged environment, then inconsistent reads can damage the memory and crash the application. Moreover, being one of the earliest VM-based STM works, no comparison with other STMs was made in [40].

[18] presents a new programming language based on Java, called Atomos, and a VM to run it. Standard Java synchronization (i.e., `synchronized` block, and `wait/notify` conditional variables) is replaced with transactions. However, transactional support in Atomos is based on HTM.

Library-based STMs are largely based on the premise that it is better not to modify the VM or the compiler, to promote flexibility, backward compatibility with legacy systems, and easiness to deploy and use. However, this premise is increasingly violated as many library-based STMs require some VM support or are being directly integrated into the language and thus the VM. Most modern STM libraries are based on annotations and instrumentation, which are new features in the Java language. For example, the Deuce STM library [55] is based on a non-standard proprietary API (i.e., `sun.misc.Unsafe`), which makes it incompatible with other JVM versions, or with other JVMs that do not support that API (e.g., JikesRVM [5]). Moreover, programming languages routinely add new features in their evolution for a whole host of reasons. For example, Java 5 [93] added many new language features such as generics, enumerations, annotations, for-each, variable arguments, etc. Thus, it is only natural that STM will be integrated into the language, and thus the VM, as part of the language evolution process.[1]

Implementing STM at the VM level allows many opportunities for optimization and adding new features. For example, the VM has direct access to the memory, allowing it to directly write to the memory and calculate absolute addresses. This means that, writing values back to memory is easier, faster, and can be done without using reflection (which usually negatively affects performance). The VM also has full control of the garbage collector (GC). This means that, the potentially degrading effect of GC on STM performance can be controlled or even eliminated by manually allocating and recycling the memory needed for transactional support. Furthermore, the VM has control on each bytecode instruction executed. This means that, the behavior of bytecode instructions can be changed based on whether they executed transactionally (i.e., inside a transaction) or non-transactionally. Also, the VM has access to each object's header and thread's header. This means that, a transaction's metadata can be added to the object headers and/or the thread header transparently. Accessing a thread's header is much faster than using Java's `ThreadLocal` abstraction.

Moreover, if transactional support is based on HTM (as in [18]), then VM is the only suitable place to add HTM-based transactional support. Otherwise, the GC will abort transactions when it interrupts them, and Java synchronization semantics will contradict with transactional semantics. Also, with HTM, the memory allocator uses a centralized data structure, which increases the number of conflicts, resulting in degraded performance [14].

[1] Of course, this is assuming that STM gains increasing traction among programmers. However, we base this claim on the increasingly successful application case studies with STM [31, 60, 77, 48].

1.3 Thesis Contribution

Motivated by these observations, we design and implement a VM-level STM: *ByteSTM*. ByteSTM supports two algorithms: TL2 [25] and RingSTM [91]. ByteSTM writes directly to the memory without reflection (such as [43]) or using a non-standard library (such as [55]). It uses absolute memory address to unify memory handling of instant objects, static objects, and array elements. ByteSTM uniformly handles all variable types, using the address and number of bytes as an abstraction. It eliminates the GC overhead, by manually allocating and recycling memory for transactional metadata. ByteSTM uses field-based granularity, which scales better than object-based or word-based granularity, and has no false conflicts due to field-level granularity.[2]

ByteSTM supports implicit transactions in two modes. The first mode, called the compiler mode, requires compiler support and works with the `atomic` new keyword. The second mode, called the direct mode, works with current standard Java compilers by calling a static method to start a transaction (i.e., `xBegin`) and another one to end the transaction (i.e., `xCommit`). At the VM level, a thread header is used to hold thread-local transactional metadata (e.g., transactional flag, write-set, read-set, etc.). As mentioned before, accessing the thread header is faster than using the standard Java `ThreadLocal` abstraction.

When a transaction starts, the transactional flag is set, and all reads and writes become transactional. Moreover, any method that is called while the transactional flag is set, executes transactionally. Thus, ByteSTM supports external libraries. The transactional flag eliminates the need for method duplication and instrumentation, which saves memory space, and has a negligible overhead. The atomic blocks can be used anywhere in the code. The program state is saved at the beginning of a transaction, and restored if the transaction is restarted – i.e., similar to `setjmp/longjmp` in C.

As mentioned before, ByteSTM implements two STM algorithms: RingSTM [91] and TL2 [25]. With RingTM, a transaction's writes are buffered in a redo log. Each transaction has a read signature and a write signature that summarize all read locations and written locations, respectively. A transaction validates its read-set by intersecting its read signature with other concurrent committed transactions in a ring. The ring is a circular buffer that has all committed transactions' write signatures. At commit, a validation is done again. If the transaction is valid, then the current transaction's write signature is added to the ring using a single Compare-And-Swap operation. If it is successfully added to the ring, then the transaction is committed and it writes-back the redo log values to the memory.

With TL2, a global lock table consisting of versioned locks is used to synchronize access to shared locations. A global clock is also used to tag each transaction with its starting time (version). A transaction's writes are buffered in a redo log. Each transaction has a

[2]Note that, false conflicts can still occur due to other implementation choices, e.g., TL2 lock table [25], read/write signatures [91].

read-set. A transaction validates its read-set by checking the corresponding lock in the lock table. If the lock is acquired or it has a larger version than the transaction's version, then the transaction aborts. At commit, a validation is done again for all read-set entries. If the transaction is valid, then the current transaction's redo log entries are locked. If all locks are acquired successfully, then the transaction writes-back the redo log values to the memory, increments the global clock, and finally updates the acquired locks' versions with the new clock value and releases them.

We conducted experimental studies, comparing ByteSTM with other Java STMs including Deuce [55], Object Fabric [72], Multiverse [99], DSTM2 [43], and JVSTM [16] on a set of micro-benchmarks and macro-benchmarks. The micro-benchmarks are data structures including Linked List, Skip List, Red-black Tree, and Hash set. The macro-benchmarks include five applications from the STAMP benchmark suite [17] (Vacation, KMeans, Genome, Labyrinth and Intruder) and a Bank application. Our results reveal that, on micro-benchmarks, ByteSTM, with the TL2 algorithm, improves throughput over others by 20% to 75%, for up to 16 threads. After 16 threads, the scalability of competitor STMs is significantly degraded, and ByteSTM's average improvement is observed to be approximately 270%. On macro-benchmarks, ByteSTM's improvement ranges from 36% to 100%.

Thus, the research contribution of the thesis is the design and implementation of ByteSTM, with superior performance over state-of-the-art competitor STMs.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 overviews TM, its key building blocks, and properties. Chapter 3 surveys past and related work, and contrast them with ByteSTM. In Chapter 4, we describe the design and implementation of ByteSTM. We report our experimental results in Chapter 5. The thesis concludes in Chapter 6.

Chapter 2

Transactional Memory

Transactional Memory (TM) borrows the transaction idea from databases. Database transactions have been successfully used for a long time and have been found to be a powerful and robust concurrency abstraction. Database transactions are guaranteed with the ACID (Atomicity, Consistency, Isolation, Durability) properties. Multiple transactions can run concurrently as long as there is no conflict between them. In the case of a conflict, only one transaction among the conflicting ones will proceed and commit its changes, while the others are aborted and retried. TM transactions only access memory. Thus, they are "memory transactions," and are therefore guaranteed to only have the atomicity, consistency, and isolation properties, which are defined as follows:

Atomicity. This means that a transaction appears to execute in a single indivisible step, and follows the "all or nothing" rule. An aborted transaction's changes are discarded and have no side effects.

Consistency. This means that a transaction only changes the memory from one consistent state to another consistent state.

Isolation. This means that a transaction's intermediate changes have no effect on other concurrent transactions.

TM can be classified into three categories: Hardware Transactional Memory (HTM), Software Transactional Memory (STM), and Hybrid Transactional Memory (HyTM). HTM [39, 7, 92, 21, 19] uses hardware to support transactional memory operations, usually by modifying cache-coherence protocols. It has the lowest overhead and the best performance. The need for specialized hardware is a limitation. Additionally, HTM transactions are limited in size and time. STM [86, 49, 98, 80, 55, 43, 16, 72, 91, 25] implements all TM functionality in software, and thus can run on any existing hardware and it is more flexible and easier to change. STM's overhead is higher, but with optimizations, it outperforms fine-grained locking and scales well. Moreover, there are no limitations on the transaction size and time. HyTM [58, 57, 76, 24, 68, 102] combines HTM and STM, while avoiding their limitations,

by splitting the TM implementation between hardware and software.

In this chapter, we give a background of TMs. We describe how TM design can be classified. Then, we list some key TM properties. Finally, we present a classification of current STM implementations. Our motivation to describe this is giving the reader enough background to follow the thesis. The STM's implementations classification shows how current STMs are built, so that ByteSTM implementation choices are clarified to the reader.

The rest of the chapter is organized as follows: In Section 2.1, we describe a classification for TM design, where different concurrency control, version control, conflict detection, and conflict resolution mechanisms are shown. Section 2.2 discusses TM properties. And Section 2.3 describes different STM implementations' categories.

2.1 TM Design Classification

TM designs can be classified based on four factors: concurrency control, version control, conflict detection, and conflict resolution[41].

2.1.1 Concurrency Control

A TM system monitors transactions' access to shared data to synchronize between them. A conflict between transactions go through the following events (in that order):

1. A conflict *occurs* when two transactions write to the same shared data (write after write), or one transaction writes and the other reads the same shared data (read after write or write after read).
2. The conflict is *detected* by the TM system.
3. The conflict is *resolved* by the TM system such that each transaction makes progress.

There are two mechanisms for concurrency control: *pessimistic* and *optimistic*. In the pessimistic mechanism, a transaction acquires exclusive access privilege to shared data before accessing it. When the transaction fails to acquire this privilege, a conflict occurs, which is detected immediately by the TM system. The conflict is resolved by delaying the transaction. These three events occur at the same time.

The pessimistic mechanism is similar to using locks and can lead to deadlocks if it is not implemented correctly. For example, consider a transaction T1 which has access to object D1 and needing access to object D2, while a transaction T2 has access to object D2 and needs access to object D1. Deadlocks such as these can be avoided by forcing a certain order in acquiring exclusive access privileges, or by using timeouts. This mechanism is useful when

the application has frequent conflicts. For example, transactions containing I/O operations, which cannot be rolled-back can be supported using this mechanism.

In the optimistic mechanism, conflicts are not detected when it occurs. Instead, they are detected and resolved at any later time or at commit time, but no later than the commit time. During validation, conflicts are detected, and they are resolved by aborting or delaying the transaction.

The optimistic mechanism can lead to livelocks if not implemented correctly. For example, consider a transaction T1 that reads from an object D1, and then a transaction T2 writes to object D1, which forces T1 to abort. When T1 restarts, it may write to D1 causing T2 to abort, and this scenario may continue indefinitely. Livelocks can be solved by using a *Contention Manager*, which waits or aborts a transaction, or delays a transaction's restart. Another solution is to limit a transaction to validate only against committed transactions that were running concurrently with it. The mechanism allows higher concurrency in applications with low number of conflicts. Also, it has lower overhead since its implementation is simpler.

2.1.2 Version Control

Version control is the process of managing a transaction's writes during execution. Two types of version control techniques have been studied: *eager versioning* and *lazy versioning* [68]. In eager versioning, a transaction writes directly to the memory (i.e., in-place update) for each object that it modifies, while keeping the old value in an *undo log*. If the transaction aborts, then the old value is restored from the undo log. Eager versioning requires *eager conflict detection*. Otherwise, isolation cannot be maintained and intermediate changes will affect other concurrent transactions.

In lazy versioning, a transaction's writes are buffered in a transaction-local write buffer, sometimes called a *redo log*. After a successful commit, values in the write buffer are written back to the memory. With this approach, a transaction's reads need to check if the write buffer has the data before reading the value from the memory. If the data is not found in the write buffer, then the value is retrieved from the memory. This approach is also know as *deferred updates*.

2.1.3 Conflict Detection

TM systems use different approaches for when and how a conflict is detected. There are two approaches for when a conflict is detected: *eager* conflict detection and *lazy* conflict detection [68]. In eager conflict detection, the conflict is detected at the time it happens. At each access to the shared data (read or write), the system checks whether it causes a conflict.

In lazy conflict detection, a conflict is detected at commit time. All read and written locations are validated to determine if another transaction has modified them. Usually this approach validates during transactions' life times or at every read. Early validations are useful in reducing the amount of wasted work and in detecting/preventing *zombie* transactions (i.e., a transaction that gets into an inconsistent state because of an invalid read, which may cause it to run forever and never commit).

Many factors are considered in how a conflict is detected. First is the *granularity* of conflict detection. In STM, three granularities are considered: object, field, or word granularity.

Object granularity monitors an object as one unit. For example, if a transaction T1 writes to a field in an object (Obj.F1), while another transaction T2 reads from another field in that object (Obj.F2), then a conflict is detect. This is an example of a *false conflict*, because of the granularity. Object granularity is faster when used in transactifying data structures, as it monitors fewer number of items (i.e., one linked list node instead of at least the node's two fields).

Field granularity monitors each field in an object separately. This approach does not suffer from false conflicts and is generally faster.

Word granularity monitors each memory word separately. Word size depends on the hardware. False conflicts may occur when multiple data items share the same memory word (e.g., a C struct that contains two bytes).

HTM usually uses *cache line granularity*, which suffers from false conflicts, since multiple memory words fit in the same cache line and multiple memory locations are mapped to the same cache line.

The second factor that is considered in how a conflict is detected is the transactions that are used in detecting conflicts. Some TMs validate between concurrent transactions, while others validate between only committed concurrent transactions.

The third factor is the way transactions are compared for detecting conflicts. Some TMs use *value-based* comparison, while others use *signature-based* comparisons.

Value-based comparison eliminates false conflicts but it is generally slower. Signature-based comparison suffers from false conflicts but it is very fast. However, false conflicts are very sensitive to the signature size.

Locks are also used for detecting conflicts: a conflict is simply detected when a transaction tries to lock an already locked lock.

2.1.4 Conflict Resolution

When a conflict occurs between two transactions, a *contention Manager* (CM) is used to determine which transaction must be aborted (or delayed) in order to resolve the conflict.

A CM applies one of the following *contention management policies* [35, 36, 84, 85, 41]:

- *Passive.* When a transaction detects a conflict with another one, it simply aborts itself. This policy is prone to livelocks, but in practice, it performs well due to its low overhead.
- *Aggressive.* When a transaction detects a conflict with another one, it forces the other transaction to abort. This policy is also prone to livelocks.
- *Polite.* When a transaction detects a conflict with another one, it delays itself using a random exponential back-off strategy. After a specific number of back-offs, the transaction forces the other one to abort.
- *Karma.* This policy uses priorities of transactions to decide which transaction must be aborted when a conflict occurs. The amount of work done so far by a transaction represents the transaction's priority. The amount of work is represented by the number of reads and writes done by the transaction. This number is accumulated until the transaction commits. If the other transaction has a higher priority, the transaction backs off for a fixed amount of time and then retries again. The number of retries equals the difference between the priorities of the two transactions. When all the retries fail, the other transaction is forced to abort. An aborted transaction maintains its priority so that it has a better likelihood of commit when it restarts.
- *Polka.* This policy is a mix of the Polite and Karma policies. It works the same as Karma, but it uses random exponential back-off instead of waiting.
- *Eruption.* This policy is similar to Karma. It also uses the amount of work done so far as the transaction's priority. But when a transaction is blocked by a higher priority transaction, it adds the transaction's priority to the blocking transaction's priority. This way, the priority of the transaction that blocks many other transactions will be the highest and will finish faster. This policy ensures that a critical resource is acquired for a short time.
- *Timestamp.* In this policy, the CM stamps a transaction with the current time when it starts. When a conflict occurs, if the other transaction is younger (i.e., started earlier), then it is forced to abort. Otherwise, the transaction waits before retrying. After a certain number of retries, the other transaction is marked. After another certain amount of retries, if the other transaction is still marked, it is forced to abort. Otherwise, the waiting time is doubled and the same scenario is repeated. A transaction is unmarked if it performs any transactional operation.
- *Greedy.* This algorithm is similar to Timestamp. Each transaction has a timestamp which represents the transaction's priority. When a conflict occurs, the other transaction is aborted if it is younger or if it is waiting for another transaction. Otherwise, the transaction waits until the other one finishes.

- *Randomized*. When a transaction detects a conflict with another one, it randomly chooses between aborting the other one or waiting for a random amount of time.
- *Kindergarten*. In this policy, the CM maintains a list of blocking transactions for each transaction. When a conflict occurs, the CM checks if the other transaction is contained in the transaction's list. If so, then the other transaction is aborted. Otherwise, the other transaction is added to the transaction's list and the transaction backs off. After a certain amount of back offs due to the same conflict, the transaction aborts itself.
- *KillBlocked*. This policy marks a transaction as blocked when it encounters the same conflict twice. Then if the other transaction is marked as blocked also, it is forced to abort. Otherwise, the transaction waits. If the transaction exceeds the maximum wait time, then it forces the other transaction to abort.
- *QueueOnBlock*. When a transaction detects a conflict with another one, it adds itself to the other transaction's queue. When the other transaction commits or aborts, it notifies all waiting transactions in its queue. Also, if a transaction waits for a long time in a queue, then it will force the queue's owner transaction to abort.

There is no "best" contention management policy because each one is better in some situation and worse in others. The choice depends on the workload and the properties of the TM. So, some TM systems give the programmer the choice of which CM to use. Others adapt automatically depending on the workload.

CM is not critical in TM systems that validate only between concurrent committed transactions. Livelock-freedom is granted when the committed transaction always wins [88, 91].

2.2 TM Properties

We now present correctness and progress properties that are relevant to TM systems. We also discuss other TM properties including strong atomicity, privatization, and nesting models.

2.2.1 Correctness Properties

Correctness properties that are relevant to TM include linearizability, serializability, strict serializability, 1-copy serializability, and opacity.

Linearizability means that all operations in a committed transaction appear as if they are executed at a single point in the transaction's lifespan. The problem with linearizability is that it does not guarantee complete isolation for running transactions. Thus, running transactions can be affected by other transactions before a conflict is detected.

Serializability means that committed transactions can be ordered so that they appear as if they were executed sequentially. Serializability does not preserve program's real-time order, but it can be restricted to support real-time order (i.e., *Strict Serializability*). There are three problems with serializability. First, it assumes that a transaction always reads the latest value written to a shared object. Second, it only supports read and write operations. Third, it does not guarantee a consistent state for a transaction.

1-Copy Serializability is similar to serializability but it allows multiple copies of the same object to exist. It solves (only) the first problem with serializability.

Opacity means that (1) all operations in a committed transaction appear as if they were executed at a single point in the transaction's lifespan; (2) transactions are completely isolated so that an aborted transaction's operations do not affect other transactions, and (3) a consistent state is always granted for each transaction. Opacity is similar to strict serializability but without its problems. Opacity guarantees that a transaction's read-set is always consistent, which eliminates zombie transactions.

2.2.2 Strong Atomicity

A memory transaction is different from a database transaction in that access to shared data is not controlled by the database system exclusively. Non-transactional code, i.e., those which are not under TM system control, can access shared data. *Strong atomicity*[12] means that a TM can detect non-transactional access to shared data, and thus transactional and non-transactional code can coexist. *Weak atomicity* means that transactional properties (i.e., atomicity, isolation, consistency) are only guaranteed when transactional code is used to access shared data.

2.2.3 Privatization

Privatization[89] means providing private access to a transactional data so that it can be accessed by non-transactional code in a safe way. Privatization is used to obtain faster access to data in a private way without transactional overhead. The main problem with privatization is how to transfer the data safely from a public state to a private one. A TM system must provide complete isolation and a consistent view during transactions' lifespan.

Figure 2.2.3 shows a privatization example. In this example, thread A may execute while thread B is still inside its transaction. If thread B is not notified that the list head has been modified after it has read it, then it will continue manipulating the list. This will conflict with thread A's non-transactional access to the list. The result is unpredictable, since thread A will not know about the conflict with thread B.

Thread A

```
1 atomic{
2       node = List.head;
3       List.head = null;
4 }
5 //List is now private
6 reverseList(List);
```

Thread B

```
1 atomic{
2       node = List.head;
3       if (node != null){
4           ...
5           node = node.next;
6       }
7 }
```

Figure 2.1: Privatization example. Thread A privatizes the list before reversing it non-transactionally. Thread B does some operations on the list inside a transaction.

2.2.4 Nesting

As mentioned before, TM supports composition. Composition requires the ability to nest transactions – i.e., invoke a transaction inside another transaction. Three types of nesting models have been studied for TM: *flat nesting*, *closed nesting*, and *open nesting*[69]. Each type provides a different level of concurrency. In each type, a child transaction (i.e., a transaction running inside another one) must have access to its parent transaction's intermediate changes.

Flat nesting means that a conflict in a parent or a child transaction aborts both of them. A child's intermediate changes are visible to the parent. A child's committed changes are not visible outside the parent until the parent commits successfully. Flat nesting provides low concurrency.

Closed nesting means that a conflict in a child transaction aborts the child only, while a conflict in a parent transaction aborts the parent and all its children. Closed nesting provides higher concurrency if aborts are frequent in the child transaction, since only the inner transaction will re-execute.

Open nesting means that a conflict in a parent or a child transaction has no effect on each other. Moreover, a child's committed changes are visible to the program's other transactions before the parent commits. Later, if the parent aborts, a list of compensation operations are executed to undo the child's committed changes. Open nesting provides the highest level of concurrency

2.3 Categories of STM Implementations

STM implementations can be categorized into four classes: *Lock-Based STMs using Local Version Numbers* , *Lock-Based STMs using a Global Clock*, *Lock-Based STMs using Global Metadata*, and *Non Blocking STMs* [41].

