ReserveTM: Optimizing for Eager Software Transactional Memory

by

Gaurav Jain

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2013

© Gaurav Jain 2013

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Software Transactional Memory (STM) helps programmers write correct concurrent code by allowing them to identify atomic sections rather than focusing on the mechanics of concurrency control. Given code with atomic sections, the compiler and STM runtime can work together to ensure proper controlled access to shared memory. STM runtimes use either lazy or eager version management. Lazy versioning buffers transaction updates, whereas eager versioning applies updates in-place. The current set of primitives suit lazy versioning since memory needs to be accessed through the runtime. The goal of this thesis present a new set of runtime primitives that better suit eager versioned STM.

We propose a novel extension to the compiler/runtime interface, consisting of *memory reservations* and *memory releases*. These extensions enable optimizations specific to eager versioned runtimes. A memory reservation allows a transaction to perform instrumentation-free access on a memory address. A release allows a read-only address to be modified by another transaction. Together, these reduce the instrumentation overhead required to support STM and improve concurrency between readers and writers. We have implemented these primitives and evaluated its performance on the STAMP benchmarks. Our results show strong performance and scalability improvements to eager versioned algorithms.

Acknowledgements

To my advisor Dr. Patrick Lam, it has been a privilege to pursue graduate studies under your guidance, and I sincerely appreciate your continuous support and encouragement. Thank you for giving me the opportunity and intellectual freedom to explore the areas of my interest. You taught me not to fear failing, and instead just try, because without trying I would inevitably fail.

I would also like to acknowledge my readers, Dr. Wojciech Golab and Dr. Frank Tip. Thank you for all your invaluable suggestions, which have contributed tremendously to this thesis. A special thanks to Dr. Ondřej Lhoták and Dr. Peter Buhr, for allowing use of the evaluation system.

To my family, thank you for your unwavering support that enables me to achieve my goals. To my wife Swati, it is your devotion and support that has pushed me along this journey. Thank you for giving me the confidence to pursue my dreams. Finally, to my daughter Avni, you have been my constant source of joy and inspiration.

Table of Contents

List of Figures vii

1 Introduction 1

1.1 Background . . . 1
1.2 Approach . . . 4
1.3 Limitations . . . 8
1.4 Contributions . . . 9
1.5 Organization . . . 9

2 Related Work 11

2.1 Privatization and Reservations . . . 12
2.2 Releases . . . 12

3 ReserveTM 14

3.1 STM Primitives . . . 16
3.2 ReserveTM Primitives . . . 19
3.3 Further optimizations . . . 22
3.3.1 Compression . . . 22
3.3.2 Reserve Dependencies . . . 23
3.4 ByteEager . . . 25
3.4.1 Read-Write Byte-locks . . . 26
3.4.2 ReserveTM Byte-locks . . . 27

4 Experimental Evaluation 31
4.1 Benchmark Optimizations 32
4.2 Evaluation 33
4.2.1 kmeans 33
4.2.2 Yada 35
4.2.3 Vacation 39
4.2.4 Genome 40
4.2.5 Intruder 41
4.2.6 Labyrinth 42
4.2.7 SSCA2 43
5 Conclusions and Future Work 44
5.1 Future Work 44
References 46

List of Figures

1.1 Application of privatization on a linked list 3
1.2 Comparison of instrumentation primitives 6
1.3 Writer stall 7

3.1 Motivating example using with language-support for transactions 15
3.2 Motivating example instrumented with the current STM primitives 18
3.3 Motivating example instrumented with ReserveTM primitives 21
3.4 Reducing instrumentation with compression 22
3.5 Thread pool code excerpt 24
3.6 Byte-lock layout 26
3.7 ReserveTM byte-locks 28

4.1 Original iterator access in STAMP 32
4.2 Modified iterator access in STAMP 32
4.3 kmeans transaction 33
4.4 Speedup of STM threads for kmeans-low++ 34
4.5 Speedup of STM threads for kmeans-high++ 34
4.6 Speedup of STM threads for yada++ 36
4.7 Speedup of STM threads for vacation-low++ 37
4.8 Speedup of STM threads for optimized vacation-low++ 37
4.9 Speedup of STM threads for vacation-high++ 38

4.10 Speedup of STM threads for optimized vacation-high++ 38
4.11 Vacation cancellation 39
4.12 Speedup of STM threads for genome++ 40
4.13 Speedup of STM threads for intruder++ 41
4.14 Speedup of STM threads for labyrinth++ 42
4.15 Speedup of STM threads for ssca2++ 43

Chapter 1

Introduction

Concurrency control in multithreaded programs is notoriously difficult for programmers to implement correctly. Transactional memory provides a clean, high-level interface for specifying required concurrency control, in the form of atomic sections, *i.e.* transactions. Under Software Transactional Memory (STM), it is the responsibility of the developer to specify atomic sections in a program's implementation. Given this annotation, an STM runtime enables concurrent execution of transactions and ensures that atomicity and serializability is maintained while accessing shared memory.

In the presence of concurrency, runtime instrumentation could be explicitly invoked by the programmer on accesses to shared memory within a transaction. In recent work [28] [22] [8] [1], the compiler has taken responsibility for adding runtime instrumentation. The design of the STM primitives play a fundamental role in a compiler's ability to apply program transformations. In this work, we propose improvements to the runtime interface that can be easily applied without domain-specific knowledge and reduce prior restrictions imposed on the compiler.

1.1 Background

Current trends in processor technology have led to the parallel programming model being applied to almost all types of systems, ranging from high-end servers to desktop computing and even mobile phones. Despite the community's experience with parallelism, it has proven notoriously difficult to move from a sequential programming model to a parallel one. Parallel programs are harder to understand and debug, and it is especially difficult to

write parallel programs that unlock the expected performance benefits. Programmers have developed a number of synchronization tools to aid in concurrent software development, the most fundamental being a lock. Locks are low-level primitives that lack an association with the higher-level programming language. It is up to the programmer to design and enforce a lock discipline that is correct (deadlock-free, race-free, etc.) and enables concurrency between parts of a parallel program.

Transactional memory provides synchronization between program threads by allowing a programmer to define a block of code as a transaction. Transactions appear to execute in an atomic step, with memory side-effects atomically visible to other threads. They allow a programmer to focus on identifying the logic that needs to be synchronized, and leave it to the runtime to determine how to perform the synchronization safely and efficiently. As the program logic changes, the runtime adapts.

An STM runtime captures all shared memory accesses during the execution of a transaction to guarantee that once a transaction commits, there exists a valid serial ordering of transactions. A serial ordering defines a sequence by which each transaction appears to have executed sequentially and in isolation from each other. This allows the programmer to understand transactions as if they executed one at a time. The code within a transactional block should view memory as a snapshot of program state between transaction commits. The runtime enables transactions to execute in parallel, when possible, while ensuring that the serial ordering conditions are met.

When a transaction *commits*, it attempts to finalize any side-effects of an atomic block. If a transaction cannot be serialized with another transaction, there is a *conflict* and the runtime must *abort* and possibly rollback any memory side-effects. Thus a key responsibility of an STM runtime is to support rollback of uncommitted transactions and their speculative updates through version management of memory.