We start by discussing metadata and logs representations that are used by all categories. Then, we discuss each category in the subsections that follow.

2.3.1 Metadata and Logs Representations

Metadata. STMs use metadata to coordinate concurrency between transactions. Metadata examples include read-set, write-set, read signature, old object's versions, etc. In STMs with object-based granularity, metadata is added to the object, whereas in STMs with field-based or word-based granularity, the memory address is used to associate a word/field with metadata. Thus, it is impractical to use one-to-one mapping for each address. Usually, hashing is used to map the address to an entry in the metadata. Other implementations use one global metadata.

There are four factors in implementing metadata that affect the efficiency of the implementation: (1) the memory overhead, (2) the performance overhead, (3) how to map between metadata and an object or a word, and (4) false conflicts. False conflicts due to metadata representation usually occur in word/field granularity when the memory address is used with hashing. Hashing may map more than one element to the same metadata entry (i.e., collision) and cause false conflicts. The four factors depend on each other, saving memory usually affect performance, and vice versa. Also, using hashing to map affects the false conflicts ratio.

Redo log and Undo log. As mentioned before, eager versioning uses undo log, while lazy versioning uses redo log (i.e., *write buffer*). A log is implemented as a Linked List, a Hash Table, or an Array. The log implementation has a strong effect on performance. This is especially the case with a redo log, when every read needs to search the redo log, in case a transaction has written to the location before. A simple linked list implementation slows down every read operation. Each read requires $O(n)$ operations to search the linked list log linearly, if the log has n elements.

Performance can be enhanced by using a hash table, but implementation must take into consideration cache performance (i.e., reduce cache miss rate) and provide a fast clear operation[88]. Another performance improvement is to use Bloom filters[11], which provide a fast mechanism to check if the read location is contained in the write-set. If the location is found, then the write-set is searched [25]. Another important issue with logs is the granularity and the information that is maintained about each location. For example, if the log granularity is larger than the written value, then we need to store the address, value, and size.

Read-set and Write-set. Optimistic concurrency control needs to keep track of each read and write operation in order to detect conflicts. The write-set is used to store information about transactional writes. It may have pointers to items in the redo log, or redo log can also serve as a write-set. The read-set is used to determine each read location. Some implementations

remove the read-set and uses a read-signature [91].

2.3.2 Lock-Based STMs using Local Version Numbers

In this category, pessimistic concurrency is used for writes, and optimistic concurrency for reads. Each object has its own local version, which is incremented with every committed write to the object. The local version is compared when the object is read to detect conflicts. STMs use locks internally to manage concurrency. If a lock is acquired before the object is accessed, then eager or lazy versioning can be used. However, if a lock is acquired at commit time, then only lazy versioning can be used.

A *transactional read* adds the read location to the read-set and then returns the value. A *transactional write* locks the versioned lock, adds the old value to the undo log, and writes the value in memory.

The *commit* operation first checks if the read-set is still valid. It compares the read version in the read-set with the current version and checks if the lock is acquired. If the read-set is invalid, the transaction aborts. Otherwise, the transaction releases all acquired locks and increments each lock's version. This implementation does not ensure opacity.

Most STM implementations in this category use eager versioning and locking on access [1, 3, 42, 83, 82]. They do eager versioning and locking on access to improve performance (i.e., no redo log scanning with every read, and lower contention). Example STMs in this category include McRT-STM[83] and Bartok-STM[42]. Both use *versioned locks*, which provide mutual exclusion for writes and the associated version is used to detect conflict in reads. If the lock is not acquired, then the associated version is the latest version.

Deadlocks can occur, because a transaction waits until the lock is released. Deadlocks can be avoided by using a timeout, or a (deadlocked) transaction can abort itself and release all acquired locks and then wait for the necessary locks to be released before retrying.

2.3.3 Lock-Based STMs using a Global Clock

In this category, a global clock is used to support opacity without incremental validation. The global clock is updated at the application level. A transaction starts by reading the global clock into its local clock or start time.

A *transactional read* checks the write signature (represented using a Bloom filter) to determine if the transaction wrote to this location before. If so, the transaction searches the redo log for the location. If it is found, then the value in the redo lock is returned. Otherwise, the location version is read (v1), the value is read from memory, and finally, the location version is read again (v2). Then, the read is validated as follows: (1) the location is not locked (i.e., no concurrent writing), (2) v1 and v2 are compared to determine if they are

the same (i.e., the read value and its version are consistent, and no change to the value has occurred after the version is captured), and (3) the location version is not greater than the transaction's start time. Condition (3) means that the read value has not been modified since the transaction started and ensures opacity. If validation fails, then the transaction is aborted. Otherwise, the location is added to the read-set and the read value is returned.

A *transactional write* logs the write to the redo log or updates the value in the log if this is not the first time to write to the location. The write signature is updated by adding the location to the Bloom filter.

The *commit* operation starts by trying to acquire all the write-set locks. If any lock cannot be acquired, then the transaction aborts and all locks are released. The read-set is then validated to ensure that all read locations' versions are not greater than the transaction's start time. If validation fails, then the transaction aborts and all locks are released. The third step is to write-back all redo log entries to memory. Finally, the global clock is incremented, the locks are released, and the associated versions are updated with the new global clock value.

An example algorithm in this category is the TL2 algorithm[25]. TL2 uses a global clock, versioned locks, lazy versioning, and commit-time locking. Thus, a TL2 transaction has a redo log, and the write-set is locked during the commit operation. A version in a lock is relative to the global clock.

Other algorithms that use a global clock are snapshot isolation[81], LSA[78], and SwissTM[26].

2.3.4 Lock-Based STMs using Global Metadata

In this category, a global shared metadata is used without any per object/location metadata. The global metadata has a better cache performance and requires smaller number of atomic operations during a transaction's lifespan. Global metadata faces two challenges: i) how to reduce the contention on the global metadata and ii) how to detect conflicts. The first issue is solved by reducing the number of updates to the global metadata during a transaction. Moreover, most operations involving the global metadata are executed without exclusive access to it. Conflict detection is done using *Bloom filters* [91] or *value-based validation* [73].

The *Bloom filter*[11] is a data structure that represents a set. It supports adding an element to the set, checking whether it is contained in the set or not, and intersecting two Bloom filters to determine if they have common elements. All Bloom filter operations have $O(1)$ time complexity. The Bloom filter stores the set elements in an array of bits. Each element is represented by a number of bits in the array. Each bit index in the array is determined using a hash function. The main problem with Bloom filters is that it is not exact. It can claim that an element exists in the set when it is not. Also, it can claim that a common element exists between two sets when no such common elements exist. However, the opposite is not true: if the Bloom filter indicates that an element does not exist in the set or there are no

common elements between two sets, then it is true.

RingSTM[91] uses Bloom filters in conflict detection (i.e., read signature and write signature are represented using Bloom filters). RingSTM uses a global metadata called the *ring* and it uses lazy versioning. The ring contains an element for each committed transaction that contains the following: write signature, timestamp, priority, and status. Each transaction has a read signature, write signature, and a redo log. A transaction starts by finding the recent timestamp in the ring and saves it as its start timestamp.

A *transactional write* adds the write location to the write signature and logs the write to the redo log or updates its old entry. A *transactional read* adds the read location to the read signature, and checks the write signature to determine if the transaction wrote to this location before. If so, it then searches the redo log for that location. The transaction is validated by intersecting the read signature with other committed concurrent transactions in the ring. If validation fails, then the transaction aborts. This guarantees opacity.

At commit, the transaction is validated and, if it is valid, a CAS operation is used to add an entry to the ring. The new ring entry status is "writing" and the transaction starts writing back its redo log. Finally, the ring entry status is changed to complete.

The ring is a circular buffer and its size is limited. Thus, in case of an overflow, all transactions that are older than the oldest timestamp in the ring are aborted. Priority is used to solve the starvation problem. If a transaction suffers multiple aborts, it raises its priority, which is done by adding a dummy transaction with a high priority and a full write signature to the ring. This prevents other transactions from committing and allows the transaction to finish without interruption.

Value-based validation is another mechanism for detecting conflicts. It is based on logging read values (not just locations) and validating them based on those values. If no change has occurred in the values, then no conflict is detected. Value-based validation alone can miss some conflicts when a value is changed more than once, and the last change updates it to the original logged value. JudeSTM[73], TML[90] and NOrec[23] are example STMs in this category.

JudeSTM logs a read with its value, and consecutive reads to the same location obtain the value from the log. Thus, the value of any read location does not change unless the transaction writes to it. Commits are serialized using locking. Thus, other transactions cannot commit until the current committing transaction finishes.

Two types of locking are used to serialize committing. First is coarse-grained locking, where a single versioned lock is used. Any writer transaction needs to acquire the lock before it can commit, and at the end of a successful commit, the version of the lock is incremented and the lock is released. Validating the read-set at commit time is value-based. A *transaction-instance-specific* code is used to speedup the validation by emitting a validation code that is optimized for each transaction instant. A read-only transaction's commit does not acquire the lock: the transaction reads the lock version, then validates, and finally reads the lock

version again and compares to ensure that no commit occurred during validation.

Second is fine-grained locking, where hashing is used to map a location to a versioned lock.

2.3.5 Non Blocking STMs

This category of STMs support concurrency between transactions without using any lock. It aims to provide the lock-freedom or obstruction-freedom property for an STM. This requirement is difficult, since a transaction's memory accesses must appear as one atomic operation using a series of atomic memory operations and without using locks. In a lock-free execution, a transaction cannot block another transaction from modifying shared data. Also, the transaction must guarantee that all changes appear once and no intermediate changes are visible.

An example from this category is DSTM[44]. It is called "dynamic" because it is the first STM that does not require listing of all the shared data in advance, and allows adding new data dynamically. It is implemented in Java and uses explicit transactional objects. It supports transactional operations on transactional objects only.

A *two level of indirection* is used to access a transactional object using a *locator*. The locator contains the following: (1) the last transaction that opened the object for writing, (2) the new value, and (3) the old value. The locator is immutable and can only be modified by replacing it with a new locator using the CAS atomic operation. Each transaction has a descriptor, which contains the transaction status (ACTIVE, COMMITTED, or ABORTED). When a transactional object is accessed for read through the locator, the locator's new value is returned if the locator's last writing transaction's status is COMMITTED. Otherwise, the locator's old value is returned (i.e., if the status is ABORTED or ACTIVE).

When a transactional object is accessed for write through the locator, if the last writing transaction is another transaction and its status is ACTIVE, then a conflict occurs. The other transaction is aborted with one CAS operation and its status is changed from ACTIVE to ABORTED.

Commit is simply done by a single CAS operation to change the transaction status from ACTIVE to COMMITTED. This change will affect all objects related to the transaction, since each object points to the transaction.

Other STMs in this category include: Practical lock-freedom [29], Concurrent programming without locks [30], ASTM [63], and RSTM [64].

Chapter 3

Related Work

Lomet was the first to map database transactions to memory transactions in 1977 [59]. He described the idea, but his implementation was not competitive. Later, Herlihy and Moss presented the first practical implementation of TM at the hardware level (i.e., the first HTM) in 1993 [45]. Their HTM proposal was to modify hardware cache-coherence protocols (e.g., MESI [74]) with transactional caches. In 1995, Shavit and Touitou presented the first STM. Theirs was a static STM, which required listing all shared objects in advance before starting a transaction [86]. Since [86], TM (both HTM and STM) has been gaining significant attention from researchers. Moreover, chip vendors have increasingly begun to integrate HTM in their processors. Examples include the Oracle/Sun Rock processor [20] and AMD's Advanced Synchronization Facility (ASF) [21]. In the STM space, major C/C++ compilers are being rolled out that support memory transactions. Examples include Intel's C++ STM compiler® [52], and GCC version 4.7 compiler [32, 4].

In this chapter, we survey past and related efforts on STM implementations, and contrast them with ByteSTM. Our main focus is on Java STM implementations, since that is ByteSTM's problem space. We also describe C/C++ implementations.

3.1 Java STM Implementations

We classify Java STM implementations into two categories: library-based implementations and virtual machine-based implementations. These are surveyed in the subsections that follow.

3.1.1 Library-based Implementations

Deuce [55] is a complete STM framework for Java. It is implemented as a Java library and requires no changes to the JVM or the Java language (i.e., no new compiler is required). Deuce provides implementations of two STM algorithms: TL2 [25] and LSA [78]. Moreover, it supports plug-ins of other STM algorithms or implementations by providing a new `Context` class that contains the new implementation.

Deuce's usage is simple as it supports transactions implicitly. Transactions are supported by simply marking a method with the `@Atomic` annotation. Once marked, the method will run automatically in a transaction, and all variables inside the method will be accessed transactionally (except local variables, which are not accessible outside the method).

Deuce uses instrumentation at the bytecode level to modify the methods with transactional support code. An `@Atomic` method is wrapped in a loop to retry the implicit transaction if the commit fails. Other methods called within the `@Atomic` method are duplicated. The new duplicate is a transactional version of the method which is only called from another transactional code. A transactional version of a method redirects all memory accesses to the STM `Context` and changes all method calls so that they call the transaction version of each method.

Instrumentation can be done online (using a Java Agent during class loading) or offline. Bytecode instrumentation allows Deuce to support external libraries without having access to the source code. ASM [10] is used for instrumentation.

Deuce uses field-based granularity. It uses `sun.misc.Unsafe` [95] non-standard proprietary API to access memory.

In contrast, ByteSTM supports all Deuce features but plug-ins. ByteSTM uses different implementation. It does not use instrumentation, has direct access to the memory without using non-standard libraries, supports atomic blocks anywhere in the code, and its metadata bypass the GC.

JVSTM [16] is an STM Java library that uses a multi-version STM algorithm (i.e., *versioned boxes*). It is optimized for read-only transactions, which are wait-free and do not conflict with any other transaction. A versioned box keeps a history of an object's old values. It uses lazy conflict detection at commit time and reduces conflicts by (1) delaying computations that use high contention boxes, and (2) restarting only the part of a transaction that causes a conflict.

JVSTM implementation works on transactional objects only. A transactional object is defined as a `VBox` that wraps the original object. Like Deuce, the `@Atomic` annotation is used to mark a method as atomic. Once marked, an offline instrumentation (using ASM) modifies the method and adds transactional code. Also, it supports explicit transactions. External libraries are not supported since transactions work on `VBox` objects only. JVSTM uses object-based granularity.

In contrast, ByteSTM works with all data types (not just transactional objects), supports external libraries, does not use instrumentation, and uses field-based granularity. Currently, ByteSTM has no multi-version STM algorithm implementation.

ObjectFabric [72], according to the authors, is a "lightweight, cross-platform library, which helps developers store and exchange data in real-time with the cloud and between applications." It contains an STM implementation based on XSTM [70]. XSTM is called extensible, because it allows pluggable components to access and process a transaction's read-set and write-set. XSTM is a multi-version STM and supports opacity by taking a snapshot and keeping it consistent. It uses a *heap version*, which is a queue that contains all writes to an object by a transaction. When a transaction reads an object, it returns the version that matches the transaction's start time.

XSTM supports strong atomicity by allowing only transactional objects inside a transaction, and non-transactional code can only access the object through small transactions. All shared data are immutable, and mutable data are only available privately. This way, no synchronization is required from transaction's start to commit. Commit is lock-free and only takes $O(1)$ time. XSTM's idea is similar to source code version control systems. A transaction starts by taking a snapshot, do its work, then finally merge with the repository.

ObjectFabric uses explicit transactions, which can only work on transactional objects. Thus, it does not support external libraries.

In contrast, ByteSTM supports implicit transactions, works with all data types (not just transactional objects), supports external libraries, and uses field-based granularity. Currently, ByteSTM has no multi-version STM algorithm implementation.

AtomJava [**49**] supports implicit transactions through atomic blocks. Atomic blocks are supported by adding the `atomic` keyword to the Java language. In order to compile the new code, a source-to-source conversion is used, which converts the new code to regular Java code, which can then be compiled using any standard Java compiler. During this conversion, transactions are used to replace atomic blocks, and objects are modified to support transactional access. Moreover, strong atomicity is supported during the code conversion.

Polyglot [71] is used to do the code conversion. The source-to-source conversion significantly changes the original code and makes debugging difficult.

AtomJava uses object-based locking. It adds a lock field to each object class, which reduces the lock overhead but increases the memory overhead. The object lock is acquired on first access to an object. Thus, AtomJava does not use optimistic concurrency, and is similar to fine-grained locking. Deadlock detection is used to prevent deadlocks.

AtomJava can support external libraries, but the source code must be provided. (In our experimental studies (Chapter 5), the source-to-source conversion did not work as expected, and the resulting applications did not work correctly.)

In contrast, ByteSTM works with or without a compiler's support, works at the VM-level,

and uses field-based granularity.

DSTM2 [43] is an object-based STM library. Transactional objects are created using special *factory* classes. DSTM2 has two built-in factories. One of them is like DSTM [44], but with more optimization. Custom factories can be added to DSTM2 to test new algorithms or implementations. DSTM2 provides a new thread class that supports running transactions and can access transactional objects. Also, it supports user-defined contention managers.

A transactional object class is defined as a Java `interface` having the `@atomic` annotation. A DSTM2 factory object is created for each transactional object. When the factory object is created, a synthetic anonymous class is created that implements the given interface. Reflection and a bytecode engineering library (BCEL [9]) are used to create the class. This class is created only once and then the factory uses it to create transactional objects. Transactional objects consist of primitives or other transactional objects only.

DSTM2 uses explicit transactions that works with only transactional objects. Thus, it does not support external libraries. The initial version does not support transactional nesting.

In contrast, ByteSTM supports implicit transactions, works with all data types (not just transactional objects), supports external libraries, and uses field-based granularity. Currently, ByteSTM has no non-blocking STM algorithm implementation.

Multiverse [99] is a language-independent STM implementation that can work with any language running on a JVM (e.g., Scala, Groovy, or JRuby). Thus, it does not depend on instrumentation and uses explicit transactions. Note that, supporting only JVM languages is not really language independent. Moreover, explicit transactions are generally difficult to use and are not transparent to the programmer.

Multiverse uses GammaSTM [99], which is an optimized version of the TL2 [25] algorithm. It reduces the contention on the global clock by updating it only on conflicts and not with every write (it calls the global clock, the "conflict counter").

GammaSTM adds more information to an object ownership record: (1) the number of current readers (i.e., *surplus*), (2) whether it is read-biased or update-biased, and (3) the lock information, which contains a read lock and an exclusive write lock. GammaSTM has semi-visible readers, which means that only the number of transactions that reads an object is known. The object's surplus is incremented when a transaction reads an object, and it is decremented when the transaction commits successfully. When a writer transaction commits and finds that the object surplus is not zero or it is marked as update-biased, it increments the conflict counter. A reader transaction validates only if the conflict counter is changed after the transaction was started.

An object is marked as read-biased when it is read multiple times without any writes. A read transaction will not update a read-biased object's surplus, which reduces access to the object's metadata. An object is marked as update-biased with any write to the object.

In contrast, ByteSTM supports implicit transactions, works with all data types (not just

transactional objects), and supports external libraries.

LSA-STM [81, 79] is an STM library that uses the LSA algorithm [78]. It uses online instrumentation to generate transactional code – one of the first Java STM libraries to do so – using the ASM library. Instrumentation relies on annotations. Transactional objects are marked with `@Transactional`, and atomic methods are marked with `@Atomic`. Other annotations are available to mark methods of transactional objects as read-only methods using `@ReadOnly`, or update methods using `@Update`.

LSA-STM transactions can only access transactional objects. Therefore, LSA-STM does not support external libraries.

The LSA algorithm is object-based, and is a multi-version obstruction-free STM. High performance requires having invisible readers (i.e., a transaction that writes to an object does not know if another transaction has read it). But to maintain a consistent view for the transaction, read objects must be validated with every read. Visible readers add a list to an object containing all the transactions that read from the object. Thus, no validation is required (but this has a higher overhead). LSA solves the problem by providing a transaction with a consistent snapshot to work on. The snapshot is kept consistent by lazily extending its validity range.

In contrast, ByteSTM works with all data types (not just transactional objects), supports external libraries, does not use instrumentation, supports atomic blocks anywhere in the code, and uses field-based granularity. Currently, ByteSTM has no multi-version STM algorithm implementation.

3.1.2 Virtual Machine Implementations

Harris and Fraser proposed one of the first Java STM implementations [40]. In this work, the Sun Java Virtual Machine (JVM) for Research was modified to support transactional memory operations. The Sun JVM is written in C. Therefore, this work can be considered as a Java STM implemented in C. It supports atomic blocks using the new `atomic` keyword. A modified compiler is used to compile these blocks into bytecode. To simplify the conversion, only methods can be atomic. Thus, local variables are not monitored. Only fields and array elements are monitored. Atomic methods are recognized by the virtual machine by adding a suffix to their name during compilation. An atomic block inside a method is converted to a transactional method by extracting the code inside the atomic block and adding it to the new method. The method name suffix idea allows supporting transactions without adding new bytecode instructions. A transactional copy is created from each method, but it is only compiled on demand.

This C-based STM is implemented in the JVM to get benefits from its managed environment. The algorithm used allows inconsistent reads to occur, which can cause unrecoverable errors. For example, an inconsistent read can cause a loop to write outside the bounds of an array.

In an unmanaged environment, the memory will be overwritten. No comparison is presented in [40] with other STM implementations since this is one of the first Java STMs.

In contrast, ByteSTM provides opacity, can work with or without a compiler's support, and is implemented in Java using JikesRVM [5].