The two main types of version management are lazy and eager. In lazy versioning, a transaction performs memory operations on a runtime-managed snapshot of the system and buffers any updates. Usually the runtime detects conflicts in a lazy manner during the commit sequence by validating the snapshot and then applying the buffered writes. This enables *invisible readers* and/or *invisible writers*, whereby in-flight transactions are not aware of each other's read or write set during execution. Lazy versioning can result in more overhead at commit time due to conflict management, but allows for transactions to operate in isolation. Isolation improves efficiency when communication cost between threads may be high. The cost could be due to the cache line coherency protocol, processor interconnect traffic, or network communication in the case of Distributed Transactional Memory (DTM). Lazy runtimes are successful for workloads where transactions are under high contention,

```
1  List pL;
2
3  atomic {
4    pL.head = L.head;
5    L.head = NULL;
6  }
7
8  for (each n in PL) {
9    process(n);
10 }
```

(a) T_1: process list elements outside the transaction

```
1  atomic {
2    Node n;
3    n = L.head;
4
5    while (n.next) {
6      n = n.next;
7    }
8
9    process(n);
10 }
```

(b) T_2: process list elements inside a transaction

```
1  atomic {
2    Node n = L.head;
3    int x = n.val;
4    while (n.next) {
5      if (n.next.val == x)
6        n.next = n.next.next;
7      else
8        n = n.next;
9    }
10 }
```

(c) T_3: cache the head and process inside a transaction

Figure 1.1: Application of privatization on a linked list

and hence are likely to have conflicts. If a conflict is only detected at commit time, a lazy system can ensure forward progress by making sure at least one transaction will win the conflict and commit.

Conversely, eager versioning reads and writes data in-place, resulting in a need to detect conflicts right away. The runtime protects access to addresses with metadata locks, allowing it to track ownership and conflicts. A transaction accesses its snapshot of the program state with direct memory access. Thus concurrent transactions cannot have snapshots with different values for a specific address without runtime protection. Snapshots with overlapping addresses that have only been read do not conflict. However, if an address is in the write set of one transaction and in the write or read set of another, the snapshots are conflicting. A runtime can support conflicting snapshots, where values for an address are different, by enforcing ordering on transactions. If a transaction will no longer access a specific address, the address can be modified by another transaction if ordered after the initial one. Eager versioning makes direct memory access possible at the cost of early conflict detection. An undo log that records the previous value of an address is still needed to support transactions that have written to memory and may abort. The previous value is recorded either when a transaction starts of before the address is first written to.

Transactional memory can be supported through software (STM) or hardware (HTM). STM allows for a more flexible design, supporting large transactions and advanced conflict resolution policies. However, HTM offers a low overhead which STM is unable to compete with due to runtime overhead. As a result, considerable effort has been made to support *privatization* in STM runtimes. Privatization allows transactions to access memory without incurring the overhead of the runtime. Memory can be accessed within a thread's local scope or outside of a transaction context.

Figure 1.1 demonstrates where transaction T_1 privatizes the head of List L in order to process the remainder of the list outside of a transaction. T_1 records the head of the list and makes the list nodes unreachable by setting the value to $NULL$. The list is then iterated over outside the transaction using the previously recorded value of the list head. On the other hand, transaction T_2 iterates over the list within a transaction. Thus T_1 may commit while T_2 is processing a node deeper in the list, causing the node to be processed both within and outside of a transaction. T_3 utilizes privatization within a transaction, by caching the value of the head element in the stack variable `x`, and thus avoids repeated runtime entries to read the value. *Weak isolation* [24] is the model where transactional code and non-transactional code do not concurrently access the same memory addresses. Privatization is less of an issue with *strong isolation*, where the runtime is aware of the memory accesses of non-transactional code. With weak isolation though, conflicts between non-transactional and transactional code cannot be detected by a runtime as memory accesses in non-transactional code is not instrumented. Most high performance STMs adopt a weak isolation model in order to avoid the overhead of instrumenting non-transactional code. *Privatization safe runtimes* handle conflicts between non-transactional and transactional code. Supporting privatization safety is non-trivial and techniques to support it can incur undesired overhead [26].

This work explores a means to support software transactions for eager versioned runtimes, with privatization safety and efficient runtime performance.

1.2 Approach

A memory access through the STM runtime can have significant overhead. Each access has to read and process some runtime-managed metadata which can pollute the CPU cache and cause additional traffic for the cache line coherency protocol. Operations on this metadata may invoke memory fences or execute locked CPU instructions, further hampering instruction level parallelism. In addition, the presence of function calls in place of direct memory access prevents certain compiler optimizations, such as load and store reordering. These optimizations typically cannot cross function boundaries as the compiler may not have knowledge of what memory side-effects a function call may have. As a result, privatization attempts to avoid runtime overhead for memory accesses by either doing work outside of a transaction or working on a thread-local private copy of the data.

Only accesses performed within a transaction are instrumented. To access memory outside of a transaction, data accessible from within a transaction needs to be made unreachable by other transactions. T_1 in Figure 1.1 makes all the nodes of the list unreachable

by other transactions by clearing the head pointer. The remaining nodes can then be freely modified outside a transaction without interference from the runtime. Similarly, a transaction may choose to operate on a copy of data in a thread-local scope, while within a transaction and thus avoid runtime instrumentation. T_3 operates on a local copy of `L.head.val` with the local stack variable `x`. The technique utilized by T_1 usually requires programmer intervention as it requires domain-specific knowledge to understand how to make data unreachable. However, this thesis demonstrates that due to visibility of reads and writes in eager versioned systems, the privatization technique utilized by T_3 can be applied without application knowledge.

Current STM primitives (including those proposed for LLVM [8], gcc [22], and Intel [1] compilers) do not expose whether the compiler is instrumenting for an eager or a lazy versioned runtime. As a result, these primitives require instrumentation at every shared memory access. Compilers identify reads and writes to shared memory locations and replace them with STM API calls to `TM_READ` and `TM_WRITE`.

On a transactional memory access, eager versioning performs conflict resolution through metadata manipulation, but does little version management as it reads and writes memory in-place This suggests that the conflict resolution and version management can be split, allowing the runtime to only perform metadata operations. Eager algorithms tend to allow concurrency between read operations, while writes require an exclusive lock. Until the writer transaction has committed, other transactions cannot freely read or write in-place to the same address. This is because there is no knowledge that the transaction has completed all its updates, or that it may abort and rollback the updates. Thus an *encounter-time locking* [5] scheme is used to abort any other transaction that reads a locked address. If a transaction accesses a location that it has already locked for write, it simply needs to validate that its lock is still held and proceed with accessing the address for read or write. Reads may acquire a shared reader lock to enable concurrent access and block writers from modifying the address during the lifetime of reader transactions. This results in a simple commit sequence that does not need to validate the read set since the memory stored at the read addresses is never modified. Another approach to reads is to perform a read without acquiring a lock. Instead of holding a lock the address metadata may contain version number that allows a subsequent accesses to know if the value has changed since the last access. This approach requires the read set to be validated at commit time.

Analyzing this read and write behavior, we can demonstrate how privatization can be leveraged to reduce instrumentation overhead. We consider that once a transaction is free to write to an address, it has in effect privatized the address and it need no longer access the location through the runtime. We refer to this as a *write reservation*. A write reservation privatizes read and write accesses to an address for the remainder of a transaction. Since

```
1  TM_BEGIN();
2  Node n = TM_READ(L.head);
3  while (TM_READ(n.next)) {
4    int val = TM_READ(n.next.val);
5    if (val == TM_READ(L.head.val)) {
6      Node next = TM_READ(n.next.next);
7      TM_WRITE(n.next, TM_READ(next));
8    } else {
9      n = TM_READ(n.next);
10   }
11 }
12 TM_END();
```

(a) Standard primitives

```
1  TM_BEGIN();
2  Node n = TM_READ(L.head);
3  TM_READ_RESERVE(L.head.val);
4  while (TM_READ(n.next)) {
5    int val = TM_READ(n.next.val);
6    if (val == L.head.val) {
7      Node next = TM_READ(n.next.next);
8      TM_WRITE(n.next, next);
9    } else {
10     n = TM_READ(n.next);
11   }
12 }
13 TM_END();
```

(b) ReserveTM primitives

Figure 1.2: Comparison of instrumentation primitives

the address is protected by a lock that can only be released by the writer, it is no longer considered to be shared memory since other transactions can neither read nor write to it. The runtime need only record the original value stored at an address so if the transaction should abort, the value could be restored. For reads, if the program acquires a shared reader lock, we can privatize future read accesses in the same manner: a *read reservation* privatizes read accesses for the remainder of a transaction. Since the reader lock protects the address from being modified, it is possible to continue reading the memory location without runtime overhead. Unlike with writes, this privatization can be shared with other readers. Eager versioning algorithms that do not acquire locks during a read, need to be extended to stall or abort writers during a read reservation.