Atomos [18] presents a new programming language that is based on Java. It replaces Java monitors with transactions. The keyword `synchronized` is replaced with `atomic`, and the `wait/notify` conditional variable is replaced with `watch/retry`. Removing Java's standard synchronization mechanisms is a major change. This means that there is no backward compatibility for legacy code. Also, there are situations where some transactions may require different handling, such as irrevocable transactions which cannot be aborted. One type of transactions cannot support all situations.

Atomos is implemented inside the JVM. It supports implicit transactions, strong atomicity, and open nesting. Atomos requires an HTM that supports strong atomicity. The JikesRVM [5] and TCC HTM [65] are used in Atomos implementation.

In contrast, ByteSTM is implemented entirely in software, and extends the Java language. Currently, ByteSTM does not support strong atomicity, open nesting, or transactional conditional variables.

The work in [100] adds transactional monitors to the Java language. It is implemented in JikesRVM [5]. Standard Java synchronization is not removed, but it cannot be used with transactional monitors. Transactional monitors are implemented as implicit transactions.

Transactional monitors uses exceptional handling to support transaction re-execution. The Bytecode Engineering Library [9] is used to inject exception handling code that restores program state and restart the transaction. Transactional monitors uses write and read barriers, which are functions called by JikesRVM with every read or write.

In contrast, ByteSTM uses atomic blocks instead of transactional monitors. ByteSTM uses different implementation. It does not use bytecode rewriting, does not read/write barriers, and its metadata bypass the GC. ByteSTM access memory directly, without using barriers, by changing how bytecode is translated.

Adl-Tabatabai et. al. [3] optimize STM operations by using a JIT compiler in the ORP [22] managed environment with a new language extension. They modified the StartJIT Java compiler [2] to optimize STM calls, which are handled by McRT-STM [83]. First, the source code is compiled by Polyglot [71] to handle the new language extensions. Then, the JIT compiler optimizes STM calls that use McRT-STM.

This work concentrated on optimizing STM calls to McRT-STM using a JIT compiler which is part of the VM. ByteSTM is integrated in the VM. The bytecode instruction can run in transactional mode and no calls to external STM libraries are required.

3.2 C/C++ STM Implementations

RSTM [98] is an STM library that contains several STM algorithm implementations. Originally, it used the RSTM algorithm [64] and supported only explicit transactions. Currently, it is integrated with the Intel C++ STM compiler, which supports implicit transactions. RSTM can work on several architectures and operating systems.

SwissTM [26] is a new STM algorithm and it is available as a library. The SwissTM algorithm efficiently supports large as well as short transactions. SwissTM is lock-based, word-based, uses lazy conflict detection for read/write conflicts, uses eager conflict detection for write/write conflicts, and has a two-phase contention manager that enhances the performance of long transactions without affecting short ones.

TBoost.STM [34] is a C++ STM library. It is based on DracoSTM [33], and is planning to be integrated with the Boost C++ libraries [13]. It is an object-oriented STM that only uses object-oriented language features. DracoSTM is a lock-based STM that supports both eager and lazy versioning.

JudoSTM [73] uses *dynamic binary-rewriting* instrumentation, which dynamically changes the program binary instructions. This allows it to support external libraries, functions that contain locks, and irrevocable system calls. The instrumentation is done on-the-fly at runtime. Moreover, the commit validation and write-back code is generated on the fly to support value-based validation efficiently.

TinySTM [80, 27, 28] is a lightweight STM library. The implementation is based on the LSA algorithm [78], but it is word-based. It is also a lock-based STM and share several locking concepts from lock-based algorithms such as TL2 [25].

3.3 Summary

ByteSTM is implemented at the VM-level. Its implementation focuses on addressing current STM limitations. ByteSTM uses implicit transactions, does not use instrumentation, works with all data types, supports external libraries, supports atomic blocks anywhere in the code, has a direct access to the memory, uses field-based granularity, and eliminates GC overhead. ByteSTM is implemented entirely in software.

In Table 3.1, we summarize our comparison of ByteSTM. Each row of the table describes an STM feature, and each column describes an STM implementation. The table entries describe the features supported by the different STMs.

Feature	Deuce	JVSTM	ObjectFabric	AtomJava	DSTM2	Multiverse	LSA-STM	Harris and Fraser	Atomos	Transactional monitors	ByteSTM
Implicit transactions	✔	✔	✖	✔	✖	✖	✔	✔	✔	✔	✔
No instrumentation	✖	✖	✔	✔	✔	✔	✖	✖	✔	✖	✔
All data types	✔	✖	✖	✔	✖	✖	✖	✔	✔	✔	✔
External libraries	✔	✖	✖	✔[1]	✖	✖	✖	✔	✖[2]	✔	✔
Unrestricted atomic blocks	✖	✖	✔	✔	✔	✔	✖	✔	✔	✔	✔
Direct memory access	✔[3]	✖	✖	✖	✖	✖	✖	✔	✔	✖	✔
Field-based granularity	✔	✖	✖	✖	✖	✖	✖	✖	✖	✖	✔
No GC overhead	✔[4]	✖	✖	✖	✖	✖	✖	✔	✔	✖	✔
Compiler support	✖	✖	✖	✔	✖	✖	✖	✔	✔	✔	✔ & ✖[5]
Strong atomicity	✖	✖	✔	✔	✖	✖	✖	✖	✔	✖	✖
Closed/Open nesting	✖	✔	✔	✖	✖	✖	✖	✖	✔	✖	✖
Conditional variables	✖	✖	✖	✖	✖	✖	✖	✖	✔	✖	✖

[1] Only if source code is available.
[2] It is a new language, thus no Java code is supported.
[3] Using non-standard library.
[4] Uses object pooling, which partially solves the problem.
[5] ByteSTM can work with or without compiler support.

Table 3.1: Comparison of Java STM implementations.

Chapter 4

Design of ByteSTM

ByteSTM is a Software Transactional Memory at the Virtual Machine level. It is built on JikesRVM [5] which is a Research Java Virtual Machine implemented in the Java programming language and runs on itself (i.e., no other VM is required to run the VM Java code). In ByteSTM, the same bytecode instructions can run in two modes: transactional and non-transactional, thus it is named ByteSTM.

ByteSTM is implemented inside the VM so that it can be tuned carefully to get the best performance. The visible modifications to the users of the VM are very limited: two new instructions are added (`xBegin` and `xCommit`) to the VM bytecode instructions. These two instructions will need compiler modifications to support generating the correct bytecode for them when it translates the *atomic* blocks. Also, the compiler should handle the new keyword `atomic` correctly. However, in order to eliminate the need for a modified compiler, a simpler workaround is used which is calling the static method `stm.STM.xBegin()` to begin a transaction, and `stm.STM.xCommit()` to commit the transaction. These two methods are defined empty and static in the class `STM` in `stm` package.

ByteSTM is implicitly transactional: the program only specifies the start and the end of the transaction and all memory operations (loads and stores) inside these boundaries are implicitly transactional. This simplifies the code inside the atomic block and also eliminates the need for making a transactional version for each memory load/store instruction, thereby keeping the number of added instructions minimal. When `xBegin` is executed, the thread enters in transactional mode. In this mode, all writes are isolated and the execution of the instructions is speculative until `xCommit` is executed. At that point, the transaction is compared against other concurrent transactions for a conflict. If there is no conflict, the transaction is allowed to commit and at this point only, all the transaction modifications are visible to the outer world. If the commit fails, all the transaction modifications are discarded and the transaction restarts again from the beginning.

In this chapter, we start by discussing the Algorithms used in ByteSTM and compare between

them, then we give detailed description of ByteSTM implementation.

4.1 Algorithms

In ByteSTM, two of the best STM algorithms are implemented with slight modifications: *RingSTM* and *TL2*.

4.1.1 RingSTM

The *RingSTM* [91] algorithm uses lazy versioning and lazy validation. The brilliant idea is the elimination of the read-set and using signatures (i.e., *Bloom filters* [11]). All read locations are added to the read signature, and all written locations are added to the write signature. Also, a write buffer is used to buffer all the transaction writes until commit time. At commit time, if there is no conflict with other concurrent transactions, the write buffer is written back to the main memory. In order to detect conflicts between concurrent transactions, a global circular buffer, called the *ring*, is used to hold committed transactions' write signatures. A global clock is also required to know the timing relation between the committed transactions and the current ones.

Each transaction has a read signature, a write signature, a write buffer, and its start time, as its local data structures. The global data structures include the ring (a circular buffer of committed transactions' write signatures) and the clock.

When a transaction begins, the local data structures are initialized and the transaction start time is set to the last committed transaction time in the ring. At every store, the address is added to the write signature and the address with the new value is added to the write buffer (or replaced if it already exists in the write buffer). At every load, the address is added to the read signature, and then, if this value exists in the write buffer (i.e., the writing transaction has already written to this location before), then the value in the write buffer is returned. Otherwise, the value in the memory is returned.

To prevent errors due to reading a location again after being changed by another committed transaction, a transaction is validated with every read and if a conflict is detected, the transaction is aborted. Thus, RingSTM provides opacity [37].

At commit time, if a transaction is read-only, then nothing more is needed to be done. Otherwise, the transaction's read signature is compared to all committed transactions in the ring that are committed after the transaction has started (i.e., concurrent transactions). (The global clock's current value is used to determine the concurrent transactions.) If no conflict is detected, the global clock is advanced using only one atomic *CAS* (Compare And Swap) operation after ensuring that any concurrent write-back is completed. Advancing the global clock means reserving a location in the ring, which is used to store the transaction's write

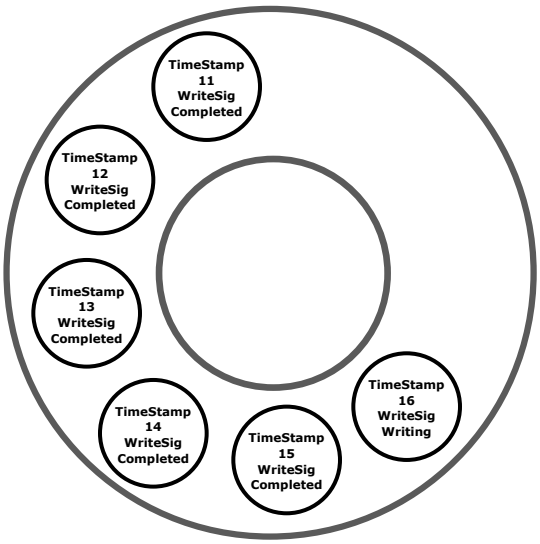


Figure 4.1: RingSTM's ring data structure holding 5 completed transactions and one committed transaction which is still in the writing-back phase.

signature. If the CAS operation fails, which means that another transaction has committed, and a new comparison is needed before retrying CAS again. The transaction is aborted if a conflict is detected. After the CAS has executed successfully, the transaction adds its write signature to the ring. At this point, it is committed and only the write-back is remaining.

Figure 4.1 shows the ring data structure. Each entry has a committed transaction's timestamp, write signature, and status. Figure 4.2 shows the RingSTM algorithm pseudo code.

The Bloom filter [11] is a data structure for Set representation that uses a hash function (or multiple hash functions), and uses a small amount of memory. The data structure is an array of bits with a specific length and supports the following operations:

add: adds an element to the set and has $O(1)$ complexity. The element is hashed to one or more bits in the array, and those bits are set to 1.

contains: checks if an element is contained in the set or not. This function has false positives, which means that it can return true, when the element is actually not present in the set. However, it does not has false negativies. The complexity of the function is $O(1)$. Figure 4.3 shows a Bloom filter with a false positive case.

intersect: checks if two sets have common elements or not. The function also has false positives, and has an $O(1)$ complexity.

The $O(1)$ complexity of all its operations make the Bloom filter very fast in detecting conflicts between transactions. Also, the read-set space overhead is reduced with a Bloom filter. However, a small overhead is added to the memory loads/stores to add the location to the

```
1 xBegin(){
2    initializeData();
3    startTime = Ring.lastCommittedTimeStamp;
4 }
5
6 xLoad(address){
7    readSignature.add(address);
8    if (checkInFlight()) abort();
9    if (writeSet.contains(address))
10      return writeSet.get(address);
11   else
12      return Memory.load(address);
13 }
14
15 checkInFlight(){
16   // intersect against all new entries
17   for (i = Ring.lastCommittedTimeStamp; i >= startTime + 1; i--)
18      if (Ring.entry(i % RING_SIZE).intersect(readSignature))
19         abort();
20   // no need to check again the same ring elements, so advance the startTime
21   startTime = Ring.lastCommittedTimeStamp;
22 }
23
24 xStore(address, value){
25   writeSignature.add(address);
26   writeSet.put(address, value);
27 }
28
29 xCommit(){
30   if (writeSet.isEmpty())  //read only transactions need no more work
31      return;
32   do{
33      commitTime = Ring.lastCommittedTimeStamp;
34      if (commitTime != startTime){
35         for (i = commitTime; i >= startTime + 1; i++)
36            if (Ring.entry(i % RING_SIZE).intersect(readSignature))
37               abort();
38         startTime = commitTime;
39      }
40   }while(!Ring.lastCommittedTimeStamp.compareAndSet(commitTime, commitTime
        +1));
41   Ring.addEntry((commitTime + 1) % RING_SIZE, writeSignature);
42   writeSet.writeBack();
43 }
```

Figure 4.2: RingSTM pseudo code.

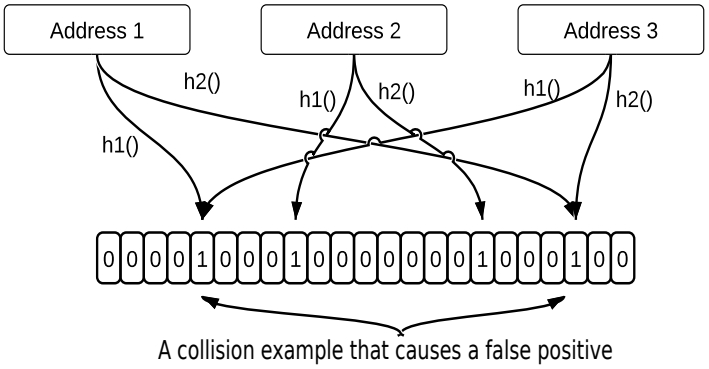


Figure 4.3: Bloom filter example. An array of bits and two hashing functions *h1 & h2* are used to map the addresses to two bits in the array. Also, the example shows a false positive case when *contains(address3)* will return true although the set contains *address1* and *address2* only.

read/write signatures. To detect a conflict, at commit time, a transaction's read signature is intersected with all concurrent committed transactions' write signatures (which are stored in the ring). This overhead is much lower than testing a transaction's read-set against other transactions' write-sets.

The drawback of using the Bloom filter is the false positives, which falsely claim a conflict when there is no conflict. This increases the ratio of aborts and wastes work. The ratio of false positives can be reduced by increasing the size of the signature. But increasing the size of the signature too much will affect the performance as the overhead is increased.

4.1.2 Transactional Locking II (TL2)

The *TL2* [25] algorithm also uses lazy versioning and lazy validation.

This algorithm uses a global clock and a global lock table (a table of versioned write-locks). The global clock is used to determine the timing relation between transactions. The lock table is used to lock memory locations and to also store the time when a transaction locked the location. Of course, it is not possible to create a lock for every memory location. Instead, the address of the memory is mapped to an entry in this table, and multiple memory locations are mapped to the same table entry. In order to reduce collisions in the lock table, the difference between two addresses mapped to the same location is made equal to the table size. This is easily done by removing the higher bits from the memory address and using the remaining lower bits as the table index (assuming no memory alignment). Figure 4.4 shows a lock table example.

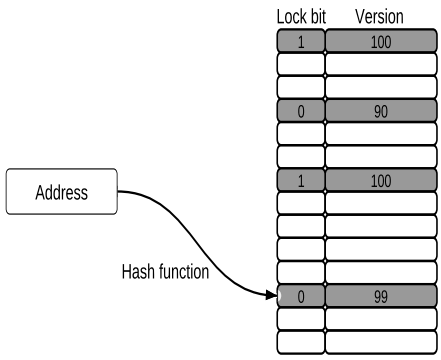


Figure 4.4: A TL2 lock table. Each entry has a lock bit and a version. A hash function is used to map an address to an entry in the table.

Each transaction has the following local data structures: a read-set, a write-set (buffer), a write signature, and a local clock. When a transaction begins, the local data structures are initialized and the local clock is set to the global clock. At every store, the address is added to the write signature and the address with the new value is added to the write-set (or replaced if it already exists in the write-set). At every load, the address is added to the read-set, and then the lock table is checked to see if this location is locked or has a clock value larger than the writing transaction's local clock (which means that a newer committed transaction has written to this location after the transaction started). If the location is not locked and is not changed, then the write signature is checked to determine if the transaction has written to the location before. If so, the value in the write-set is returned. Otherwise, the value in the memory is returned. If the lock table location is locked or is changed, then the transaction is aborted.

At commit time, if the write-set is empty, then the commit is done. Otherwise, all locations in the write-set are locked using the lock table. If any location cannot be locked (i.e., the location is already locked), then the acquired locks are released and the transaction is aborted. After all locks are acquired successfully, all locations in the read-set is checked in the lock table to ensure that no other concurrent committed transaction changed them. If the read-set check fails, then the acquired locks are unlocked and the transaction is aborted.

Finally, the global clock is incremented, and for each entry in the write-set, the value is written back to the memory and the location's associated entry in the lock table is unlocked and set to the new global clock value. Figure 4.5 shows the TL2 algorithm pseudo code.

4.1.3 Comparison between RingSTM and TL2

RingSTM and TL2 are in the same category of STM algorithms: Lock-based, lazy versioning, and lazy validation. RingSTM eliminates the read-set and only uses a read signature to speed

```
1 xBegin(){
2    initializeData();
3    localClock = globalClock.get();
4 }
5
6 xLoad(address){
7    readSet.add(address);
8    // Check the read is still valid
9    if (!LockTable.checkLock(address, localClock)) abort();
10   if (writeSet.contains(address))
11      return writeSet.get(address);
12   else
13      return Memory.load(address);
14 }
15
16 xStore(address, value){
17   writeSet.put(address, value);
18 }
19
20 xCommit(){
21   if (writeSet.isEmpty())  //read only transactions need no more work
22      return;
23
24   for each (element in writeSet){
25      if (!LockTable.lock(element)){
26         //couldn't lock all the writeSet elements, so unlock the currenlty
               locked elements then abort
27         LockTable.unlock(writeSet)
28         abort();
29      }
30   }
31
32
33   for each (element in readSet){
34      if (!LockTable.checkLock(element, localClock)){
35         //Some elements changed externally in the readSet, so unlock the
               writeSet elements then abort
36         LockTable.unlock(writeSet)
37         abort();
38      }
39   }
40
41   newClock = globalClock.incrementAndGet();
42   for each (element in writeSet){
43      Memory.store(element);
44      //unlock the writeSet element and set the value of the lock to the new
           clock
45      LockTable.setAndReleaseLock(element, newClock);
46   }
47 }
```

Figure 4.5: TL2 pseudo code.

up the validation. Also, one atomic operation is used in the commit operation. On the other hand, TL2 reduces the overhead of validating the read-set by adding a global lock table and it has almost no false positives (false positives can only happen if two memory locations are mapped to the same table entry). However, at commit time, one atomic operation per write-set entry is required.

RingSTM serializes the commit operations, since each transaction must wait for the currently committing transaction to finish its write-back phase. In contrast, TL2 allows transactions to commit and write-back concurrently if they write to different memory locations.

RingSTM uses *committed transactions wins* contention management policy. It compares only against concurrent committed transaction. This policy guarantees livelock-freedom. TL2 uses *passive* contention management policy, where a transaction aborts itself when a conflict is detected.

4.2 Implementation details

Implementing STM at the virtual machine level allows many opportunities for performance tuning. Some first generation STMs [43, 64, 99, 72] used explicit transactional code, i.e. start the transaction, commit the transaction, do a transactional read, do a transaction write. Moreover, some STMs only work with objects that are defined as transactional, and cannot do transactional reads or writes on any other objects [43, 64, 99, 72, 16, 81]. Other STMs used reflection to generate transactional objects and handle them [43]. Others used source-to-source conversion [49]. Finally, modern STMs [55, 81, 16] uses automatic instrumentation, which uses Java annotations to mark methods as atomic. The instrumentation engine handles all code inside atomic methods and modify them to run as a transaction. This conversion does not need the source code and can be done offline or online. Instrumentation allows, for the first time, using external libraries so that code inside a transaction can call methods from these external libraries that may modify the program data [55].

ByteSTM is the next step. No instrumentation is required. This means lesser overhead than online instrumentation, and smaller program size compared with the instrumented program, resulting in less memory usage. Atomic blocks can be used anywhere in the code (either using the `atomic` keyword or by calling `xBegin` and `xCommit`), and it is not necessary to make a whole method atomic; any block can be atomic. External libraries can be used inside transactions without any change. Memory access is monitored at the field level, and not at the object level. Field-based granularity scales well and eliminates false conflicts resulting from two transactions changing different fields in the same object.

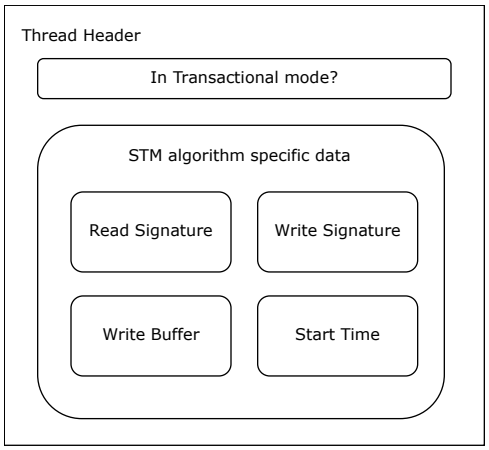


Figure 4.6: A thread header example with added metadata.