We claim that this scheme of privatization is possible to apply safely without any domain knowledge. The stripping of runtime calls is possible with path-sensitive compiler alias-analysis which identifies whether a runtime call refers to the same memory address as a previous call. The locking of readers achieves privatization safety through *Pessimistic Concurrency Control* [26], Figure 1.2 demonstrates how transaction T_3 would be instrumented had programmer privatization not been used, and how our approach avoids the need for excessive runtime calls to access `L.head.x`.

Acquiring a lock before a read can incur a high overhead [20]. As a result, we only apply a reservation if it is clear that there are multiple runtime calls that can be omitted. In Figure 1.2 the value of the head node is needed on each loop iteration. Thus, the locking incurred by a single read reservation alleviates the need to have an instrumented read on each loop iteration. Another problem with using shared reader locks to support

```
1  atomic { // T_1
2    a = x;
3    b = y;
4  }
```

```
1  atomic { // T_2
2    x = x + 1;
3    y = y + 1;
4  }
```

Figure 1.3: Writer stall

privatization, is that it prevents writers from making forward progress on addresses that have been read by a transaction. This places a cost to concurrency control that may be detrimental to performance.

Memory reservations therefore allow efficient runtime-free access. However, there is a performance cost associated with this scheme, as we have chosen a reader-writer lock solution that limits concurrency between readers and writers. New readers or writers cannot operate on a write-reserved address, as the address is subject to change without communication to the runtime. Similarly, writers cannot write to a read reserved address until all readers have committed. Locking out writers is more of a concern as transactions tend to have larger reader sets. In addition, when operating on dynamically sized data structures, it is common for addresses to be read only once. For example a lookup in a tree would first read parent nodes. Herlihy et al. [13] observed a resulting problem for accesses to, for instance, the root node. It would be read, but not usually modified, in most tree operations. They proposed the utilization of *early releases* to remove an address from the read set of the transaction, enabling more concurrency and simplifying validation.

We therefore pair read reservations with a release: Owners may release addresses which are no longer being used. A transaction can only release an address in its read set, allowing writers to make forward progress and modify released addresses. It is also possible for a transaction to access a released address again for read or write. If the address has been locked for write by another reading transaction, that reader must abort.

An implication of releases is that a writer may stall during commit. If any released readers are still in-flight, the writer must wait for them to commit or abort. This is because it is possible that a reader attempts to read a value that has been written to by the writer. Consider Figure 1.3, if T_1 releases `x` early, and then T_2 modifies both `x` and `y`, T_1 could read an incorrectly modified value of `y`. This would cause an inconsistent snapshot, since `x` and `y` should both be incremented atomically.

Using reservations to privatize access to shared memory and releases to allow writers to make forward progress, we are able to improve the performance of eager versioned runtimes.

1.3 Limitations

The main benefit of reservations is the omission of unnecessary library calls to the runtime. This hinges on the capabilities of compiler alias-analysis to identify when shared memory accesses are to the same address. This is easier to do with intra-procedural analysis as opposed to inter-procedural. However there are a number of programming patterns where inter-procedural analysis is essential. In object oriented programming, access to internal object data is often wrapped by functions. If multiple object methods are called that access the same member data, inter-procedural analysis would be needed to add a reservation early on and remove the instrumentation from follow-on accesses. Intra-procedural alias analysis faces additional limitations when the functions are not in the same compilation unit, as it then requires whole-program analysis. Thus our approach hinges on the strength of a compiler's alias-analysis capabilities.

Library instrumentation points provide a means for an STM runtime to perform operations upon a running transaction. This includes, externally initiated actions such as remote aborts [21], metadata modifications or swapping out the STM algorithm [29]. By reducing the number of instrumentation points, the runtime has fewer opportunities to perform such actions, and as a result, actions may be delayed. We do not expect this to be a serious issue as it is unlikely that readers will generally make a release runtime call, and writers are not generally aborted.

The runtime is also unable to understand the full usage pattern of memory accesses. Wang et al. [29] choose the best runtime algorithm based on the access pattern of the workload. Omitting runtime calls complicates their analysis.

Our use of releases is flexible in that it allows addresses to be re-read after being released. To permit this, the runtime must be able to detect whether the value has changed since it was released. This can add a blocking operation to a commit sequence or require the storing of version numbers in the metadata. The impact of this approach maybe minimal, since the writer is guaranteed to commit, and is only waiting for reads to commit or abort. Thus, a runtime could choose to schedule other work instead of stalling the writer. Also, we argue that it is important for application developers to consider transactions as critical sections, in their code. As with most critical sections it is important to do the minimal amount of work necessary in order to facilitate a quick release of the critical section. The onus is on the programmer to be efficient and avoid repeated memory accesses.

A major limitation of our approach is that it is only applicable to eager versioned solutions. Lazy versioned runtimes do not expose information on how to directly access an address in the memory snapshot. There is an opportunity to explore an API that

exposes direct access to an address snapshot. However it is unclear whether this is possible or beneficial. We find that our approach allows better performance of eager versioned runtimes and improved scalability. Though lazy versioning may be important for certain workloads, our results show that using our instrumentation scheme, eager systems can enjoy performance that is comparable to lazy versioning. Eager systems are also more suitable for unmanaged languages such as C/C++. Since these languages are used to having direct access memory, reservations allow the runtime operations to be separate from memory operations. This allows the use of existing tooling such as debuggers.

By focusing our analysis on eager versioned runtimes and constraining the design space of STM runtime we find that we are able to provide performance benefits to STM operation.

1.4 Contributions

This thesis makes the following contributions:

- **Memory Reservations:** A privatization technique that can be applied without domain-specific knowledge, thereby reducing the instrumentation overhead required to support STM.
- **Memory Releases:** A means to remove items from the read set of a transaction, increasing opportunities for concurrency between readers and writers.
- **ReserveTM Byte-locks:** An implementation of byte-locks [6] that supports read-write lock primitives as well as reservations and releases. Byte-locks are used by the ByteEager algorithm provided by the RSTM [18].
- **Benchmark Evaluation:** Results of our approach on the STAMP [19] benchmark suite. Additionally, we provide a comparison of our approach to unmodified ByteEager and ByteLazy algorithms.

1.5 Organization

The remainder of this thesis is organized as follows:

Chapter 2 discusses related work in the area of STM. We describe how our work applies previously described techniques and how our approach differs.

Chapter 3 describes our API in detail and provides examples on how it can be used. We further discuss the optimizations that we were able to apply that extend from our work. We describe the ByteLock algorithm and how we extended it to support reservations and releases.

Chapter 4 shows our results with the STAMP benchmark suite and discusses in detail how our approach changed the behavior of each of the benchmark tests. If any modifications were made to a benchmark they are described with a clear motivation.

Chapter 5 discusses future work and how we plan to apply our approach more thoroughly to a compiler as well as to more eager versioned algorithms.

Chapter 2

Related Work

The work presented in this thesis extends many of the previous STM concepts, in an effort to describe the primitives needed to build an efficient and scalable STM runtime that does not require domain-specific knowledge.

Herlihy and Moss [14] first introduced Transactional Memory as a means for hardware to provide non-blocking synchronization. Their work described transactions as a concurrency primitive for programmers to build lock-free data structures. Transactions ensure serializability and atomicity. Serializability allows transactions to be ordered such that they don't appear to interleave with one another, whereas atomicity ensures that either the transaction completes in its entirety or fails to execute at all.