4.2.1 Metadata

Working at the VM level allows changing the thread header without modifying the program code. For each thread that executes transactions, metadata added include the read signature, the write signature, the write buffer, and the start time. These metadata is added to the thread header and is used by all transactions executed in the thread. Figure 4.6 show a thread header example with added metadata. Since each thread executes one transaction at a time, there is no need to create new data for each transaction, allowing reuse of the metadata. Accessing a thread's header is much faster than using Java's standard `ThreadLocal` abstraction.

4.2.2 Memory Model

At the VM-level, the memory address of each field of an object can be easily obtained. As mentioned before, ByteSTM is not object-based, it is field-based. The address of each field is used to track memory reads and writes. A conflict occurs only if two transactions modified the same field of an object. Static objects are also supported.

In Java, arrays are objects. ByteSTM tracks memory accesses to arrays at the element level. That way, unnecessary aborts are eliminated. Moreover, no reflection is needed and data are written directly to the memory, as a memory address is already available at each load and store.

Absolute memory address is used to unify different addressing mechanisms used in Java. An instance field absolute address equals the object's base address plus the field's offset. A static object's field absolute address equals the global statics memory space's address plus the field offset. Finally, an array's element absolute address equals the array's address plus

the element index in the array (multiplied by the element's size).

To optimize memory access, we handle memory access in its raw format. This means that the address of a memory location and the number of bytes at that address are all the information needed to access that location, irrespective of whether it is a long field, integer field, integer array element, or a reference to another object. This abstract view simplifies how the read-set and the write-set are handled. At a memory load, all information needed to track the read is the memory address of the read location. At memory store, the memory address, the new value, and the size of the value are the information used to track the write. When data is written back to memory, the write-set information (address, value, and length of the location) is used to store the committed values correctly. This abstraction also simplifies the code, as there is now no need to differentiate between different data types, as they are all handled as a sequence of bytes in the memory. The result is simplified code that handles all the data types, and has a smaller number of branches (no type checking), yielding faster execution.

4.2.3 Atomic Blocks

ByteSTM supports atomic blocks anywhere in the code. When `xBegin` is executed, the current program state is saved. If a transaction is aborted, the saved state is restored and the transaction can restart as if nothing has changed in the local variable – i.e., similar to `setjmp/longjmp` in C. This technique simplifies handling of the local variable. Note that, at the VM level, full control of the program state is possible.

4.2.4 Write-set Representation

ByteSTM is optimized for small transactions. We found that using a complex data structure to represent read-sets and write-sets that only contains a small number of elements affects performance. Given the simplified raw memory abstraction used in ByteSTM, we decided to use simple arrays of primitive data types. This decision is based on two reasons. First, array access is very fast and has access locality, resulting in better cache usage. Second, with primitive data types, there is no need to allocate a new object for each element in the read/write set (an array of objects in Java is allocated as an array of references) and each object need to be allocated alone. Hence, there is a large overhead for allocating memory for each element of an array. Even if object pooling is used, the memory will not be contiguous since each object is allocated independently in the heap.

Using arrays to represent the write-set means that the cost of searching an n-element write-set is $O(n)$. For $n \leq 10$ (which is the case in micro-benchmarks [54]), this is acceptable, and was found to be faster than using hashing, given the overhead of the standard Java hash table (which uses linked-lists for bucket overflows and supports only objects).

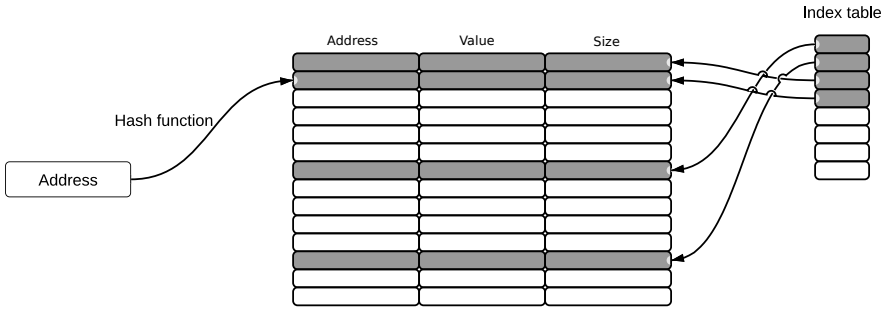


Figure 4.7: ByteSTM's write-set using open addressing hashing.

To obtain the benefits of arrays and the speed of hashing, open-addressing hashing is used. In open-addressing hashing, a large array is used to store the data. The index of the element in this large array is calculated using a hash function. If a collision occurrs, a linear probing is used. We use an array of size 2^n, which simplifies the modulus calculation.

The hash function that we used is simple, which simply removes the upper bits from the memory address using bitwise *and* operation (which is equivalent to calculating the modulus of the address): *address AND mask = address MOD arraySize*, where *mask = arraySize* - 1. For example, if *arraySize = 256*, then *hash(address) = address AND 0xFF,* where 0xFF is 255 in hexadecimal representation, which is eight ones in the binary format. This hashing function is very efficient with addresses, as the collision ratio is very small. When a collision happens, there is always an empty cell after the required index because of the memory alignment gap (so linear probing will give good results). This way, we have a very fast and efficient hashing function that adds very little overhead to each array access, enabling $O(1)$-time searching and adding operations on large write-sets.

Iterating over the write-set elements by going through all the array elements is not efficient. We solve this problem by keeping a log of all the used indices, and then iterating on the log.

Open addressing has two drawbacks: memory overhead and rehashing. These drawbacks can be controlled by choosing the array size such that the number of rehashing is reduced, while minimizing memory usage. ByteSTM uses a predefined array size that gives good results over different workloads. We are planning to allow the programmer to set the initial array size. Figure 4.7 show how ByteSTM's write-set is represented using open-addressing.

4.2.5 Garbage Collector

One major drawback of building an STM for Java (or any VM-based language) is the garbage collector (GC) [66]. STM uses metadata to keep track of transactional reads and writes. This requires allocating memory for the metadata and then releasing that memory when not needed. Frequent memory allocation (and implicit deallocation) will force the garbage

collector to run more frequently to release unused memory. This increases the overhead on the STM operations.

Some STMs have tried to solve this problem by reducing memory allocation and recycling the allocated memory [55]. For example, object pooling is used to reduce the pressure on the memory system in [55]. The idea is to first create a pool of objects. When an object is needed, it is requested from the pool. When an object is released, it is recycled and returned back to the pool so that it can be reused. If the object needs exhaust the pool, then more objects are created and added to the pool. Such memory reuse and recycle results in improved performance [55]. However, this is still done under the control of the Java memory system. The GC will continue to check if the pooled objects are still referenced. Also, memory allocation is done through the Java memory system.

Since ByteSTM is integrated into the VM, its memory allocation and recycling is done outside the control of the Java memory system. That is, ByteSTM directly allocates its own memory, recycles the memory, and releases the memory if needed. STM's memory requirement, in general, has a specific lifetime. When a transaction starts, it requires a specific amount of metadata, which remain active for the duration of the transaction. When the transaction commits successfully, meta data is recycled. Thus, manual memory management does not increase complexity or overhead to the implementation.

The GC causes another problem for ByteSTM. ByteSTM stores intermediate changes in a write buffer. Thus, program's new allocated objects will not be stored in the program's variable. The GC scans only the program's stack to find objects that are no longer referenced. The GC will not find any reference to the new allocated objects and will release their memory and recycle it. When ByteSTM commits a transaction, it will be writing a *dangling* pointer that will crash the program. ByteSTM solves the problem by giving the GC a list of all intermediate objects in a transaction's write buffer, so that the GC will not touch.

Figure 4.8 shows a simplified overview of ByteSTM architecture. It shows how ByteSTM works with or without a compiler, how memory operations can run in two modes (i.e., transactional and non-transactional), how ByteSTM accesses the memory directly when it writes back values to memory and to handle its metadata, and how ByteSTM communicates with the GC to prevent it from destroying the program's intermediate changes.

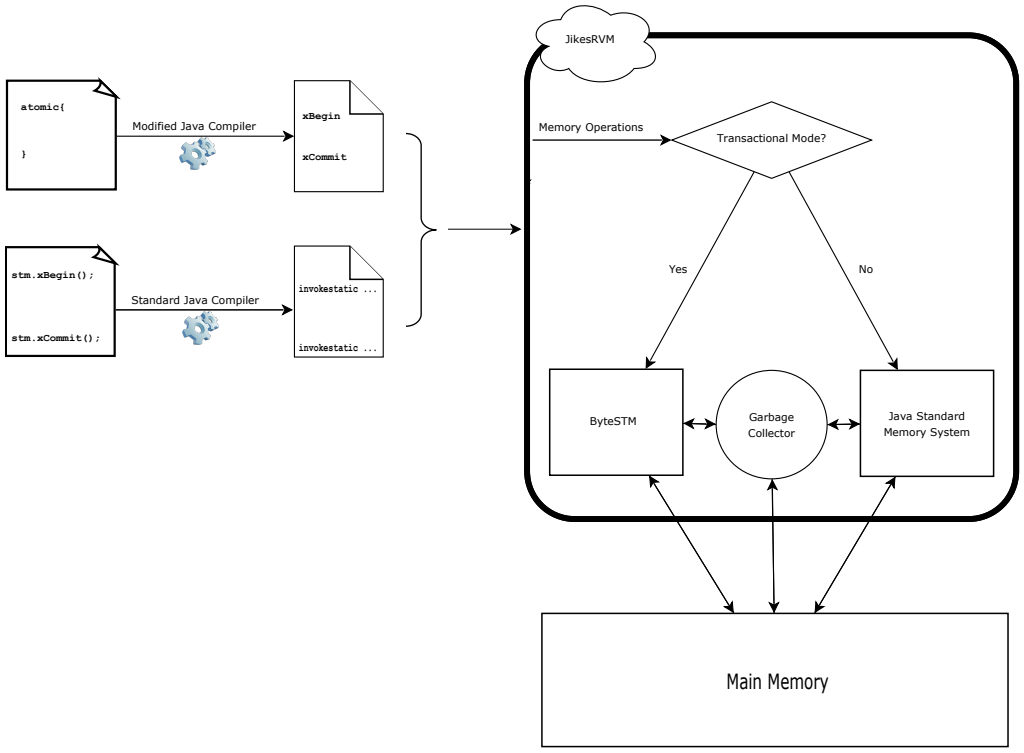


Figure 4.8: ByteSTM architecture.

Chapter 5

Experimental Results & Evaluation

In this chapter, the performance of ByteSTM is compared against other Java STMs using micro-benchmarks and macro-benchmarks. The competitor STMs include Deuce [55], ObjectFabric [72], Multiverse [99], DSTM2 [43], and JVSTM [16]. The micro-benchmarks are data structures including Linked List, Skip List, Red-black Tree, and Hash set. The macro-benchmarks include five applications from the STAMP benchmark suite [17] (Vacation, KMeans, Genome, Labyrinth and Intruder) and a Bank application. We also compare ByteSTM against a set of highly concurrent data structures including Lock-Free Linked List, Lock-Free Skip List, Lock-Free BST, and Concurrent Hash Set.

Micro-benchmarks give basic information about STM designs, and show the effect of specific cases on STM performance. But, they do not represent real-world workloads. Macro-benchmarks provide real-world workloads that test a wide range of transaction attributes (e.g., transaction length, read/write set size, and contention level). We compare against highly concurrent data structures to have an insight into how close ByteSTM performance is to highly concurrent data structures.

The chapter is organized as follows. We first describe the test environment. Experimental results and analysis for micro-benchmarks, macro-benchmarks, and highly concurrent data structures are subsequently described. We conclude with a summary of the analysis.

5.1 Test Environment

We conducted the experiments on a 48-core machine. The machine has four AMD Opteron^{TM}Processors (6164 HE), each with 12 cores running at 1700 MHz, and 16 GB of memory. The machine runs Ubuntu Linux Server 10.04 LTS 64-bit. JikesRVM is used to run all experiments.

The competitor STMs include Deuce [55], ObjectFabric [72], Multiverse [99], DSTM2 [43], and JVSTM [16]. Deuce is a highly optimized Java STM library that uses a fast memory

access library, supports implicit transaction, and instruments code to add transactional support. We used offline instrumentation to eliminate online instrumentation overhead. ObjectFabric is a Java STM library that only supports explicit transactions and uses a multi-version STM algorithm. Multiverse is a Java STM library that only supports explicit transactions and uses a TL2-like algorithm. DSTM2 is a Java STM library that only supports explicit transactions and uses a non-blocking STM algorithm. JVSTM is a Java STM library that supports implicit transactions, and instruments code to add transactional support. It works on transactional objects only, and uses a multi-version STM algorithm. For more details refer to Chapter 3.

For the micro-benchmarks and the bank application, we measured the transactional throughput (i.e., the number of transactions per second) under the benchmarks, thus higher is better. For the STAMP macro-benchmarks, we measured the execution time under the benchmarks, thus smaller is better (i.e., faster).

Each experiment was repeated 10 times. The plots show the average and the 90% confidence interval for each data point.

5.2 Micro-Benchmarks

The micro-benchmarks that we considered include the data structures Linked List, Skip List, Red-Black Tree, and Hash Set. We converted these data structures from using course-grain locking to use transactions. The transactions contain all the code that was inside the critical sections in the course-grain locking version. Different ratios of writes and reads were used to measure the performance under different levels of contention.

We also varied the number of threads. Threads were incremented in exponential steps (i.e., 2, 4, 8, ...), up to 48, which is the number of cores in the test machine. Increasing the number of threads beyond the number of available cores will add the effect of multi-programming, so that was not considered.

5.2.1 Linked List

In this benchmark, an ordered, singly-linked linear list data structure is used. The list supports three operations: add, remove, and contains. Concurrency is supported by running each of these operations in a transaction.

Linked-list operations are characterized by a high number of reads (the range is from 70 at low contention to 270 at high contention), due to traversing the list from the head to the required node, and a few writes (about 2 only). This results in long transactions. Moreover, we observed that transactions suffer from a high number of aborts (abort ratio is from 45% to 420%), since each transaction keeps all visited nodes in its read-set, and any modification

to these nodes, by another transaction's add or remove, will abort the transaction.

We conducted four experiments, each one with a different read/write ratio: 80% reads to 20% writes, 50% reads to 50% writes, 20% reads to 80% writes, and 100% writes without reads.

Figure 5.1 shows the performance of the six Java STMs. Deuce has two curves representing the LSA [78] and TL2 [25] algorithms. Also, ByteSTM has two curves representing the RingSTM [91] and TL2 [25] algorithms. The y-axes represents the throughput, and the x-axes represents the number of threads.

We observe from the figure that, in all cases, ByteSTM with RingSTM achieves the best performance and scalability. This is followed by ByteSTM with TL2. Deuce/TL2 is the third in performance and scalability, while Deuce/LSA's performance degrades quickly as the number of threads is increased. Other STMs perform in a similar way, and all of them have a very low throughput. DSTM2 has a very poor performance, which is close to zero.

At higher contention (i.e., higher write ratio), TL2's scalability degrades. However, RingSTM continues to scale well up to 100% write ratio.

ByteSTM with TL2 outperforms Deuce by as much as 25% and up to 47% with an average of 38%. Moreover, ByteSTM with RingSTM outperforms Deuce by as much as 75% and up to 220% with an average of 140%. The large gap between TL2 and RingSTM is due to the elimination of the read-set and using signatures in the RingSTM, given the very small number of writes.

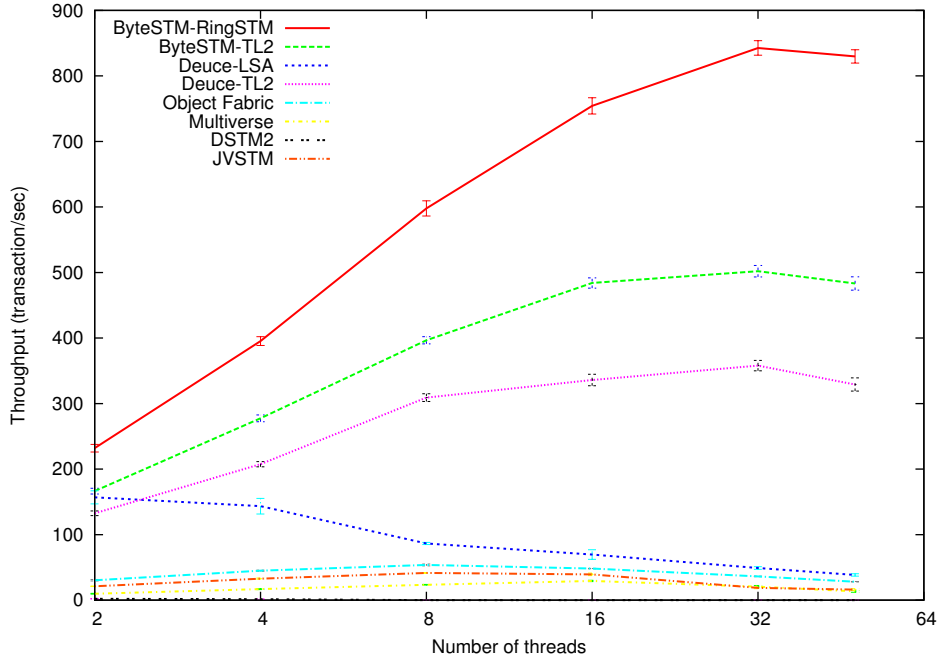
5.2.2 Skip List

In this benchmark, a standard Skip List [75] data structure is used. The Skip List supports three operations: add, remove, and contains. Concurrency is supported by running each of these operations in a transaction.

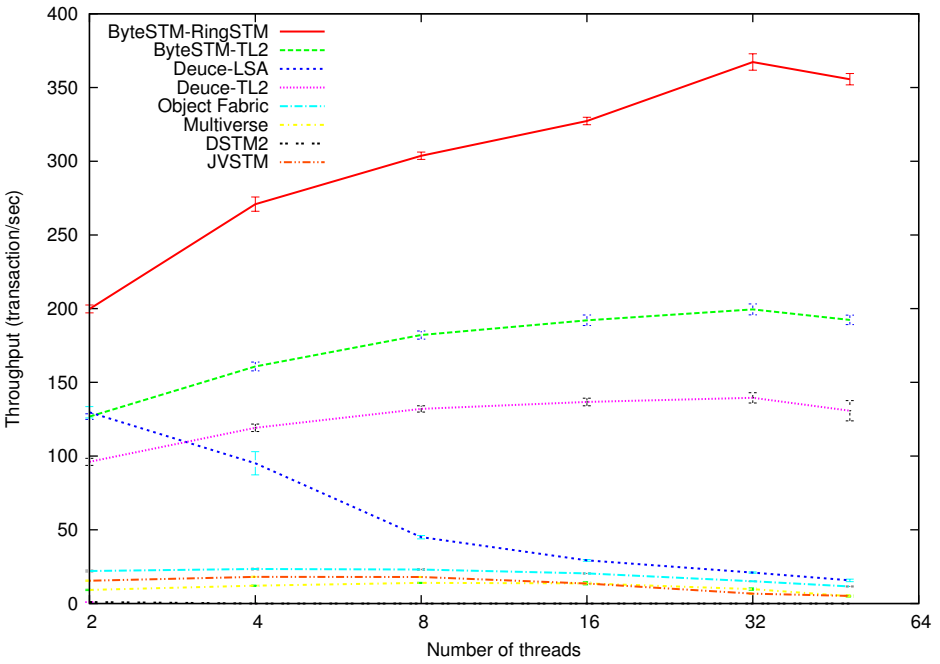
Skip List is a probabilistic data structure and has an expected $O(log\, n)$ time complexity. So, the Skip List operations are characterized by a medium number of reads (from 20 to 40), and a small number of writes (from 2 to 8). This results in medium-length transactions. Moreover, transactions suffer from a low number of aborts (abort ratio is from 4% to 20%). From our experimental studies, we observed that ByteSTM with RingSTM suffers from a higher number of aborts (abort ratio is from 18% to 150%).

We conducted four experiments, each with the same read/write ratio, similar to the Linked List benchmark: 80% reads to 20% writes, 50% reads to 50% writes, 20% reads to 80% writes, and 100% writes without reads.