Shavit and Touitou [23] developed Software Transactional Memory, which enabled the transactional semantics to be simulated by software. Their implementation required a transaction to declare, at the start of a transaction, a vector of addresses that it may access. Despite this constraint, STM could be utilized right away as it utilized existing concurrency primitives to build transactions. Following work improved the efficiency of STM, defined program workloads and designed conflict resolution mechanisms. DSTM [13] made the key contribution of the first *Dynamic Software Transactional Memory* system, whereby a transaction need not declare its working set in advance. Since then, the community has expanded to make transactional memory (both hardware and software) more viable for programmers. Currently, both Intel [2] and IBM [11] support Hardware Transactional Memory and STM solutions exist for GCC [22], LLVM [8] and the Intel Compiler [1].

2.1 Privatization and Reservations

Shavit and Touitou [23] described the most basic form of a reservation. A transaction must declare at the start, all addresses that it might access. The runtime would then generate a serializable snapshot of memory for a transaction to execute with. As a result, the system was inherently lazy versioned as transactions needed to access memory through the generated snapshot. Generating a vector of addresses for a transaction requires domain-specific knowledge. Generally a transaction does not know, in advance, all the addresses which it would need. Thus, a transaction pessimistically reserved all the addresses it could possibly access, rather than only those that it needed. When accessing data structures, a transaction reserves the entirety of a data structure that it could access. This not only added excessive overhead, but is simply impossible for many dynamically sized data structures such as linked lists and trees. Dynamic data structures require traversals to obtain the addresses of all the allocated nodes. In addition, compilers cannot necessarily build a vector of memory addresses simply from static analysis (except for trivial examples). Our approach avoids these limitations, as it dynamically applies read and write reservations, and operates with an eager versioned runtime.

Privatization originates from lock-based concurrency control, where programmers try to limit the amount of work done while holding a lock. Code attempts to do as much work as possible outside of the lock by moving heavy processing out of the critical section. In Figure 1.1a, T_1 efficiently removed all elements from the list within the transaction, and processed the list outside of the transaction. Spear et al. [26] noted that privatization imposes correctness issues for previously proposed STM algorithms and stated that many algorithms are not privatization-safe, which posed a problem for programmers expecting STM to be a drop-in replacement for locks. They provided an exhaustive investigation of techniques for handling privatization and proceeding work provided alternatives [17][24]. Our work applies many of the techniques presented, but uniquely uses different techniques, depending on whether an address has been reserved or released. When an address has been reserved, the runtime can utilize Pessimistic Concurrency Control, while releases are supported with a Transactional Fence [26].

2.2 Releases

Early releases were first introduced by DSTM [13] to support dynamically sized data structures. When a transaction released a memory address, it indicated that other transactions

may freely access the address without conflicting with the current transaction. It required manually inserted releases with domain-specific knowledge to determine if a release would be "safe". A transaction performing a lookup in a binary search tree, could release parent nodes during its traversal since lookups do not backtrack on previously accessed nodes. DSTM does not require commit-time validation of addresses that have been released, thereby avoiding blocking and reducing commit overhead. Though early releases increase concurrency, manually instrumented releases are error-prone, as the conditions supporting a release may change. In our approach, we are able to apply early releases to any read and enable significant concurrency between readers and writers. Writers need to perform a blocking commit-time validation for any released addresses that it read. We also allow an address to be re-read after being released.

ReserveTM releases differ fundamentally from early releases as it does not require a balancing of reserve and release calls. DSTM proposed that for each address access, there should be a corresponding release. Each access would increment a reference count which would be decremented by a release. Reference counting adds CPU bus traffic and adding release calls results in additional runtime overhead. ReserveTM releases do not require reference counts to be maintained and only requires a single release call to release an address.

TinySTM [9] supports releases on reads by never holding any locks while doing a read. Thus, a reader never blocks a writer from making forward progress. Its policy favors writers, as readers cannot safely privatize an address for instrumentation-free access. Since transactions have larger read sets than the write sets, reducing instrumentation for reads is essential for performance. Our approach recognizes this behavior and allows read accesses to be instrumentation-free while ensuring privatization-safe access.

Chapter 3

ReserveTM

We recognize the importance of privatization and early releases in Section 2, and introduce a new set of runtime primitives to instrument transactional memory accesses. ReserveTM primitives enrich the communication between a transaction and the runtime in order to improve application performance. The runtime is explicitly aware of when privatization occurs and can adjust accordingly. These primitives come at the cost of restricting the design space of STM runtimes, as we only support eager versioning. However, there is still sufficient room to innovate and design new runtime algorithms, while observing the benefits of adopting ReserveTM primitives.

We continue by presenting the running example, shown in Figure 3.1. Our example extends the C language by adding an `atomic` block that denotes the start and end of a transaction. Functions `foo` and `bar` operate on both global and thread-local addresses and can be called from within or outside of a transaction. Either a programmer or a compiler must transform the code within an atomic block to utilize STM primitives for shared memory accesses.

The following sections describe the current STM primitives used today and then demonstrate how they would be applied to our example. We then present and apply the ReserveTM primitives and discuss how it improves the instrumentation and enables unique optimizations. Finally, we adapt an eager versioned STM algorithm to support ReserveTM primitives.

```
1  int a, b, c, d;
2
3  int bar(int& x) {
4    return x+1;
5  }
6
7  int foo() {
8    a = 2;
9    b = 2;
10   if (d >= 0) {
11     b = c + d;
12   } else {
13     a = bar(b);
14   }
15   return a + b;
16 }
17
18 int main() {
19   int i, j, k;
20
21   atomic {
22     i = foo();
23   }
24
25   atomic {
26     int l = 2;
27     j = bar(l);
28   }
29
30   k = bar(j);
31 }
```

Figure 3.1: Motivating example using with language-support for transactions

3.1 STM Primitives

An STM runtime provides a mechanism to denote the start and end of a transaction, as well as memory access operations. A runtime may also support self-abort, whereby a thread can explicitly abort the current transaction. We describe the API as follows:

- `TM_BEGIN()`: Signals to the runtime that the current thread wishes to start a transaction. A runtime may use this opportunity to pre-allocate any data structures needed during transaction execution, notify other threads of the new transaction and apply any scheduling policies such as disallowing thread preemption.

  If a transaction were to abort, it would return to this point for re-execution. A architecture-specific operation (such as `setjmp/longjmp` on x86) restores the program state including the program counter, registers and stack pointer.

- `TM_END()`: Instructs a runtime to commit the actions performed by a transaction and make them visible to other transactions. If a conflict is detected, the transaction will abort and return the thread to the point at which `TM_BEGIN` was called. Otherwise, the thread continues execution outside of a transactional scope.[1]

  Lazy versioning systems will attempt to resolve conflicts and apply their buffered updates upon commit. Eager versioning usually has a more lightweight commit routine, as updates have already been made and most of the conflicts would have already been resolved. If a transaction performed any speculative operations, such as reading a write of an in-flight transaction, a commit might stall waiting for the writer to commit. Additionally, a transaction may need to validate its read set and abort if it has changed.

- `TM_ABORT()`: Invokes an explicit abort to the runtime. STM allow transactions to issue a *self-abort*. The primary motivation is to allow a programmer to easily rollback any modifications made by a transaction and return to the start. A lazy versioned runtime would simply discard any updates made, whereas an eager runtime would need to apply its undo log to restore modified values to their original state.

[1] We stated that after a transaction commits the thread is not in a transactional scope. This is not necessarily true for *nested transactions* whereby a transaction can be started from within another transaction. Nested transactions allow a runtime to do nested aborts whereby the thread only returns to the start of the sub-transaction on abort.

A runtime that does not have explicit support for nested transactions can increment a nesting level on each `TM_BEGIN()` and decrement it on each `TM_COMMIT()` such that only a commit on level 0 results in the commit logic being applied. Correspondingly, an abort in a nested transactions would abort all nested transaction levels and return to the outermost `TM_BEGIN`.

- `TM_READ(addr)`: Returns the value at `addr` in the current transaction's memory snapshot. Any address that could be accessed by another concurrently running transaction needs to be protected by an access through the runtime. A runtime must return a value that is consistent with the snapshot of memory which it is presenting to the currently running transaction.

  Lazy versioning systems may build the snapshot on demand and copy the current value into the snapshot state before returning the value. Subsequent reads of `addr` would be loaded from the snapshot. If the running transaction has written to `addr`, the runtime must return the value that was written. Eager versioning systems need to perform some metadata checks before returning the value stored at address `addr`.

  The behavior in the presence of writes shows a stark contrast between lazy and eager versioned systems: lazy versioning requires use of the runtime to ensure that the correct value is loaded from the snapshot, while eager versioning systems can access the value in-place.

- `TM_WRITE(addr, val)`: Writes the value `val`, to address `addr`. For lazy versioned systems, if the value was previously read, the runtime updates the value in its snapshot, else it needs to record a new address in its snapshot. Eager systems may need to resolve conflicts before a write if the address has not previously been written to. A write may cause a metadata lock to be acquired, which may stall thread execution, or abort the transaction in the failure case.

Figure 3.2 shows how the STM primitives are applied to the running example.

The function `bar` is duplicated to `__TX_bar`. The duplicate function is needed because `bar` could be called from a non-transactional scope and to avoid the overhead of calling into the runtime if it is not necessary. A runtime may support performing a read and write outside of a transaction, but in a weak isolation model it incurs unnecessary overhead. Avoiding function duplication is possible if a function, as well as functions that it could call, do not require STM instrumentation, or always requires instrumentation as with `foo`. Side-effect free functions do not require instrumentation and thus need not be duplicated.

`TM_BEGIN` and `TM_END` calls are placed at the boundaries of atomic blocks and any functions called within the block are replaced with transactional versions. The `TM_READ` and `TM_WRITE` calls replace accesses to shared memory in transactional versions of functions. In our example, `__TX_bar` accesses parameter `x`, which could be a global or a thread-local variable, when called from `__TX_foo` or `main` respectively. We must apply instrumentation conservatively on `__TX_bar` and always instrument the read to `x`. As a result, access to `l`

```
1  int a, b, c, d;
2
3  int bar(int& x) {
4    return x+1;
5  }
6
7  int __TX_bar(int& x) {
8    return TM_READ(x)+1;
9  }
10
11 int foo() {
12   TM_WRITE(a, 2);
13   TM_WRITE(b, 2);
14   if (TM_READ(d) >= 0) {
15     TM_WRITE(b, TM_READ(c) + TM_READ(d));
16   } else {
17     TM_WRITE(a, __TX_bar(b));
18   }
19   return TM_READ(a) + TM_READ(b);
20 }
21
22 int main() {
23   int i, j, k;
24
25   {
26     TM_BEGIN();
27     i = foo();
28     TM_END();
29   }
30
31   {
32     TM_BEGIN();
33     int l = 2;
34     j = __TX_bar(l);
35     TM_END();
36   }
37
38   k = bar(j);
39 }
```

Figure 3.2: Motivating example instrumented with the current STM primitives

will be mediated by the runtime, even though it is stack variable that will not be accessed by other transactions. If a private memory address is instrumented then the program has been *over-instrumented*. Compilers often suffer from over-instrumentation due to difficulty in detecting which addresses are shared.

Runtime instrumentation restricts the compiler from performing certain optimizations. In `__TX_foo` the global `a` is written to twice if `d` is less than 0. In the non-transactional version of the code, a compiler could have re-ordered the logic to avoid the duplicate writes by moving the test of `d` to be earlier. Similarly, when `d` is greater than or equal to 0, there is a second instrumented read to `d`. Typically, a compiler could avoid the second read from memory by loading `d` into a local register. The current STM primitives impedes such code transforms. Read and write calls are opaque function calls and the compiler cannot perform optimizations across them without knowledge of the side-effects.

3.2 ReserveTM Primitives

ReserveTM primitives extend the the existing API by clearly defining when an address is privatized and can be freely accessed without runtime instrumentation. In addition, it enables extensive use of releases in order to enable more concurrency between readers and writers. ReserveTM primitives are performed out-of-band, *i.e.* they do not directly perform a read or write to memory. As a result, calls can be re-ordered to allow for compiler optimizations.

The `TM_BEGIN()`, `TM_END()` and `TM_ABORT()` primitives remain unchanged. ReserveTM primarily concerns itself with handling conflict detection for memory accesses. ReserveTM defines the following calls:

- `TM_READ_RESERVE(addr)`: Indicates that `addr` is being privatized by the current transaction for read-only access. All subsequent reads to `addr` may be made without runtime instrumentation. The runtime guarantees that it will not allow any transaction to modify the address for the duration of the read reservation. Note that there is no means for a runtime to enforce this constraint, it is simply a contract between the transaction and the runtime, where the transaction will only perform read operations.

  This function returns a boolean value indicating whether the runtime supports releases and if the address can be released. If the address was already in the reserved read set of the current transaction, the call will return false. Only if the address

was never read before or had already been released does `TM_READ_RESERVE` return true. This allows us to avoid DSTM's [13] reference counting mechanism, as only when `TM_READ_RESERVE` returns true, should a thread attempt to release an address. Thus in the presence of nested reservation calls, a release is only paired with the first reservation call.

- `TM_WRITE_RESERVE(addr, val)`: Requests to privatize `addr` for read and write access. If successful, the address is exclusively privatized by the current transaction and subsequent reads and writes do not require instrumentation. If the reservation cannot be fulfilled, the runtime will abort the transaction. Performing a write reservation causes the runtime to record the old value for roll back operations and acquire any metadata locks to maintain exclusive access.

- `TM_READ(addr)`: Privatizes `addr` for read-only access and returns the value at `addr`. ReserveTM maintains compatibility with the standard STM primitives by making `TM_READ` the same as a `TM_READ_RESERVE` followed by a direct read from the address.

- `TM_WRITE(addr, val)`: Privatizes `addr` for write access and writes the value `val` at address `addr`. Similar to `TM_READ`, the write API implies a `TM_WRITE_RESERVE` followed by a direct write.

- `TM_RELEASE(addr)`: If `addr` has not been write reserved, releases `addr` to allow a writer to acquire it for write access. A release is only applicable for addresses in the read set of a transaction. For addresses that have been written to, this call is ignored.

  The call terminates a read reservation, such that an address can no longer be transparently accessed. If the transaction wishes to read to the address again it can do so after issuing a new read reservation. If it has been write reserved by another transaction, the value at the address may have changed since when it was first read. As a result, the transaction will need to abort if the address has been write reserved.

  Note that a `TM_RELEASE` should only be called if a previous call to `TM_READ_RESERVE` returned true. This avoids releasing an address earlier than expected. Once again, the runtime cannot enforce this constraint, as it is only a contract.

- `TM_READ_RELEASE(addr)`: Mimics a `TM_READ_RESERVE` followed by a direct access to the address and then a `TM_RELEASE`. Note that if `addr` was already in the read set, no release is performed. TinySTM implements reads in a similar manner.