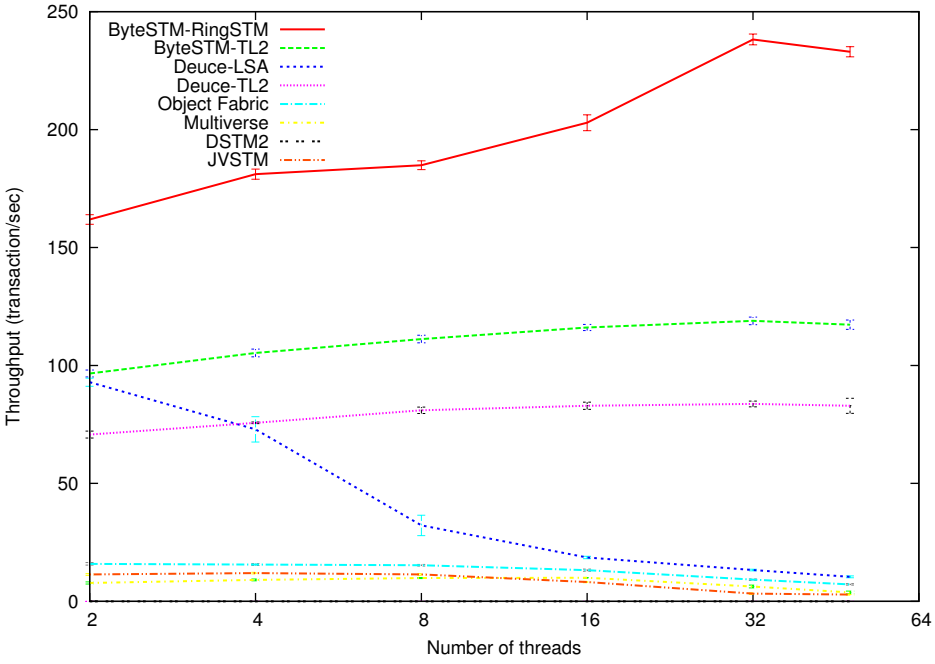
Figure 5.2 shows the results. The y-axes represents the throughput, and the x-axes represents the number of threads. From the figure, we observe that, in all cases ByteSTM with TL2



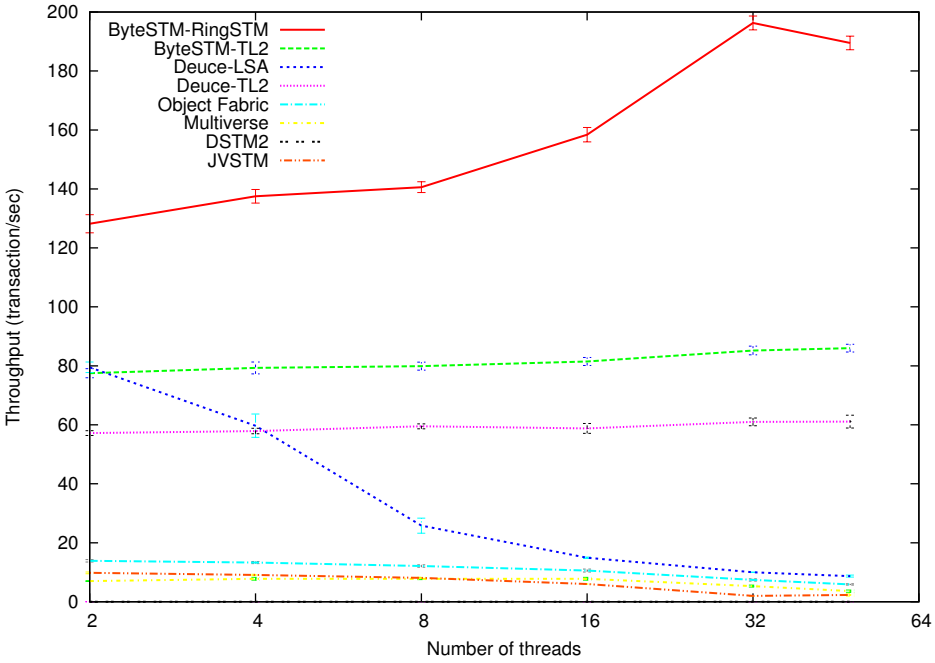
(a) 20% writes.



(b) 50% writes.

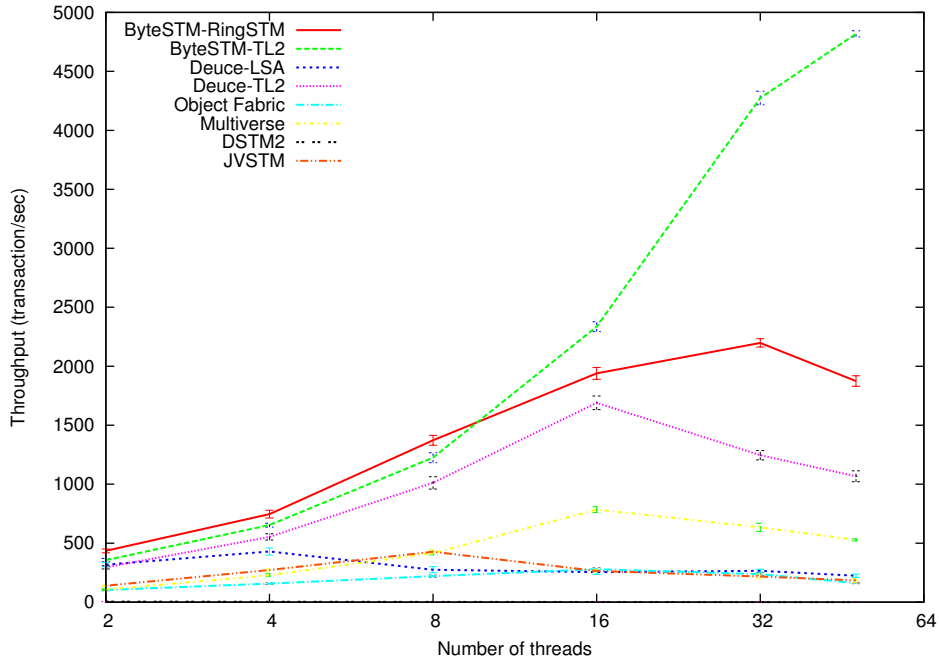


(c) 80% writes.

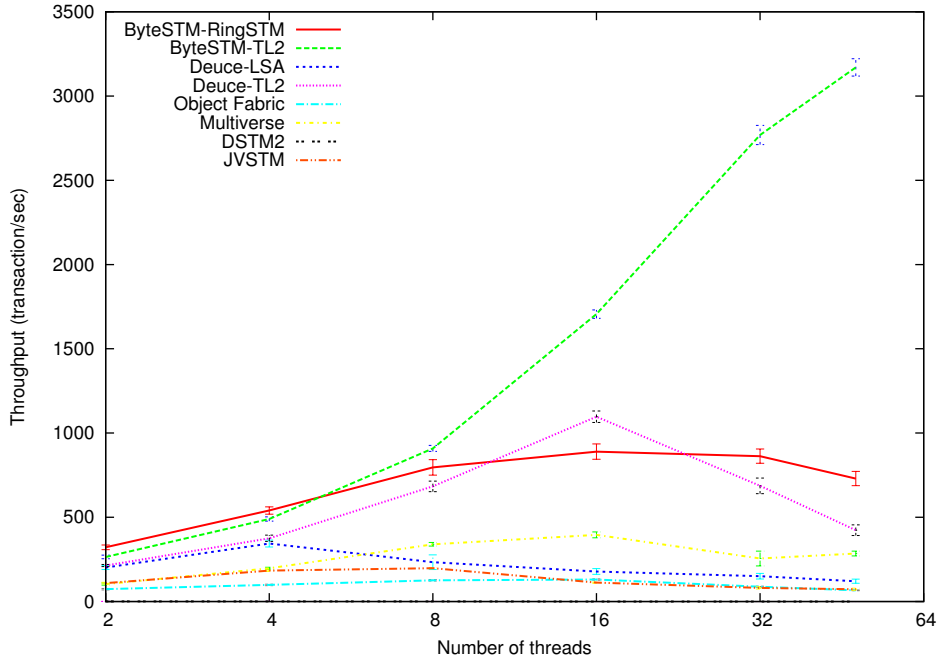


(d) 100% writes.

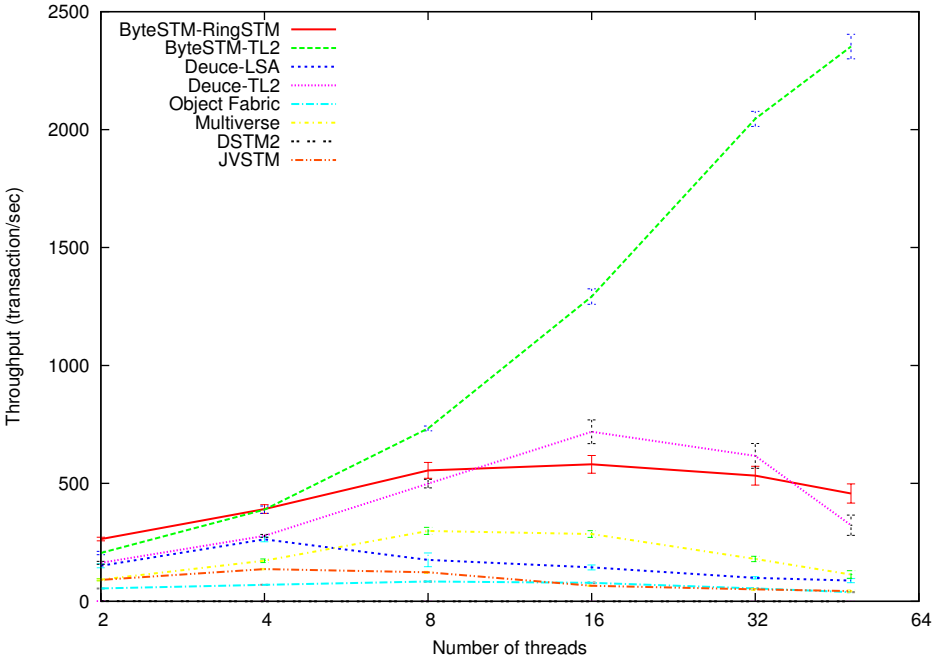
Figure 5.1: Throughput under Linked List.



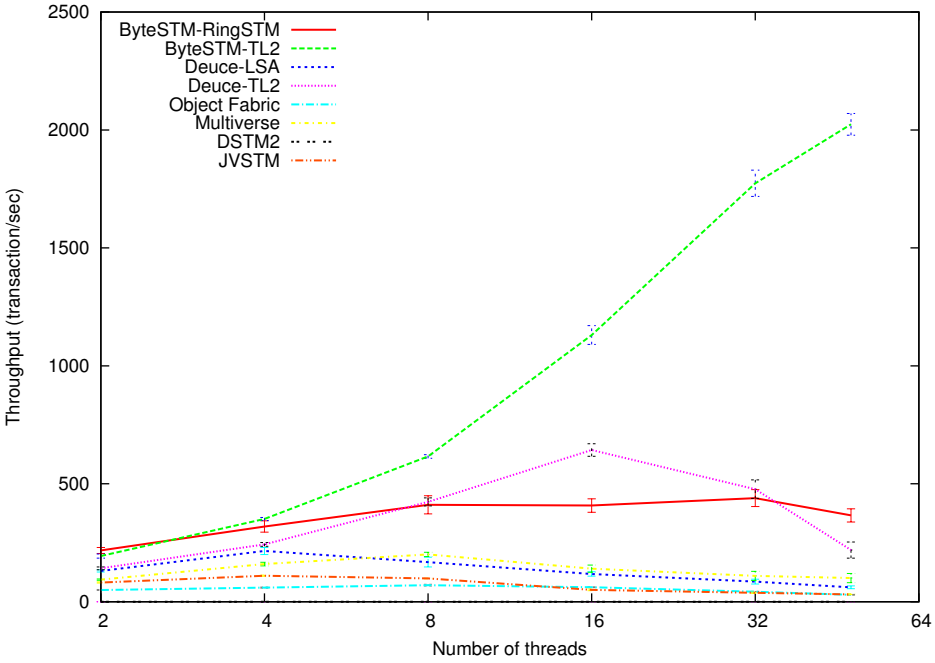
(a) 20% writes.



(b) 50% writes.



(c) 80% writes.



(d) 100% writes.

Figure 5.2: Throughput under Skip List.

achieves the best performance and scalability. ByteSTM with RingSTM performs well up to 8 threads; then the scalability is affected by the higher ratio of aborts due to Bloom filter's false positives. Deuce with TL2 is the third in performance and scalability, with a better performance than RingSTM at 16 threads. Deuce/LSA's performance degrades quickly as the number of threads is increased. Other STMs' performance and scalability are poor. DSTM2 has a very poor performance, which is almost close to zero.

Since Deuce's performance degrades significantly, we will limit our comparison up to 16 threads. ByteSTM with TL2 outperforms Deuce by as much as 20% and up to 75%, with an average of 40%.

Since Deuce with TL2 achieved the best performance among all other STMs, for all further experiments, we will use Deuce as a fair competitor against ByteSTM to avoid clutter in the figures.

5.2.3 Red-Black Tree

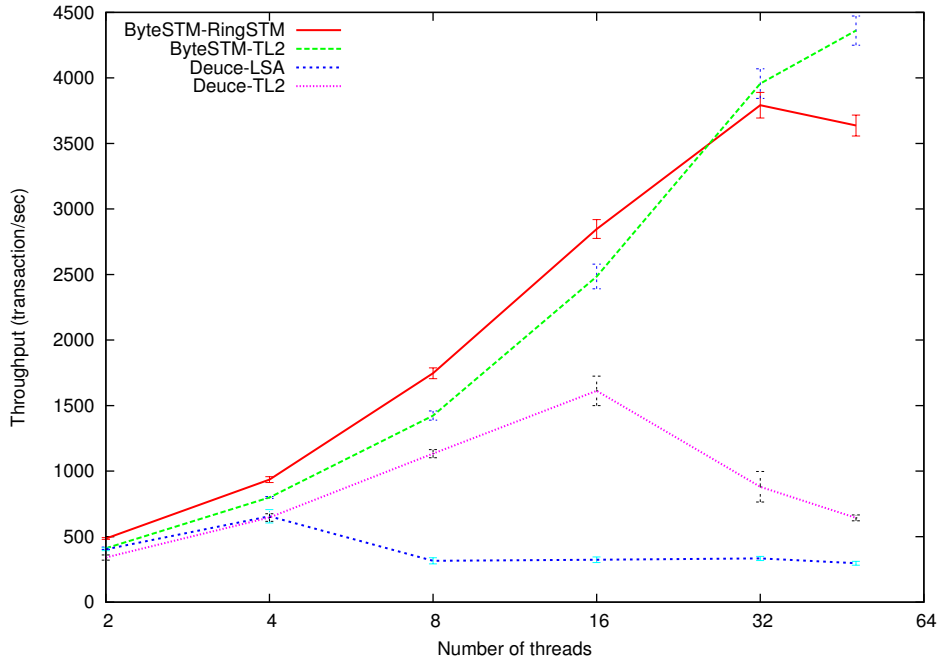
In this benchmark, a standard Red-Black Tree data structure is used. The tree supports three operations: add, remove, and contains. Concurrency is supported by running each of these operations in a transaction.

Red-Black Tree is a balanced binary search tree, and it has an $O(log\, n)$ time complexity for all its operations. Balancing the tree is accomplished by doing rotations, which modify the tree structure to keep it balanced. So, the Red-Black Tree operations are characterized by a small number of reads (from 15 to 30), and a small number of writes (from 2 to 9). This results in short transactions. Moreover, transactions suffer from a low number of aborts (abort ratio is from 4% to 30%).

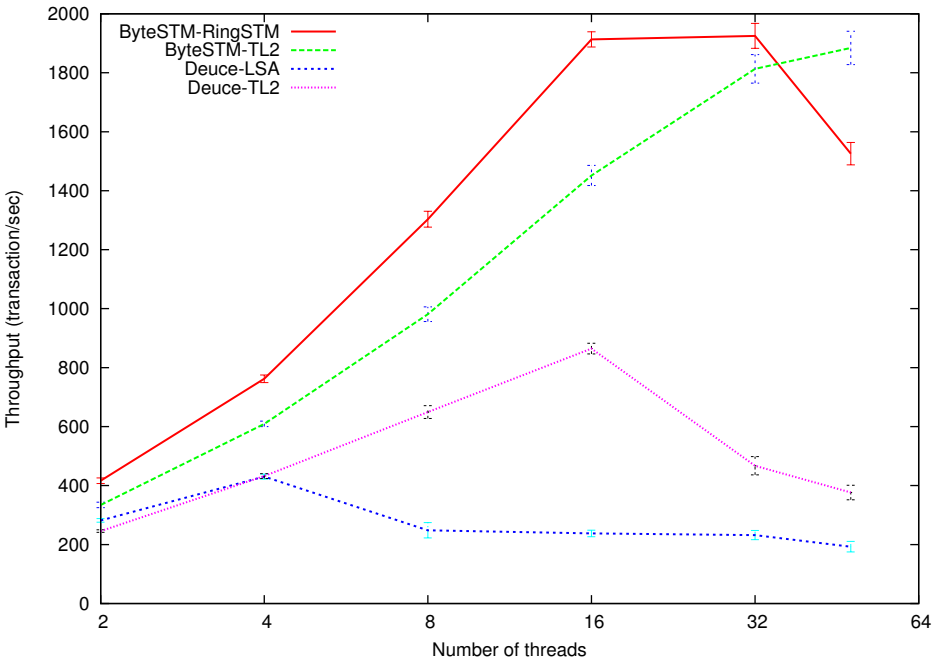
As before, we conducted four experiments, each with the same read/write ratio: 80% reads to 20% writes, 50% reads to 50% writes, 20% reads to 80% writes, and 100% writes without reads.

Figure 5.3 shows the results. The y-axes represents the throughput, and the x-axes represents the number of threads. From the figure, we observe that, in all cases, ByteSTM with RingSTM achieves the best performance for up to 32 threads. RingSTM's performance begins to degrade after 16 threads, and with increased number of writes. This performance degradation is due to the increased false positive ratio in the Bloom filter that increases the number of aborts. ByteSTM with TL2 achieves the next best performance and the best in scalability. Deuce with TL2 is the third in performance, but also suffers from performance degradation after 16 threads.

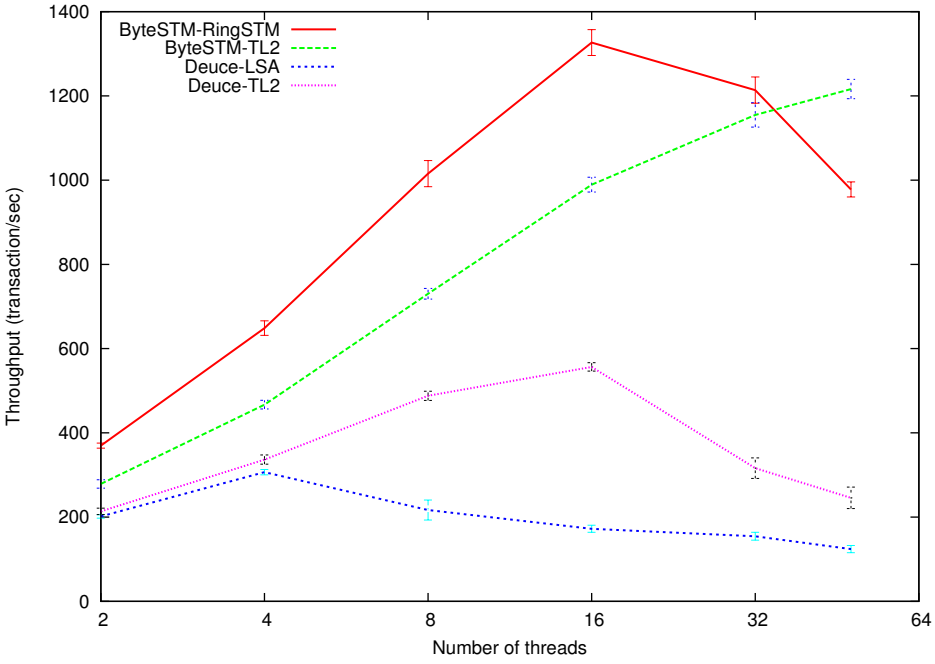
Due to Deuce's performance degradation, we will limit our comparison to 16 threads. ByteSTM with TL2 outperforms Deuce by as much as 20% and up to 74%, with an average of 45%. Moreover, RingSTM outperforms Deuce by as much as 42% and up to 143%, with an aver-



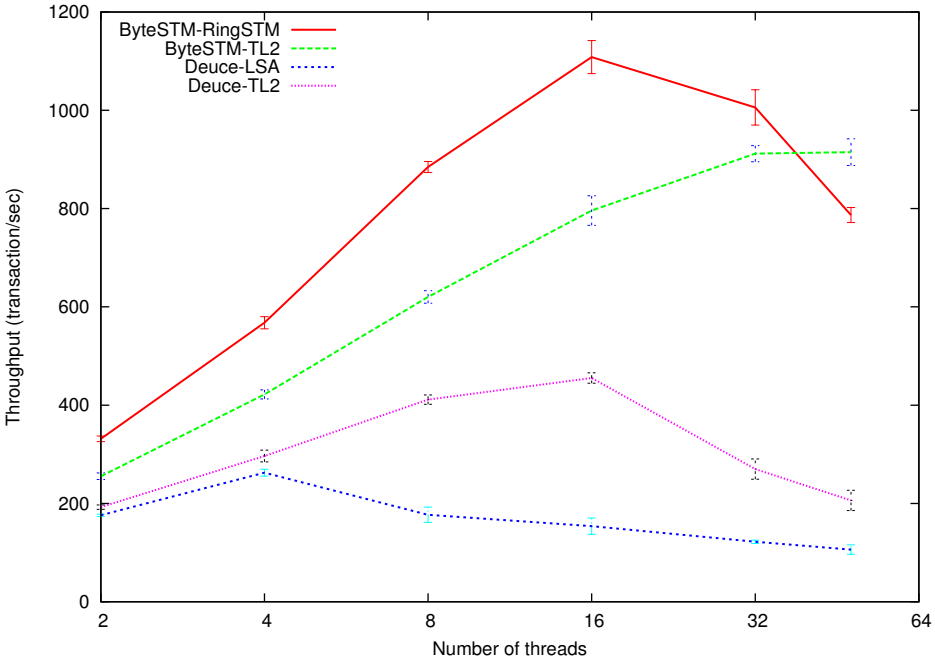
(a) 20% writes.



(b) 50% writes.



(c) 80% writes.



(d) 100% writes.

Figure 5.3: Throughput under Red-Black Tree.

age of 89%. In this experiment, the gap between TL2 and RingSTM is not large. This is because, the number of reads is small, and the number of writes is larger than that of the Linked List.