We demonstrate the use of ReserveTM primitives in Figure 3.3a and 3.3b, with the latter including optimizations. ReserveTM removes the need to duplicate the `bar` function, as it

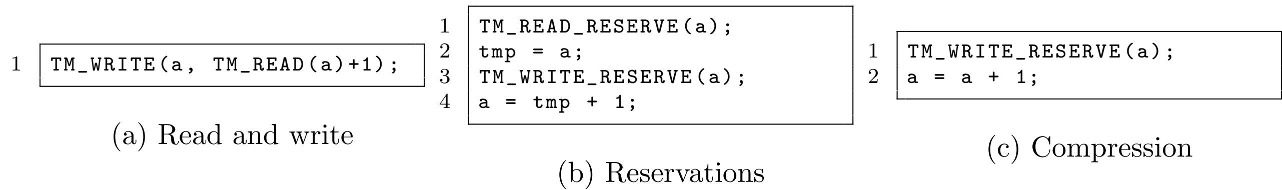
```
1  int a, b, c, d;
2
3  int bar(int& x) {
4    return x+1;
5  }
6
7  int foo() {
8    TM_WRITE_RESERVE(&a);
9    a = 2;
10   TM_WRITE_RESERVE(&b);
11   b = 2;
12   bool release = TM_READ_RESERVE(&d);
13   if (d > 0) {
14     b = TM_READ_RELEASE(&c); + d;
15     if (release)
16       TM_RELEASE(d);
17   } else {
18     if (release)
19       TM_RELEASE(d);
20     a = bar(b);
21   }
22   return a + b;
23 }
24
25 int main() {
26   int i, j, k;
27
28   {
29     TM_BEGIN();
30     i = foo();
31     TM_END();
32   }
33
34   {
35     TM_BEGIN();
36     int l = 2;
37     j = bar(l);
38     TM_END();
39   }
40
41   k = bar(j);
42 }
```

(a) Naive use of ReserveTM

```
1  int a, b, c, d;
2
3  int bar(int& x) {
4    return x+1;
5  }
6
7  int foo() {
8    TM_WRITE_RESERVE(&a);
9    TM_WRITE_RESERVE(&b);
10   bool release = TM_READ_RESERVE(&d);
11   if (d > 0) {
12     a = 2;
13     b = TM_READ_RELEASE(&c); + d;
14     if (release)
15       TM_RELEASE(d);
16   } else {
17     b = 2;
18     if (release)
19       TM_RELEASE(d);
20     a = bar(b);
21   }
22   return a + b;
23 }
24
25 int main() {
26   int i, j, k;
27
28   {
29     TM_BEGIN();
30     i = foo();
31     TM_END();
32   }
33
34   {
35     TM_BEGIN();
36     int l = 2;
37     j = bar(l);
38     TM_END();
39   }
40
41   k = bar(j);
42 }
```

(b) Optimized

Figure 3.3: Motivating example instrumented with ReserveTM primitives



```
1  TM_WRITE(a, TM_READ(a)+1);
```

(a) Read and write

```
1  TM_READ_RESERVE(a);
2  tmp = a;
3  TM_WRITE_RESERVE(a);
4  a = tmp + 1;
```

(b) Reservations

```
1  TM_WRITE_RESERVE(a);
2  a = a + 1;
```

(c) Compression

Figure 3.4: Reducing instrumentation with compression

no longer requires instrumentation. Though `bar` performs a read of `x`, it does not need to perform a read reservation when called from `main`, since `l` is already privatized from being a local stack variable. Also, when `foo` calls `bar` with `b`, it has already been write reserved earlier on.

As reservations are performed out-of-band, and the side-effects are known (*i.e.* it will not directly modify any shared memory), they can be moved to the beginning of the function `foo`. This enables optimizations of writes to `a` and `b`, such as to avoid unused writes.

Releases are applied wherever possible. Before reading `d`, a read reservation is requested and accordingly the release is conditionally called based on the value of the `release` variable. Since `c` is accessed only once we instrument it with a `TM_READ_RELEASE`. We do not instrument releases to `a` or `b` since they are both reserved for write.

Code instrumented with ReserveTM primitives is cleaner, optimized and incurs less runtime overhead. It also can avoid the need to duplicate functions under the correct conditions.

3.3 Further optimizations

ReserveTM primitives reduce the instrumentation required to support STM by explicitly defining when an address is privatized. Using these primitives, we describe how compiler static analysis could further reduce STM instrumentation.

3.3.1 Compression

A common pattern is that a value is read and then updated, as with the increment operation in Figure 3.4a. If we naively apply reservations, the `TM_READ` and `TM_WRITE` would

be replaced by a read and write reservation as illustrated in Figure 3.4c. Incurring the overhead of 2 reservation calls can be avoided by performing a single write reservation. If a read reservation is guaranteed to be followed by a write reservation to the same address, we replace the initial read reservation with a write reservation. Figure 3.4c demonstrates compression applied to the increment operation. A compiler can apply a path-sensitive alias-analysis to identify such opportunities. Similarly, we omit any read or write reservations that follow a write reservation since the write reservation has already privatized the address.

3.3.2 Reserve Dependencies

Reservations allow a transaction to avoid having to instrument each memory access, with the additional benefit where they reduce opportunities for conflict between transactions. If a transaction has reserved all of its working set, then it is guaranteed to commit (unless the programmer issues a self-abort). With the use of whole-program alias analysis we identify when a transaction cannot be aborted and eliminate any subsequent reservations.

We define a *reservation dependency* between addresses A and B, when a write reservation to A always precedes a read or write reservation to B for all program paths. Since only a single transaction can attain the reservation to A, B can be reserved if and only if the write reservation to A succeeds. Furthermore, if the last reservation of a transaction cannot have any conflicts, an eager versioned STM runtime can eliminate the reservation and directly access the associated address. Thus, we omit the reservation of B if it is the last reservation of a transaction. Reservation dependencies differ from compression as A and B can be different addresses. In addition, the dependency must hold for all program paths, thus requiring whole-program path-sensitive analysis.

We use Figure 3.5 as an example of how reservation dependencies can be applied to reduce instrumentation. The example describes a thread pool class which uses transactions to enable concurrent access to the internal pool. Clients are assigned threads from the thread pool by invoking `getThread` and return threads with the `returnThread` method. The thread pool maintains counters for the number of allocated threads (`threads`), idle threads (`idle`), and in-use threads (`active`). The `destroyThreads` method is used deallocate threads from the thread pool. It prevents additional `getThread` calls and waits till all the threads have been returned to the thread pool before destroying the threads.

We observe that a write to `threads` is always preceded by a write to `idle` in either `getThread` or `returnThread`. Thus, there is a reserve dependency between `idle` and `threads`. The write to `threads` does not need to be instrumented as any conflicts

```
class ThreadPool {
  void addThread(Thread* t) {
    atomic {
      // Add t to thread pool
      idle = idle + 1;
      threads = threads + 1;
    }
  }

  Thread* getThread() {
    Thread *t;
    atomic {
      if (drain || (idle == 0)) {
        return 0;
      }

      // get thread t from pool
      --idle;
      ++active;
    }

    return t;
  }

  void returnThread(Thread *t) {
    atomic {
      ++idle;
      --active;
    }
  }

  void destroyThreads() {
    bool spin = false;
    atomic {
      drain = true;
      if (active) {
        spin = true;
      }
    }

    while (spin) {
      sleep(1); // Sleep outside of transaction
      atomic {
        if (active == 0) {
          spin = false;
        }
      }
    }

    // destroy threads
  }

  // Further implementation
};
```

Figure 3.5: Thread pool code excerpt

would have been resolved during the write to `idle`. Conversely, `active` is preceded by writes to `idle` in `getThread` and `returnThread`, but not in `destroyThreads`. Therefore, `active` cannot have a reservation dependency with `idle` as a successful write reservation of `idle` does not guarantee conflict-free access to `active`. If a transaction is executing `destroyThreads`, it may conflict with another transaction running `getThread` or `returnThread`.

Reservation dependency analysis can remove both read and write reservations. Removing a write reservation has the additional benefit of avoiding undo log tracking, since rollback cannot occur. With reserve dependency analysis, we are able to significantly reduce instrumentation in the kmeans benchmark as described in Section 4.2.1.

3.4 ByteEager

ByteEager is an STM algorithm provided by RSTM [25], similar to TLRW [6]. In this section we describe how ByteEager operates and our extensions to support ReserveTM primitives.