5.2.4 Hash Set

In this benchmark, a hash table with chained hashing data structure is used to represent a set. This set supports three operations: add, remove, and contains. Concurrency is supported by running each of these operations in a transaction.

Hash Set has a $O(1)$ time complexity for all its operations, but there is a constant overhead due to collisions and handling the chain. When a collision occurs, the new element is added to the linked-list chain. Moreover, if the same element is added again, the hash table is updated, which increases the number of modifications to the data structure. So, the Hash Set operations are characterized by a small number of reads (from 2 to 31), and a medium number of writes (from 7 to 15). This results in short transactions. Moreover, the transactions suffer from a high number of aborts (abort ratio is from 63% to 556%) due to collisions, linked-list chains, and duplicate inserts that update the memory. The high abort ratio in this benchmark affects all implementations.

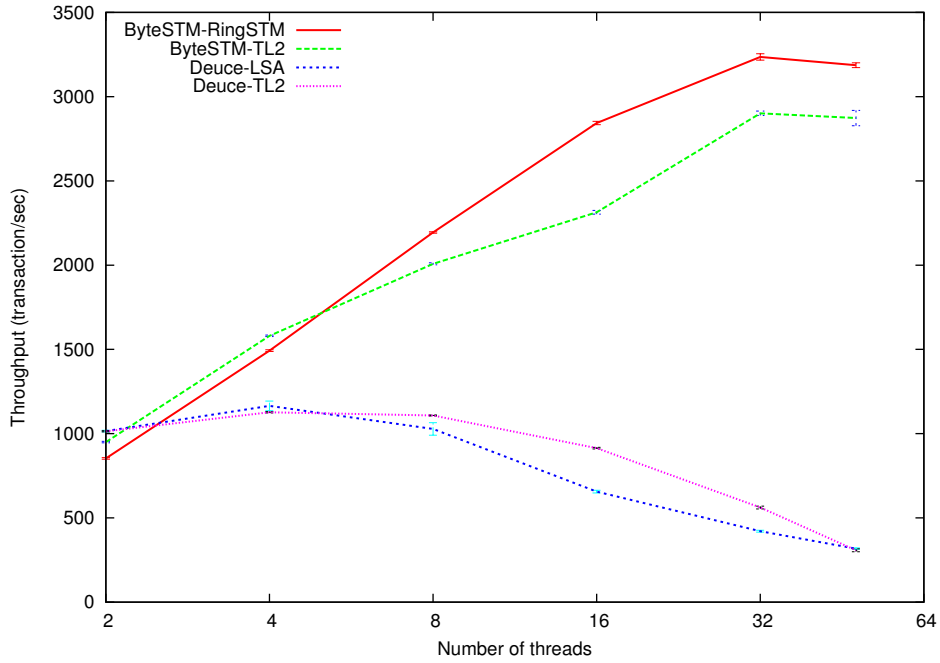
We also conducted four experiments, each with the same read/write ratio: 80% reads to 20% writes, 50% reads to 50% writes, 20% reads to 80% writes, and 100% writes without reads.

Figure 5.4 shows the results. The y-axes represents the throughput, and the x-axes represents the number of threads. From the figure, we observe that, ByteSTM with RingSTM achieves the best performance and scalability, followed by ByteSTM with TL2, and then Deuce. The high ratio of aborts and relatively high number of writes significantly affects Deuce's performance. In this benchmark, we observe that Deuce does not scale.

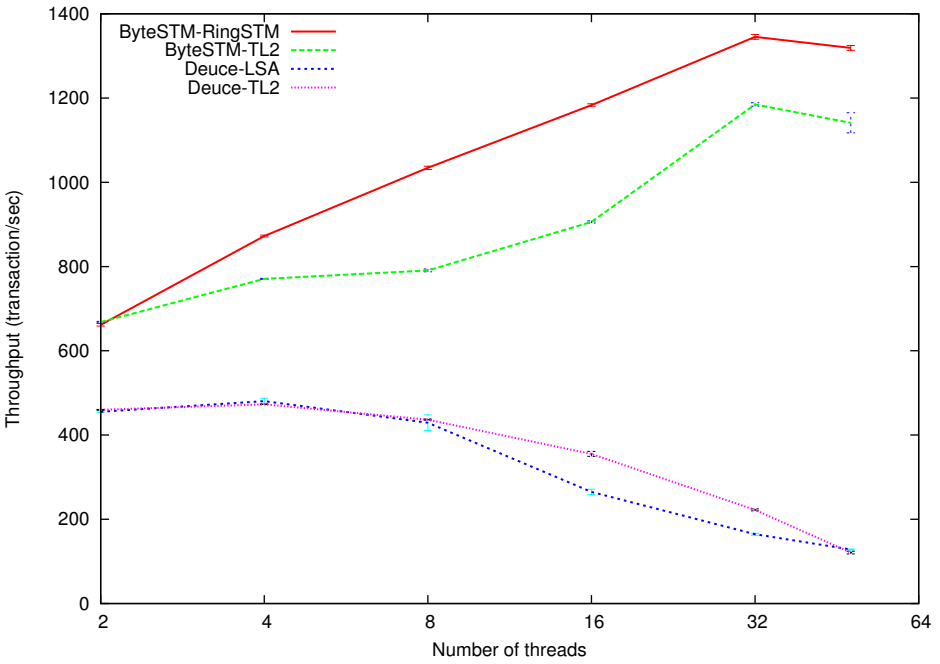
ByteSTM with TL2 outperforms Deuce by an average of 267%. Moreover, ByteSTM with RingSTM outperforms Deuce by an average of 325%. In this experiment, the gap between TL2 and RingSTM is not very large. This is because, the number of reads is small and the abort ratio is high.

5.3 Macro Benchmarks

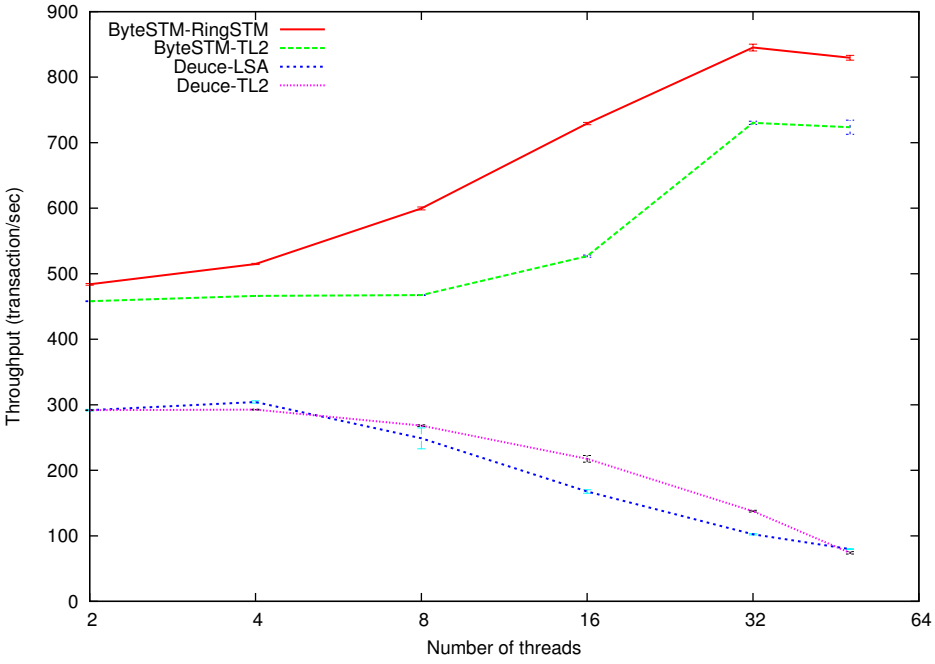
In this section, we evaluate the performance under macro-benchmarks including a Bank application and five applications from the STAMP benchmark suite [17] (Vacation, KMeans, Genome, Labyrinth and Intruder).



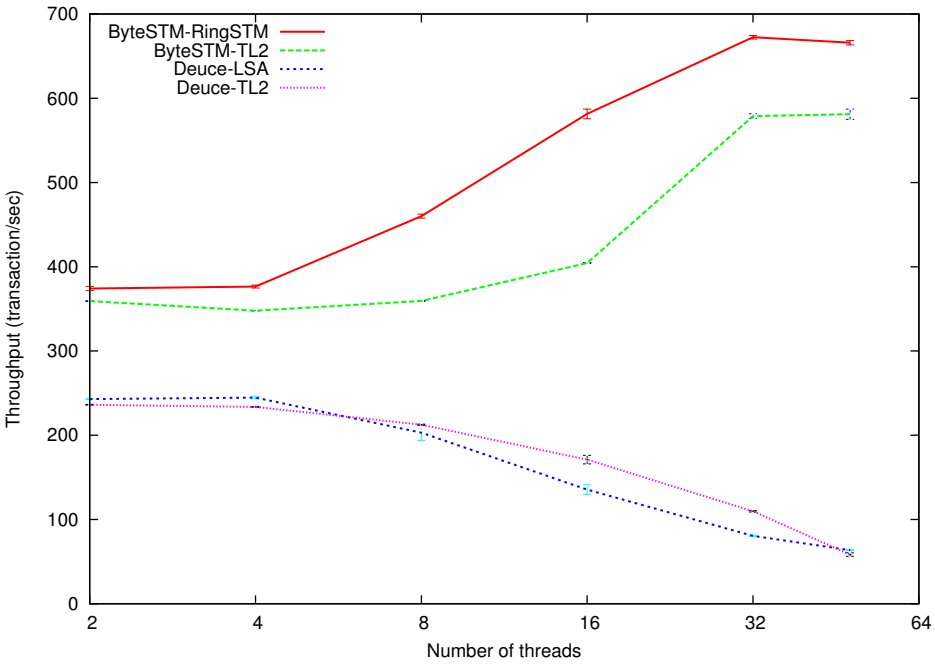
(a) 20% writes.



(b) 50% writes.



(c) 80% writes.



(d) 100% writes.

Figure 5.4: Throughput under Hash Set.

5.3.1 Bank

The Bank benchmark simulates a subset of banking operations. The benchmark is initialized with a set of accounts and an initial deposit in each account. The following operations can be done on each account: check the account balance, transfer money from one account to another, and add an interest to an account.

In our experiments, we used 8 accounts. Each thread checks the balance of all accounts, adds an interest to all the accounts, or transfers money from one random account to another random one. So, the Bank benchmark operations are characterized by a medium number of reads (from 26 to 42), and a medium number of writes (from 11 to 22). This results in medium-length transactions. Moreover, transactions suffer a high number of aborts (abort ratio is from 188% to 268%), due to the small number of accounts that all threads are contending on.

We conducted four experiments, each with a different read/write ratio: 80% reads to 20% writes, 50% reads to 50% writes, 20% reads to 80% writes, and 100% writes without reads.

Figure 5.5 shows the results. The y-axes represents the throughput, and the x-axes represents the number of threads. From the figure, we observe that, in all cases, ByteSTM with RingSTM achieves the best performance, followed by ByteSTM with TL2, and then Deuce. The benchmark does not scale well for all STMs, due to the high number of aborts in all cases.

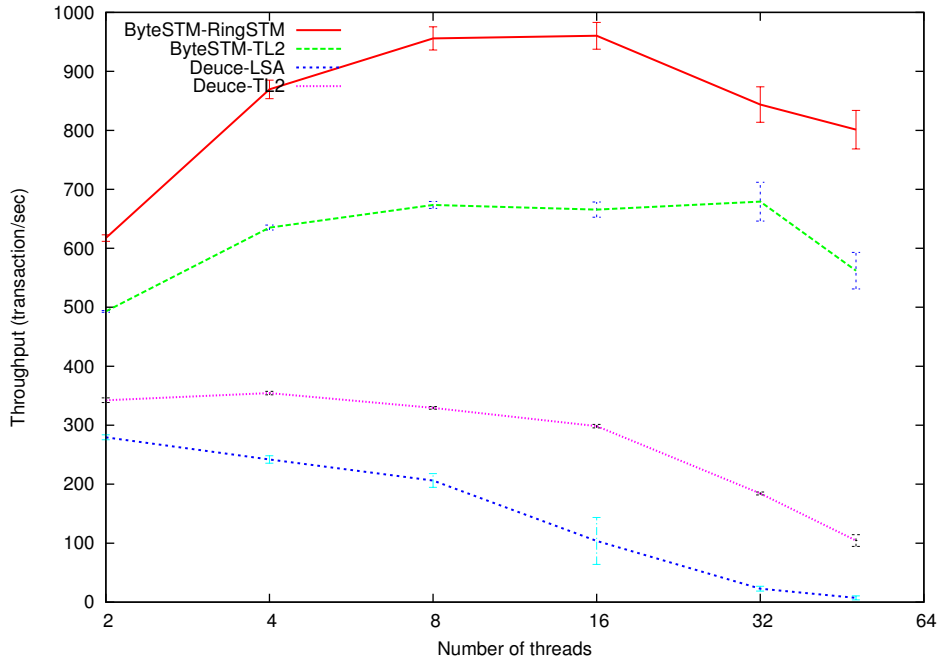
ByteSTM with TL2 outperforms Deuce by an average of 170%. Moreover, RingSTM outperforms Deuce by an average of 261%. In this experiment, the gap between TL2 and RingSTM is moderate since the number of reads is moderate.

5.3.2 Vacation

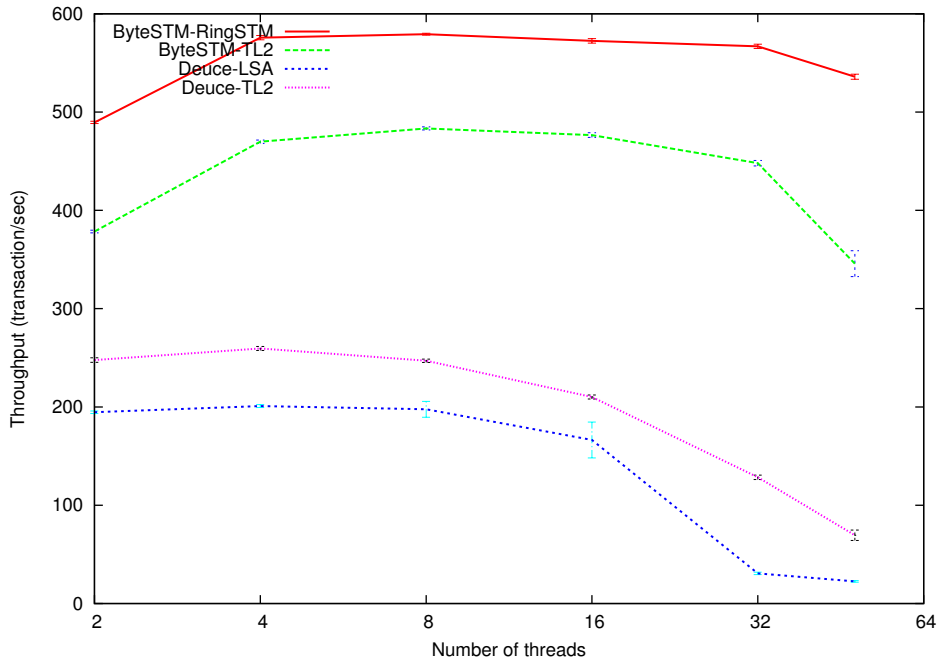
The Vacation benchmark [17] emulates a travel reservation system, similar to what is done in online transaction processing. In this benchmark, clients (i.e., threads) perform a specific number of sessions. Each client does the following operations on the in-memory database: reserve, cancel, create, or destroy an item. In each session, a client operates on a specific number of items. The benchmark is characterized by medium-length transactions, medium read-sets, medium write-sets, and long transaction times in comparison with other STAMP benchmarks.

We conducted two experiments: one with low contention (a small number of operations per session, a small ratio of create/destroy operations, and operations are performed on a smaller portion of the in-memory database), and the other with high contention.

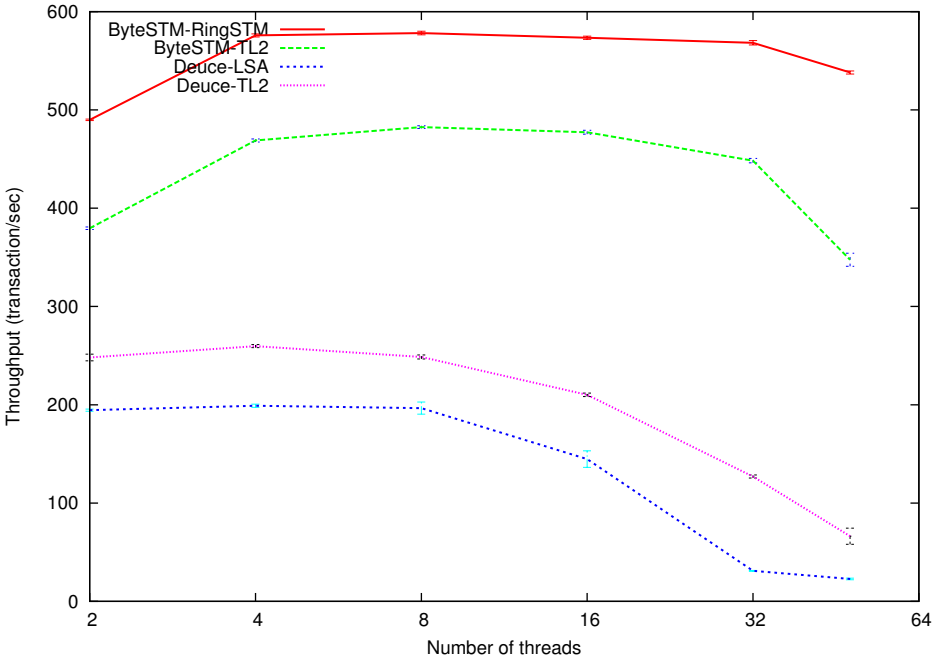
Figure 5.6 shows the results. The y-axes represents the time taken to complete the experiment in milliseconds, and the x-axes represents the number of threads. We observe that



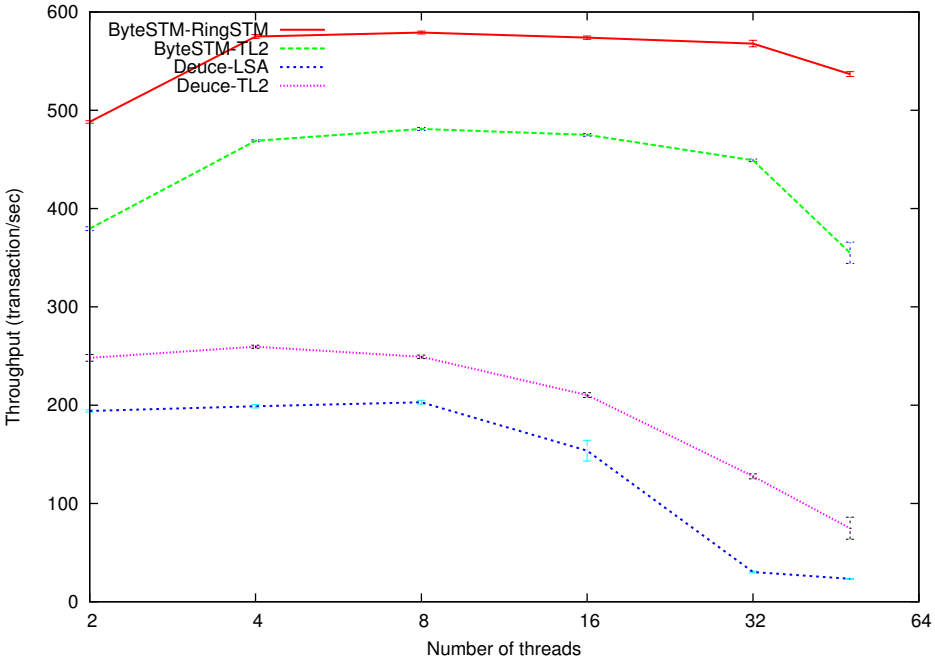
(a) 20% writes.



(b) 50% writes.

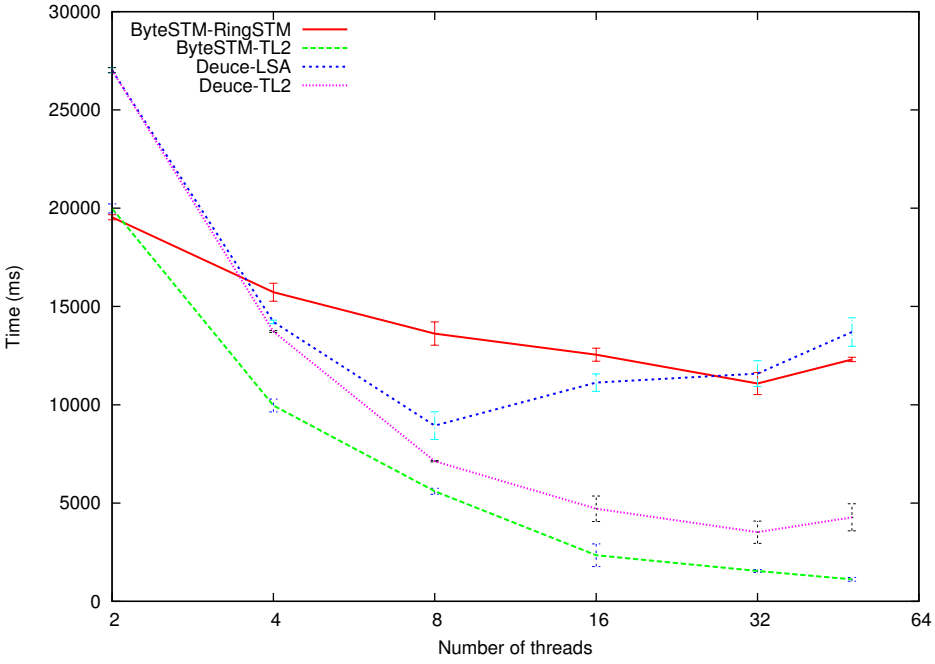


(c) 80% writes.

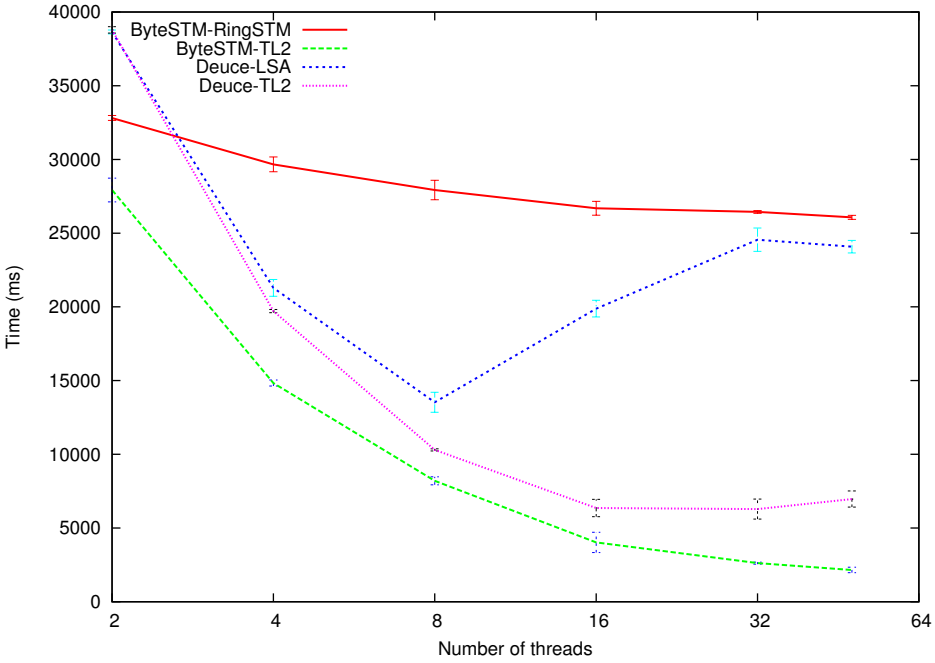


(d) 100% writes.

Figure 5.5: Throughput under a Bank application.



(a) Low Contention



(b) High Contention

Figure 5.6: Execution time under Vacation.

ByteSTM with TL2 has the best performance and scalability under both low and high contention conditions. ByteSTM with RingSTM suffers from extremely high number of aborts due to false positives and long transactions. ByteSTM with TL2 outperforms Deuce by an average of 102% in low contention and 86% in high contention.

5.3.3 KMeans

The KMeans benchmark [17] implements the K-means clustering algorithm [62], which is used in application domains including data mining, geostatistics, computer vision and market segmentation In this benchmark, the K-means algorithm is applied on an input file that consists of a number of points of a given dimension. The algorithm finds the clusters of these points with a given convergence threshold.

The benchmark is characterized by short transaction lengths, small read-sets, small write-sets, and short transaction times in comparison with other STAMP benchmarks.

We conducted two experiments: one with low contention, and the other with high contention.

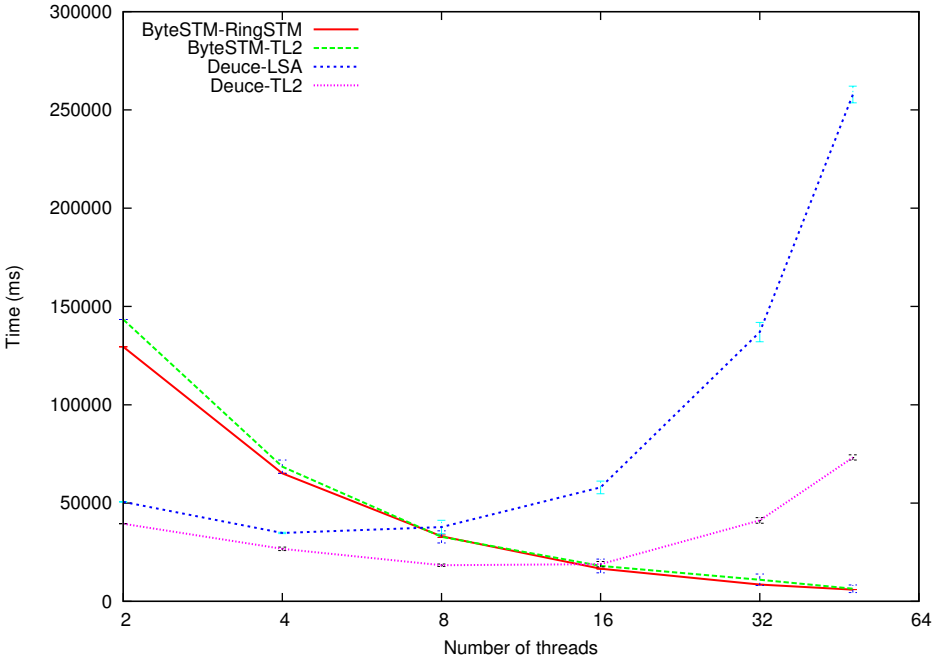
Figure 5.7 shows the results. The y-axes represents the time taken to complete the experiment in milliseconds, and the x-axes represents the number of threads. We observe that ByteSTM with TL2 and RingSTM scales well in both cases and yield similar performance. However, in low contention, Deuce performs better for up to 8 threads, and then its performance degrades quickly.

5.3.4 Genome

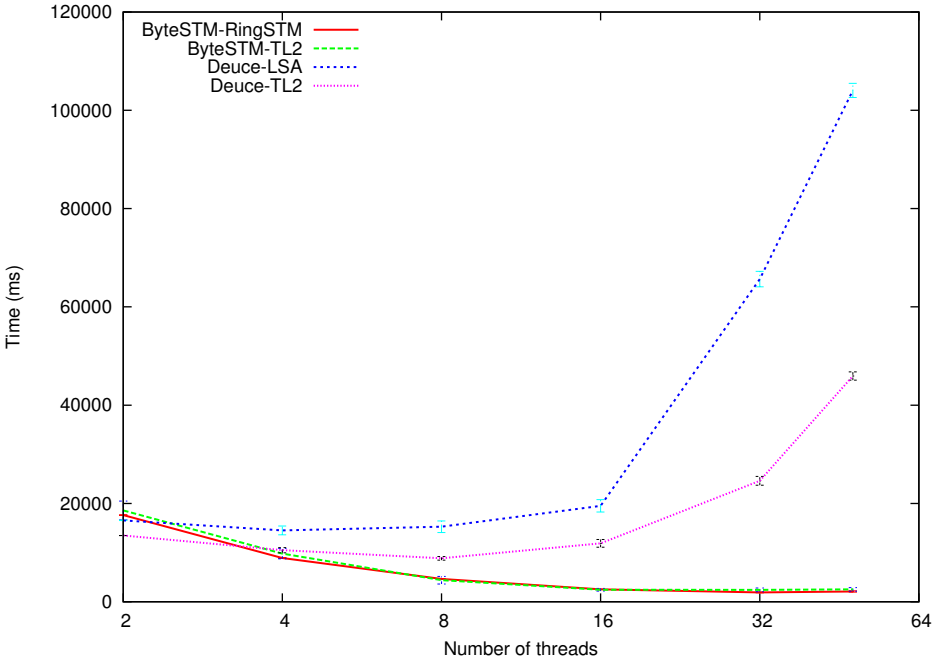
The Genome benchmark [17] performs gene sequencing. In this benchmark, gene segments of a given set of nucleotides are sampled from a gene with another set of nucleotides. Additionally, a given number of segments are analyzed to reconstruct the original gene.

The benchmark is characterized by medium transaction lengths, medium read-sets, medium write-sets, long transaction times, and low contention in comparison with other STAMP benchmarks. These characteristics are similar to that of the Vacation benchmark with low contention.

Figure 5.8 shows the results. The y-axes represents the time taken to complete the experiment in milliseconds, and the x-axes represents the number of threads. We observe that ByteSTM/TL2 and Deuce performs the same. This graph is limited to 16 threads because the benchmark does not work with higher number of threads. Thus, the scalability of the STMs is not clear. ByteSTM with RingSTM suffers from extremely higher number of aborts due to false positives and long transactions, which is similar to the behavior observed on the Vacation benchmark.



(a) Low Contention



(b) High Contention

Figure 5.7: Execution time under KMeans.

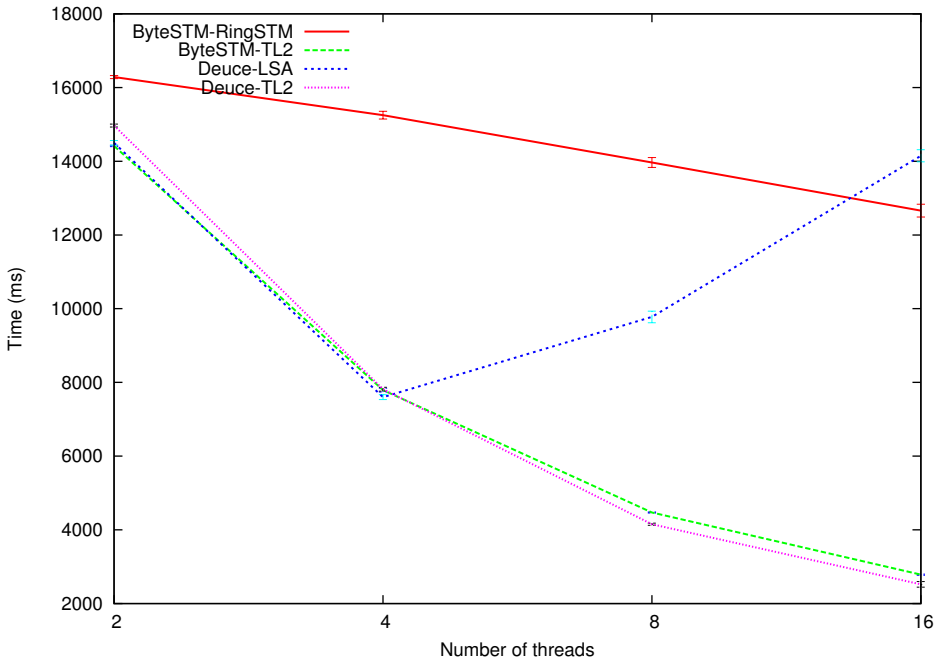


Figure 5.8: Execution time under Genome.

5.3.5 Labyrinth

The Labyrinth benchmark [17] routes paths in a 3D maze. Each thread picks a start and an end point, and then it connects between them through the maze. Calculating this path and adding it to the global maze is done in a single transaction. Thus conflicts occur when two threads find overlapping paths.

In the benchmark, a 3D maze file is given and it is required to route a specific number of routes. The benchmark is characterized by long transaction lengths, large read-sets, large write-sets, long transaction times, and very high contention in comparison with other STAMP benchmarks.

Figure 5.9 shows the results. We observe that ByteSTM with TL2 achieves the best performance and scalability. Deuce with LSA does not work after 16 threads. ByteSTM with RingSTM suffers from an extremely higher number of aborts due to false positives and long transactions. However, the high contention nature of this benchmark (i.e., all STMs suffer from high abort ratio) compensates for the false positive effect and reduces the gap between TL2 and RingSTM. ByteSTM with TL2 outperforms Deuce by an average of 50%.

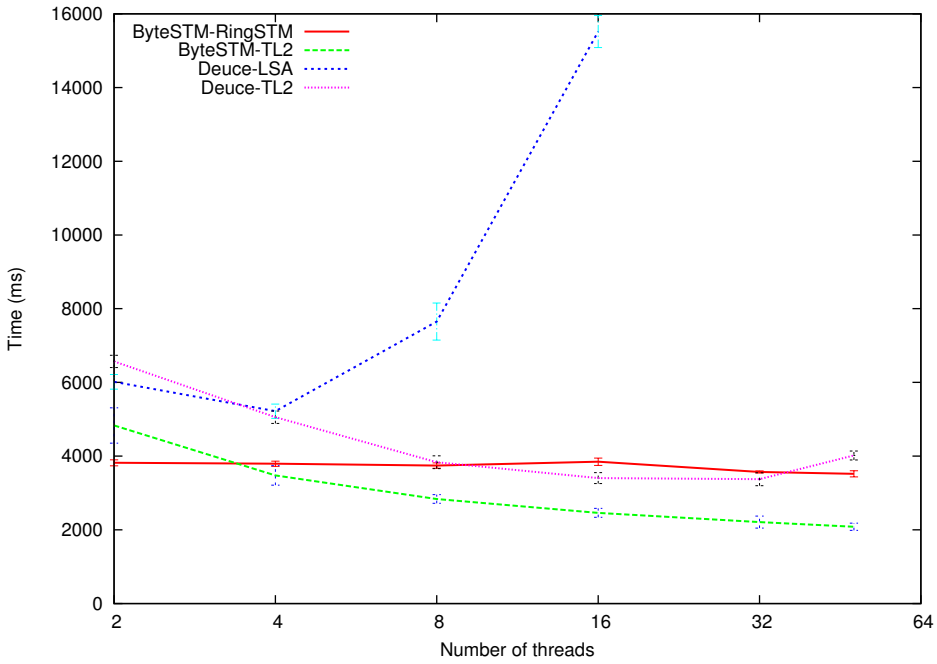


Figure 5.9: Execution time under Labyrinth.

5.3.6 Intruder

The Intruder benchmark [17] detects network intrusions. It runs a signature-based network intrusion detection system that scans network packets, and tries to match them against predefined intrusion signatures. Network packets are scanned in parallel through three phases: capture, reassembly, and detection. The reassembly phase uses a global data structure, which is subject to contention.

The benchmark is characterized by short transaction lengths, medium read-sets, medium write-sets, medium transaction times, and high contention in comparison with other STAMP benchmarks.

Figure 5.10 shows the results. The y-axes represents the time taken to complete the experiment in milliseconds, and the x-axes represents the number of threads. We observe that ByteSTM with TL2 achieves the best performance. ByteSTM with RingSTM suffers from increased aborts due to false positives. ByteSTM with TL2 outperforms Deuce by an average of 36%. (We calculate this up to 16 threads only, because Deuce's performance degrades exponentially after 16 threads.)

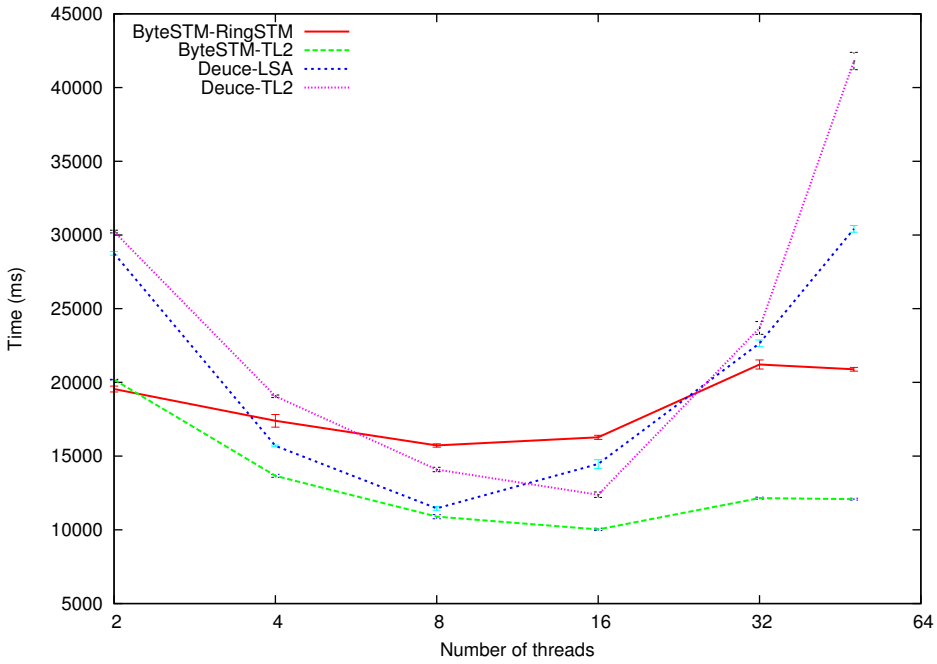


Figure 5.10: Execution time under Intruder.

5.4 Highly Concurrent Data Structures

We now compare the performance of ByteSTM against a set of highly concurrent data structures, which are (manually) fine-tuned to obtain the best performance.

5.4.1 Lock-Free Linked List

In this set of experiments, we compare ByteSTM against Michael's Lock-Free Linked List [67].

Figure 5.11 shows the results. The y-axes represents the throughput, and the x-axes represents the number of threads. From the figure, we observe that the Lock-Free Linked List outperforms ByteSTM by a large order of magnitude. As mentioned in Section 5.2.1, the linked list implementation used to test ByteSTM is not optimized. It uses one transaction for each operation. Each operation keeps all visited nodes in the transaction read-set, which increases the number of conflicts and aborts. Lock-Free Linked List outperforms ByteSTM by an average of 600%.

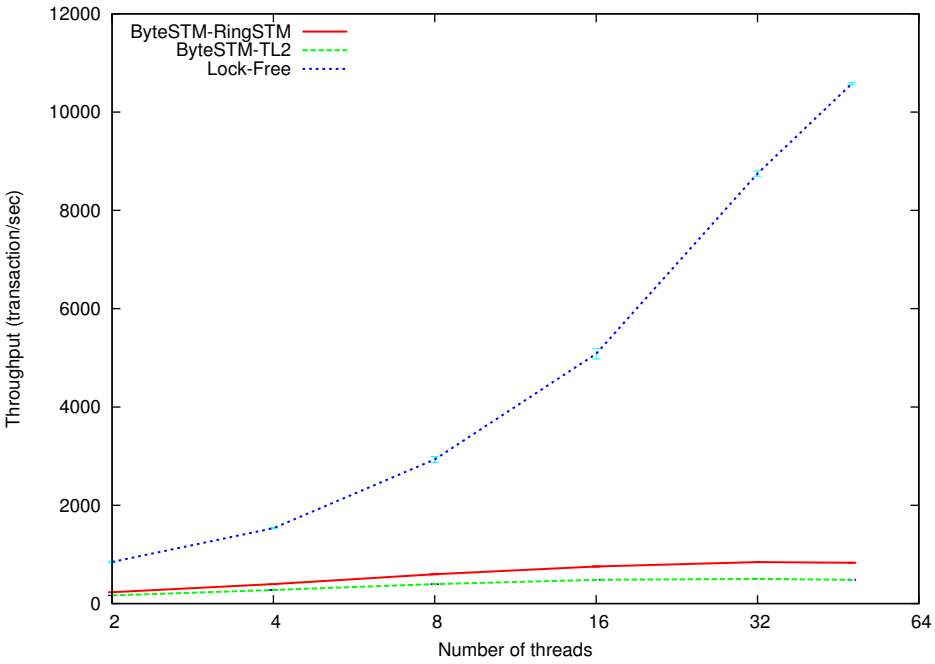


Figure 5.11: Lock-Free Linked List.

5.4.2 Lock-Free Skip List

We now compare ByteSTM against the Lock-Free Skip List implementation that uses Michael's Lock-Free Linked List [67] to implement the list at each level [61].

The results are shown in Figure 5.12. The y-axes represents the throughput, and the x-axes represents the number of threads. We observe that the Lock-Free Skip List outperforms ByteSTM, but the gap is small. Also, ByteSTM with TL2 scales similarly to that of the Lock-Free Skip List. Lock-Free Skip List outperforms ByteSTM by an average of 220%.

5.4.3 Lock-Free Binary Search Tree

We now compare ByteSTM against the Lock-Free Binary Search Tree (BST), which is based on Multi-word Compare-And-Swap (MCAS) [29].

Figure 5.13 shows the results. The y-axes represents the throughput, and the x-axes represents the number of threads. We observe that the Lock-Free BST outperforms ByteSTM, but the gap is small at smaller number of threads. Lock-Free BST scales better than ByteSTM, and outperforms ByteSTM by an average of 105%.

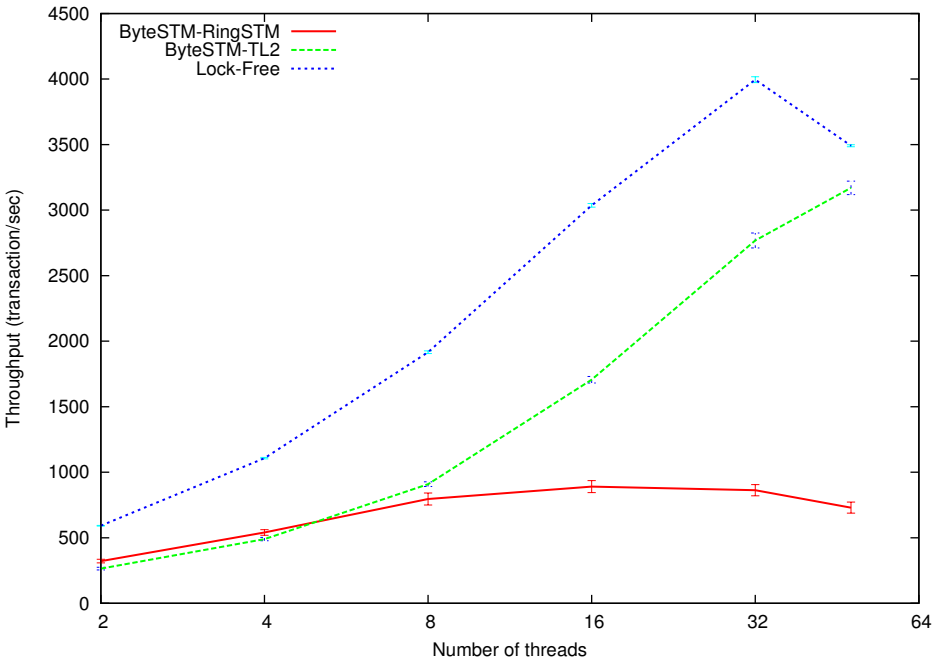


Figure 5.12: Lock-Free Skip List.