ByteEager adopts eager versioning. Writes are performed in-place while maintaining an undo log for rollback in the event of an abort. Prior to an in-place read or write, ByteEager acquires a metadata lock. ByteEager utilizes byte-locks [6] which have read-write lock semantics. A write must acquire an exclusive lock to prevent other readers or writers from accessing an address. Readers acquire a shared reader lock, allowing multiple readers to concurrently read the same address. Each memory address hashes to a byte-lock from a fixed size pool. A hash collision causes a byte-lock to protect access to multiple addresses.

Byte-locks offer low read-lock acquisition overhead by avoiding an expensive compare-and-swap (CAS) operation in favor of atomic byte operations. We extend byte-locks to support the ReserveTM primitives described in Section 3.2, while maintaining backward compatibility with read-write byte-locks.

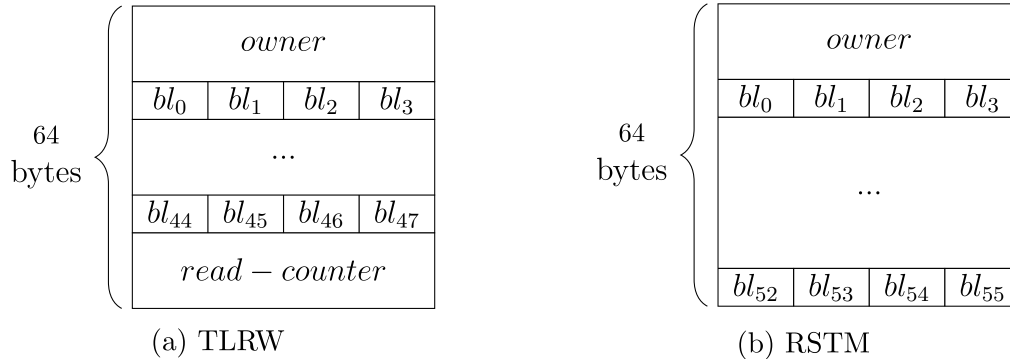


Figure 3.6: Byte-lock layout

3.4.1 Read-Write Byte-locks

Figure 3.6a illustrates the layout of byte-locks as original proposed in TLRW [6]. Byte-locks consist of 3 zones:

- **Owner field:** Signals whether a lock is held (or is in the process of being acquired) by a writer. If the byte-lock is locked, the owner field contains the thread id of the lock owner. A value of 0 indicates that the lock is free.
- **Byte-array:** Indicates *slotted* threads that hold a shared reader lock. Each slotted thread owns an element in the fixed size byte-array which indicates whether the lock is held in read mode.
- **Read-counter:** A counter for *unslotted* threads that hold the lock as a shared reader. Each unslotted thread increments the read-counter for read-lock acquisition and decrements to free the lock.

The byte-lock structure is constrained to a 64 byte cache line to minimize memory overhead and CPU bus traffic. Constraining the size limits the number of slotted threads with an associated element in the byte-array. TLRW byte-locks use atomic byte operations on byte-array elements to perform efficient read-lock acquisition for slotted threads. Unslotted threads increment and decrement the read-counter with an expensive CAS. Additionally, the read-counter is shared between unslotted threads, causing higher contention than experienced by slotted threads. Thus, slotted threads have less read-lock overhead than unslotted threads. RSTM's implementation of byte-locks maintain consistent performance of read-lock acquisition by removing support for unslotted threads and omitting the read-counter as observed in Figure 3.6b.

A `TM_WRITE` call acquires a byte-lock in exclusive mode by storing the thread id of the current transaction in the owner field. Owner acquisition uses a CAS operation to set the owner field from 0 to the thread id. If the lock is held, the CAS will fail as the owner field is not 0. Thus, a writer can only acquire a lock that is free. A writer may restart a failed acquisition for a number of times before aborting a transaction.

After owner acquisition succeeds, the writer must query all the flags in the byte-array to ensure that there are no active readers. A writer may stall for a specified interval, in order for readers to "drain out". A reader is drained when it frees the lock due to a commit or abort. Once there are no more readers, the writer has exclusive access to the memory address protected by the byte-lock. Subsequent reads and writes by a writer need only look at the owner field to verify ownership before performing a read or write.

A reader acquires a lock in shared mode by setting the current thread's flag in the byte-array and then querying the owner field for write ownership. If the lock is currently held, a reader must clear its byte-array flag and either abort the current transaction or spin before re-trying. A successful acquisition guarantees the address protected by the lock will not be modified until the read lock has been freed. When a transaction tries to re-read a previously read address, it first checks the owner field to determine if it has exclusive access before querying the byte-array.

Upon commit, a transaction's read locks are freed by clearing the respective byte-array element and write locks are freed by setting the owner field to 0. Thus, all the lock operations are non-blocking[2].

3.4.2 ReserveTM Byte-locks

To support ReserveTM primitives we extend byte-locks as illustrated in Figure 3.7. Specifically, we use 2 cache lines, add a *released* field as well as a *release* flag to byte-array elements.

TLRW and RSTM constrain a byte-lock to a single cache line. Our evaluation system has 64 hardware threads, which cannot be supported by only slotted threads. To retain the consistent performance of RSTM's implementation, we also do not support unslotted threads. We allow byte-locks to spill across 2 cache lines and dedicate a cache line to the byte-array. Larger byte-locks only incur a performance degradation [6] in artificial, random

[2] Lock operations may choose to spin and retry. However, this would be for a fixed interval and is only applied as a performance improvement for pathological races.

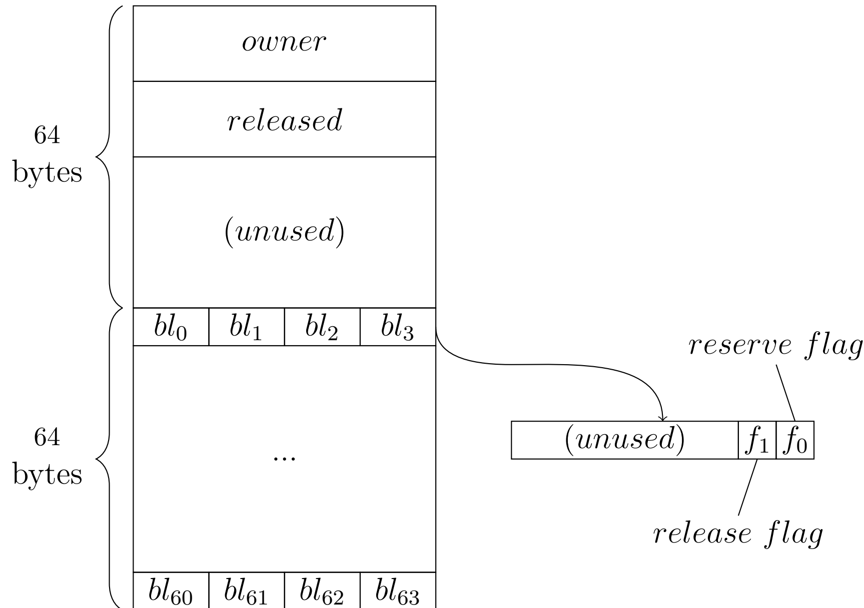


Figure 3.7: ReserveTM byte-locks

access benchmarks. Opting to use 2 cache lines was necessary for our setup and did not have a measurable impact on our evaluation benchmark.

A reader sets the reserve flag during a read reservation and sets the release flag when performing a release. The owner of the write lock uses the released field to indicate whether there were any readers that have released the lock but have yet to commit. The writer checks the release flags in the byte-array to determine if the lock has been released by a reader. If the released field is set, the writer must stall at commit until all readers have freed the lock.

To execute a `TM_READ_RESERVE` or `TM_WRITE_RESERVE` the lock acquisition logic remains largely the same as with `TM_READ` and `TM_WRITE`, except memory access is not performed by the runtime. The acquisition of a writer lock ensures that no other transaction can read or write to an address. Thus after a successful write reservation, a thread can freely modify the address without additional instrumentation. Similarly upon acquiring a read lock for a read access, the address is guaranteed to not change and can be read without instrumentation. Removing instrumentation from subsequent accesses improves the use of the CPU cache. We avoid the need to reload the byte-lock cache lines to verify that the state of the owner field or byte-array element. Avoiding the byte-array check is essential for readers, as the cache line could be invalidated by other readers that acquire or release

the lock.