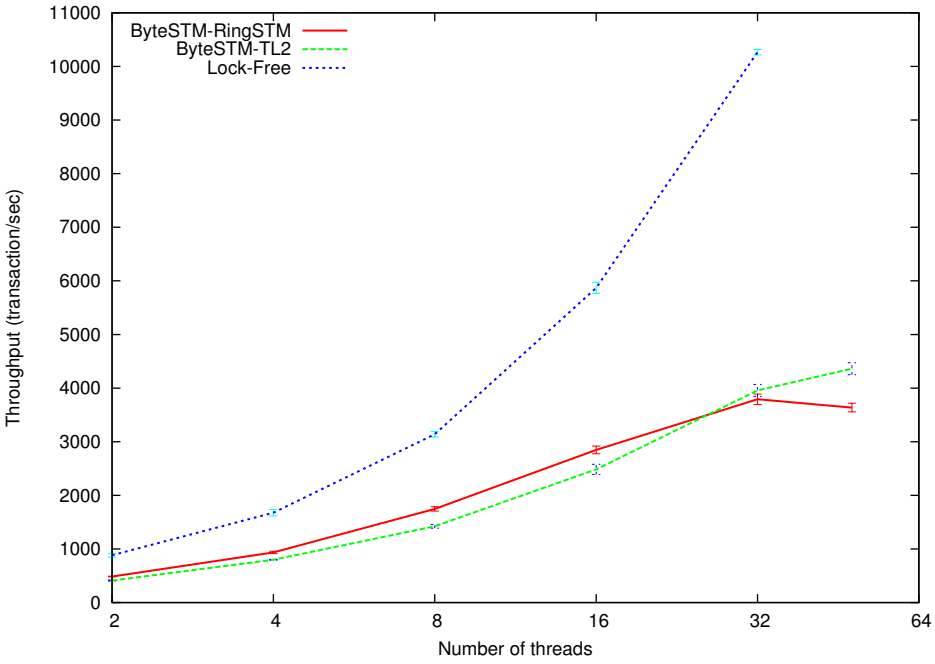


Figure 5.13: Lock-Free BST.

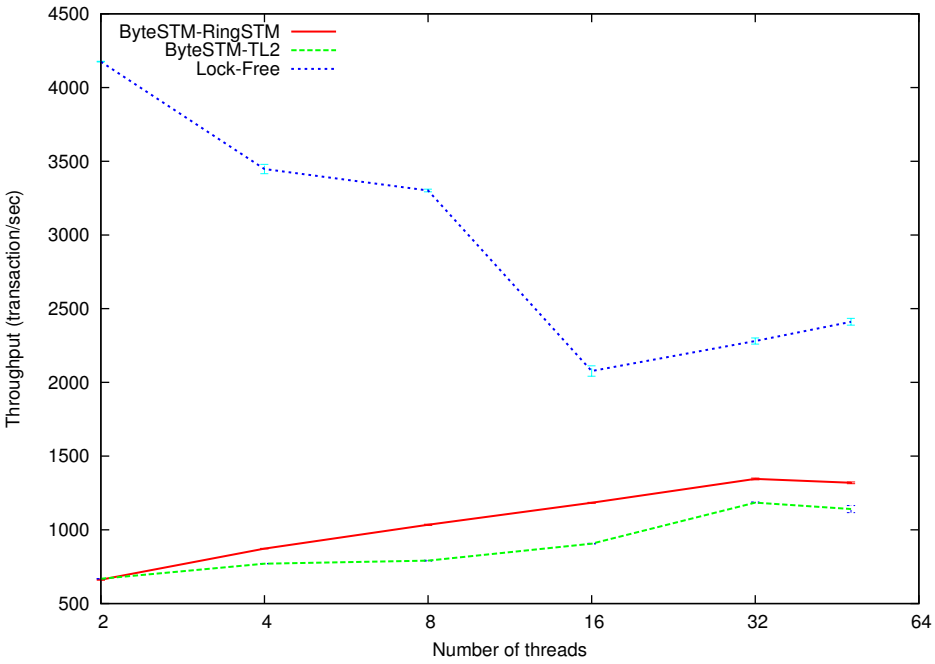


Figure 5.14: Concurrent Hash Set.

5.4.4 Concurrent Hash Set

We also compare ByteSTM against Lea's Concurrent Hash Map that is implemented in the Java Concurrency Package [56]. Lea's algorithm is a lock-based chained hashing, but it uses a small number of high-level locks.

Figure 5.14 shows the results. The y-axes represents the throughput, and the x-axes represents the number of threads. We observe that Lea's algorithm does not scale well. At higher number of threads, it's performance is comparable to ByteSTM. Lea's algorithm outperforms ByteSTM by an of average of 215%.

5.5 Summary

Implementing STM at the VM level improved the performance over other STMs by an overall average of 75%. On micro-benchmarks, ByteSTM improves throughput over others by 20% to 75%, for up to 16 threads. After 16 threads, the scalability of competitor STMs is significantly degraded, and ByteSTMs average improvement is observed to be approximately 270%. On macro-benchmarks, ByteSTMs improvement ranges from 36% to 100%. Moreover, the scalability is significantly better. We found that ByteSTM, in general, is better when the abort ratio is high and under high contention conditions.

The RingSTM algorithm performs well, irrespective of the number of reads. However, its performance is highly sensitive to false positives when the number of writes increases.

The TL2 algorithm performs well when the number of reads is not large. Additionally, it maintains good performance and scales well, when the number of writes increases.

The comparison against highly concurrent data structures shows that an acceptable performance can be achieved using STM, however, STM is clearly not competitive. This reinforces the result that, STM is beneficial, in general, for transactional workloads, which usually involve operations on multiple data structures. Additionally, the programming effort needed for STM is minimal. In contrast, highly concurrent data structures suffer from poor programmability.

Chapter 6

Conclusion and Future Work

In this thesis, we present ByteSTM, a Java STM implementation at the virtual machine level. ByteSTM supports two algorithms: TL2 and RingSTM. It supports implicit transactions transparently with or without compiler support.

ByteSTM takes advantage of being a VM-level implementation. It accesses memory directly and uses absolute memory address to uniformly handle memory. Further, it eliminates the GC overhead, by manually allocating and recycling memory for transactional metadata. ByteSTM uses field-based granularity, which has the highest scalability. It uses the thread header to store transactional metadata, instead of the slower standard Java `ThreadLocal` abstraction. Memory load/store bytecode instructions automatically switch to transactional mode when a transaction starts, and switch back to normal mode when the transaction commits successfully. ByteSTM supports atomic blocks anywhere in the code. Thus, ByteSTM overcomes limitations of other Java STM implementations.

Experimental results show that, on micro-benchmarks, ByteSTM improves throughput over state-of-the-art competitor STMs including Deuce, ObjectFabric, Multiverse, DSTM2, and JVSTM by 20% to 75%, for up to 16 threads. After 16 threads, the scalability of competitor STMs is significantly degraded, and ByteSTM's average improvement is observed to be approximately 270%. On macro-benchmarks, ByteSTM's improvement ranges from 36% to 100%.

At its core, our work shows that implementing a Java STM at the VM-level is indeed possible, and can yield significant performance benefits. This is because, at the VM-level, STM overhead is significantly reduced. Additionally, memory operations are faster, the GC overhead is eliminated, no instrumentation is required (i.e., the same code can run in two modes: transactional and non-transactional). Moreover, atomic blocks can be supported anywhere, and metadata is attached to the thread header. Furthermore, strong atomicity is easier to implement, since the VM has full control over all transactional and non-transactional memory operations. Also, irrevocable transactions can be detected automatically and han-

dled efficiently: a transaction can be automatically converted to an irrevocable one when it performs any irrevocable action, and can be guaranteed to finish and commit successfully.[1]

These optimizations are not possible at a library-level. Moreover, compiler-level STM for VM languages cannot support these optimizations also. Thus, implementing an STM for a VM language such as Java at the VM-level is likely the most "performant".

6.1 Future Work

Several directions exist for future work. These include the following:

- Currently, ByteSTM does not support advanced TM features such as nested transactions, irrevocable transactions [101] (e.g., transactions with I/O operations), and strong atomicity [12]. Each of these are immediate directions for further work.
- In our experiments, we only used micro-benchmarks and the STAMP macro-benchmark suite. It is highly desirable to evaluate ByteSTM on a more wider suite of benchmarks and other applications (e.g., STMBench7 [38], RMS-TM [54, 53], Lee-TM [8]).
- ByteSTM does not support plugging other STM algorithms transparently (e.g., by extending an abstract class). This is a desirable feature from an extensibility standpoint, and also as a research and experimentation platform for evaluating new STM algorithms.
- ByteSTM uses the Jikes Baseline Compiler [96, 5] which does not support many optimizations (e.g., inlining, local optimization, control optimization, global optimization). This compiler is simply used to translate from bytecode to native code. An immediate direction for future work is being able to use the Jikes Optimizing Compiler [97, 5] for improved performance.

[1] Currently, ByteSTM does not support strong atomicity or irrevocable transactions. This is planned as future work.

Bibliography

[1] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. *ACM SIGPLAN Notices*, 44(4):185–196, 2009.

[2] A. Adl-Tabatabai et al. The StarJIT compiler: A dynamic compiler for managed runtime environments. *Intel Technology Journal*, 2003.

[3] A. Adl-Tabatabai, B. Lewis, V. Menon, B. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *ACM SIGPLAN Notices*, volume 41, pages 26–37. ACM, 2006.

[4] A. Adl-Tabatabai and T. Shpeisman. Draft specification of transactional language constructs for C++, version 1.0, 2009.

[5] B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The Jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44:399–417, January 2005.

[6] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[7] C. Ananian, K. Asanovic, B. Kuszmaul, C. Leiserson, and S. Lie. Unbounded transactional memory. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 316 – 327, feb. 2005.

[8] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Lujn, and K. Jarvis. Lee-TM: A Non-trivial Benchmark Suite for Transactional Memory. In A. Bourgeois and S. Zheng, editors, *Algorithms and Architectures for Parallel Processing*, volume 5022 of *Lecture Notes in Computer Science*, pages 196–207. Springer Berlin / Heidelberg, 2008.

[9] Apache Software Foundation. BCEL: Byte-code engineering library. `http://jakarta.apache.org/bcel/manual.html`.

[10] W. Binder, J. Hulaas, and P. Moret. Advanced Java bytecode instrumentation. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 135–144. ACM, 2007.

[11] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[12] C. Blundell, E. Lewis, and M. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2):17–17, 2006.

[13] Boost C++ Libraries Group. Boost C++ Libraries. `http://www.boost.org/`, 2011.

[14] B. J. Bradel and T. S. Abdelrahman. The use of hardware transactional memory for the trace-based parallelization of recursive java programs. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 101–110, New York, NY, USA, 2009. ACM.

[15] T. Brown and J. Helga. Non-blocking k-ary search trees. Technical report, Technical Report CSE-2011-04, York University, 2011. Appendix and code available at `http://www.cs.utoronto.ca/~tabrown/ksts/`, 2011.

[16] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

[17] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[18] B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *ACM SIGPLAN Notices*, volume 41, pages 1–13. ACM, 2006.

[19] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 227–238, Washington, DC, USA, 2006. IEEE Computer Society.

[20] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance SPARC CMT processor. *Micro, IEEE*, 29(2):6–16, 2009.

[21] D. Christie, J. Chung, S. Diestelhorst, M. Hohmuth, M. Pohlack, C. Fetzer, M. Nowack, T. Riegel, P. Felber, P. Marlier, et al. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *Proceedings of the 5th European conference on Computer systems*, pages 27–40. ACM, 2010.

[22] M. Cierniak et al. The open runtime platform: A flexible high-performance managed runtime environment. *Intel Technology Journal*, 2003.

[23] L. Dalessandro, M. Spear, and M. Scott. NOrec: streamlining STM by abolishing ownership records. In *ACM SIGPLAN Notices*, volume 45, pages 67–78. ACM, 2010.

[24] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[25] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[26] A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *ACM SIGPLAN Notices*, volume 44, pages 155–165. ACM, 2009.

[27] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *Parallel and Distributed Systems, IEEE Transactions on*, 21(12):1793–1807, 2010.

[28] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[29] K. Fraser. *Practical lock-freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579, 2004.

[30] K. Fraser and T. Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems (TOCS)*, 25(2):5, 2007.

[31] V. Gajinov, F. Zyulkyarov, O. S. Unsal, A. Cristal, E. Ayguade, T. Harris, and M. Valero. QuakeTM: parallelizing a complex sequential application using transactional memory. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 126–135, New York, NY, USA, 2009. ACM.

[32] GNU Compiler Collection. Transactional Memory in GCC. `http://gcc.gnu.org/wiki/TransactionalMemory`, 2011.

[33] J. Gottschlich and D. Connors. DracoSTM: A practical C++ approach to software transactional memory. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, pages 52–66. ACM, 2007.

[34] J. E. Gottschlich et al. Toward.Boost.STM (TBoost.STM). `http://eces.colorado.edu/~gottschl/tboostSTM/`, 2009.

[35] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. *Distributed Computing*, pages 303–323, 2005.

[36] R. Guerraoui, M. Herlihy, and B. Pochon. Toward a theory of transactional contention managers. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 258–264. ACM, 2005.

[37] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184. ACM, 2008.

[38] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the Second European Systems Conference (EuroSys2007)*, 2007.

[39] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[40] T. Harris and K. Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Notices*, volume 38, pages 388–402. ACM, 2003.

[41] T. Harris, J. Larus, and R. Rajwar. Transactional memory. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[42] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *ACM SIGPLAN Notices*, volume 41, pages 14–25. ACM, 2006.

[43] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *ACM SIGPLAN Notices*, volume 41, pages 253–262. ACM, 2006.

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

[45] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. *ACM SIGARCH computer architecture news*, 21(2):289–300, 1993.

[46] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[47] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch Hashing. In G. Taubenfeld, editor, *Distributed Computing*, volume 5218 of *Lecture Notes in Computer Science*, pages 350–364. Springer Berlin / Heidelberg, 2008.

[48] R. Hickey. Clojure Refs and Transactions. `http://clojure.org/refs`, 2010.

[49] B. Hindman and D. Grossman. Atomicity via source-to-source translation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 82–91. ACM, 2006.

[50] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M. `http://www.intel.com/Assets/en_US/PDF/manual/253666.pdf`, 2007.

[51] Intel Corporation. Intel Itanium Architecture Software Developers Manual Volume 3: Instruction Set Reference. `http://download.intel.com/design/Itanium/manuals/24531905.pdf`, 2007.

[52] Intel Corporation. Intel C++ STM Compiler 4.0, Prototype Edition. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`, 2009.

[53] G. Kestor, V. Karakostas, O. S. Unsal, A. Cristal, I. Hur, and M. Valero. RMS-TM: a comprehensive benchmark suite for transactional memory systems. In *Proceedings of the second joint WOSP/SIPEW international conference on Performance engineering*, ICPE '11, pages 335–346, New York, NY, USA, 2011. ACM.

[54] G. Kestor, S. Stipic, O. Unsal, A. Cristal, and M. Valero. RMS-TM: A transactional memory benchmark for recognition, mining and synthesis applications. In *TRANSACT09: 4th workshop on transactional computing*, 2009.

[55] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[56] D. Lea. Java concurrency package concurrent hash map in java.util.concurrent. `http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/`.

[57] Y. Lev and J.-W. Maessen. Split hardware transactions: true nesting of transactions using best-effort hardware transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 197–206, New York, NY, USA, 2008. ACM.

[58] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology, 2004.

[59] D. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *ACM Sigplan Notices*, volume 12, pages 128–137. ACM, 1977.

[60] D. Lupei, B. Simion, D. Pinto, M. Misler, M. Burcea, W. Krick, and C. Amza. Transactional memory support for scalable and transparent parallelization of multiplayer games. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 41–54, New York, NY, USA, 2010. ACM.

[61] Y. L. M. Herlihy and N. Shavit. A lock-free concurrent skiplist with wait-free search. In *Unpublished Manuscript*. Sun Microsystems Laboratories, Burlington, Massachusetts, 2007.

[62] J. MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, page 14. California, USA, 1967.

[63] V. Marathe, W. Scherer, and M. Scott. Adaptive software transactional memory. *Distributed Computing*, pages 354–368, 2005.

[64] V. Marathe, M. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. Scherer III, and M. Scott. Lowering the overhead of nonblocking software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.

[65] A. McDonald, J. Chung, H. Chafi, C. Minh, B. Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on chip-multiprocessors. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 63–74. IEEE, 2005.

[66] F. Meawad, R. Macnak, and J. Vitek. Collecting transactional garbage. In *TRANSACT*, 2011.

[67] M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM, 2002.

[68] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: log-based transactional memory. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 254 – 265, feb. 2006.

[69] J. Moss and A. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

[70] C. Noël. Extensible software transactional memory. In *Proceedings of the Third C* Conference on Computer Science and Software Engineering*, pages 23–34. ACM, 2010.

[71] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for Java. In *Compiler Construction*, pages 138–152. Springer, 2003.

[72] ObjectFabric Inc. ObjectFabric. `http://objectfabric.com`, 2011.

[73] M. Olszewski, J. Cutler, and J. Steffan. JudoSTM: A dynamic binary-rewriting approach to software transactional memory. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 365–375. IEEE, 2007.

[74] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th annual international symposium on Computer architecture*, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM.

[75] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33:668–676, June 1990.

[76] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 494 – 505, june 2005.

[77] H. E. Ramadan, C. J. Rossbach, D. E. Porter, O. S. Hofmann, A. Bhandari, and E. Witchel. MetaTM/TxLinux: transactional memory for an operating system. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 92–103, New York, NY, USA, 2007. ACM.

[78] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. *Distributed Computing*, pages 284–298, 2006.

[79] T. Riegel, P. Felber, and C. Fetzer. LSA-STM. `http://tmware.org/lsastm`, 2006.

[80] T. Riegel, P. Felber, and C. Fetzer. TinySTM. `http://tmware.org/tinystm`, 2010.

[81] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. *TRANSACT06*, 298, 2006.

[82] B. Saha, A. Adl-Tabatabai, A. Ghuloum, M. Rajagopalan, R. Hudson, L. Petersen, V. Menon, B. Murphy, T. Shpeisman, E. Sprangle, et al. Enabling scalability and performance in a large scale CMP environment. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 73–86. ACM, 2007.

[83] B. Saha, A. Adl-Tabatabai, R. Hudson, C. Minh, and B. Hertzberg. 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*, pages 187–197. ACM, 2006.

[84] W. Scherer III and M. Scott. Contention management in dynamic software transactional memory. In *PODC Workshop on Concurrency and Synchronization in Java programs*, pages 70–79, 2004.

[85] W. Scherer III and M. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005.

[86] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213. ACM, 1995.

[87] R. L. Sites. Alpha AXP architecture. *Commun. ACM*, 36:33–44, February 1993.

[88] M. Spear, L. Dalessandro, V. Marathe, and M. Scott. A comprehensive strategy for contention management in software transactional memory. In *ACM Sigplan Notices*, volume 44, pages 141–150. ACM, 2009.

[89] M. Spear, V. Marathe, L. Dalessandro, and M. Scott. Privatization techniques for software transactional memory. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 338–339. ACM, 2007.

[90] M. Spear, A. Shriraman, L. Dalessandro, and M. Scott. Transactional mutex locks. In *SIGPLAN Workshop on Transactional Computing*, 2009.

[91] M. F. Spear, M. M. Michael, and C. von Praun. RingSTM: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.

[92] J. Stone, H. Stone, P. Heidelberger, and J. Turek. Multiple reservations and the Oklahoma update. *Parallel Distributed Technology: Systems Applications, IEEE*, 1(4):58–71, nov 1993.

[93] Sun Microsystems. Java 5.0. `http://www.oracle.com/technetwork/java/javase/1-5-0-139765.html`, `http://www.oracle.com/technetwork/articles/javase/j2se15-141062.html`, 2004.

[94] Sun Microsystems. Java Annotations. `http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html`, `http://docs.oracle.com/javase/tutorial/java/javaOO/annotations.html`, 2004.

[95] Sun Microsystems. `sun.misc.Unsafe`. `http://www.docjar.com/docs/api/sun/misc/Unsafe.html`, 2006.

[96] The Jikes RVM Project. Baseline compiler. `http://jikesrvm.org/Baseline+Compiler`, 2005.

[97] The Jikes RVM Project. Optimizing compiler. `http://jikesrvm.org/Optimizing+Compiler`, 2005.

[98] University of Rochester. Rochester Software Transactional Memory. `http://www.cs.rochester.edu/research/synchronization/rstm/index.shtml`, `http://code.google.com/p/rstm`, 2006.

[99] P. Veentjer. Multiverse. `http://multiverse.codehaus.org`, 2011.

[100] A. Welc, S. Jagannathan, and A. Hosking. Transactional monitors for concurrent objects. *ECOOP 2004–Object-Oriented Programming*, pages 494–514, 2004.

[101] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 285–296, New York, NY, USA, 2008. ACM.

[102] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, and D. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, pages 261 –272, feb. 2007.