To release an address, a thread first checks the owner field of the associated byte-lock. If it is set to the current thread id, then the release is ignored: writes cannot be released. Otherwise, the byte-array byte is atomically modified to clear the reserve flag and set the release flag. Both the reserve and release flags must be cleared on commit or abort. A repeat read to a released memory address must atomically set the read bit while maintaining the release bit. If the owner field is set to the thread id of another thread, this implies that a writer has acquired the released lock and the reader must abort. Since writers may modify the value in an address, the reader must abort to avoid reading a value inconsistent with its initial read.

DSTM [13] used reference counting between read reservations and releases, in order to detect the final release of an address. Updating a reference count makes reads and releases more expensive due to cache line invalidation. ReserveTM byte-locks avoid writes on repeat reads by returning `true` on a `TM_READ_RESERVE` if the read flag was previously unset, and `false` otherwise. A program should only perform a release if the read reservation returns `true`.

Read-write byte-locks do not allow a writer to acquire a lock while there are in-flight readers. Releases enable a `TM_WRITE_RESERVE` call to acquire a lock on a released address. Writers may modify memory locations without waiting for readers to drain, thereby increasing concurrency between transactions. To support acquisition of released locks, the writer continues to acquire the lock by setting the owner field and checking the reserve flags as with read-write byte-locks. If there are readers that are concurrently trying to reacquire a released lock, they will abort upon observing the owner has been set. Thus, no readers can reacquire the read lock after write-lock acquisition. Upon successful write-lock acquisition, the writer must check the release flags in the byte-array to identify any active released readers. If the lock has been released, the writer must set the reserved field in order to perform commit-time validation.

The introduction of a blocking operation at commit is a key difference between read-write byte-locks and ReserveTM byte-locks. During a commit, the writer must check that all released readers have been drained. The writer must block until all the released readers have committed or aborted. This ensures serializability of transactions, since readers could not have executed before the writer. In addition, the order in which read and write locks are freed is critical. Since a writer blocks during commit, it is possible to have a circular dependency whereby a writer is blocked waiting for a thread that is also blocked. If the second thread is waiting for the first writer to free a released lock, they will deadlock. Thus a writer must first free all of its read locks before blocking. This circular dependency

cannot be resolved without at least one of the transactions aborting.

Chapter 4

Experimental Evaluation

We evaluated our approach with a manually-instrumented version of the STAMP [19] benchmark suite provided with RSTM [25]. Note that the bayes++ benchmark did not run under our setup. We use the modified byte-locks [6], as described in the previous section, and adapt the benchmarks to use the ReserveTM primitives. For each benchmark, we compare our results against using the standard API with the ByteEager and ByteLazy algorithm, which use byte-locks in an eager or lazy versioned fashion respectively. Any pair of reservations and releases that we added represent a pessimistic view of what a compiler could do; *i.e.* we only apply reservations and releases with intra-procedural analysis and do not cross function boundaries. Furthermore, we favor the use of the `TM_READ_RELEASE` call over `TM_READ`, thus optimistically releasing reads to improve concurrency.

We run the benchmarks on a 4-way AMD Opteron 6100 with a total of 64 hardware threads and 128GB of RAM. To increase predictability, we execute one software thread per hardware thread, distribute threads evenly across all NUMA nodes, and disable thread migration. Disabling thread migration enables cache line traffic to be more consistent since a thread's working set does not need to be migrated. Previous evaluations report results which utilized more software threads than supported by the processor. This causes the thread scheduler to affect the performance of the benchmark, as it is possible for a thread to be scheduled out during the execution of a transaction, increasing unpredictability as follows. If an in-flight transaction has a working set which conflicts with another transaction, a thread switch may result in a longer transaction time, resulting in more conflicts than expected. Conversely, if a transaction is stalling during commit-time validation, allowing another thread to execute may improve performance as it allows useful work to be done. Understanding the role of the thread scheduler in the performance of an STM runtime is beyond the scope of this work and requires a dedicated analysis. Our methodology

```
1 if (TMLIST_ITER_HASNEXT(&it, chainPtr)) {
2   pair_t* pairPtr = TMLIST_ITER_NEXT(&it, chainPtr);
3   dataPtr = pairPtr->secondPtr;
4   break;
5 }
```

Figure 4.1: Original iterator access in STAMP

```
1 pair_t* pairPtr = 0;
2 if ((pairPtr = TMLIST_ITER_NEXT(&it, chainPtr))) {
3   dataPtr = pairPtr->secondPtr;
4   break;
5 }
```

Figure 4.2: Modified iterator access in STAMP

allows us to stably measure and compare our approach to the standard API. We expect an optimized thread scheduler to further improve upon our results.

4.1 Benchmark Optimizations

Manual instrumentation of STAMP results in very little over-instrumentation. RSTM applies domain-specific knowledge to only instrument access to memory that could be shared between transactions. However, we developed a LLVM [15] compiler pass to add profiling to the runtime instrumentation functions, looking for optimization opportunities. In particular, we used dynamic profiling to look for patterns where a benchmark was re-reading a released address and examined whether it could be optimized with a reserve and release. Our investigation was able to identify a common pattern, as shown in Figure 4.1.

Profiling showed that the definition of `TMLIST_ITER_NEXT` loads `nextPtr`, a member of the `it` iterator object. When `it` is read, our analysis identified it as a previously released address. This is because the check in `TMLIST_ITER_HASNEXT` will always read `nextPtr` first (to check that it is not-null). Rather than utilizing a reserve and release to reduce the instrumentation overhead, we modified the code as shown in Figure 4.2. The modified implementation of `TMLIST_ITER_NEXT` only accesses the data member once and returns null if `nextPtr` is null. Note that `TMLIST_ITER_HASNEXT` was used in many other places and was not always followed by a call to `TMLIST_ITER_NEXT` call. This pattern was seen with more complex data structures as well. Such data structures also introduced unnecessary duplicate lookups. This is discussed at greater length in the benchmark evaluation.

```
1 TM_BEGIN();
2 long val = TM_READ(*new_centers_len[index]) + 1;
3 TM_WRITE(*new_centers_len[index], val);
4 for (j = 0; j < nfeatures; j++) {
5   val = TM_READ(new_centers[index][j]) + feature[i][j];
6   TM_WRITE(new_centers[index][j], val);
7 }
8 TM_END();
```

Figure 4.3: kmeans transaction

4.2 Evaluation

For each benchmark we calculate the speedup over sequential performance, provided by three algorithms:

- **ReserveTM:** ReserveTM byte-locks with ReserveTM primitives.
- **ByteEager:** Read-write byte-locks with eager versioning and traditional STM primitives.
- **ByteLazy:** Read-write byte-locks with lazy versioning and traditional STM primitives.

Speedup is calculated by taking the ratio of the time to execute an STM instrumented workload versus a non-instrumented sequential workload. In addition, we compare the performance of ReserveTM and ByteLazy to ByteEager to demonstrate any improvements that ReserveTM achieves and how lazy versioning compares to eager versioning.

4.2.1 kmeans

The kmeans benchmark applies a k-means clustering algorithm to group n samples into k clusters. The transactional version of this algorithm uses 3 transactions. The main work is done in a single transaction, shown in Figure 4.3, which increments the length of the cluster centers and then sums all the objects within it.

The remaining 2 transactions increment 2 separate globals. As a result, those transactions are small and the contention is largely determined by the earlier described transaction. The kmeans benchmark has both high and low contention configurations which adjust the value of k, the number of clusters. More clusters would result in fewer conflicts since transactions are less likely to be processing the same cluster center.

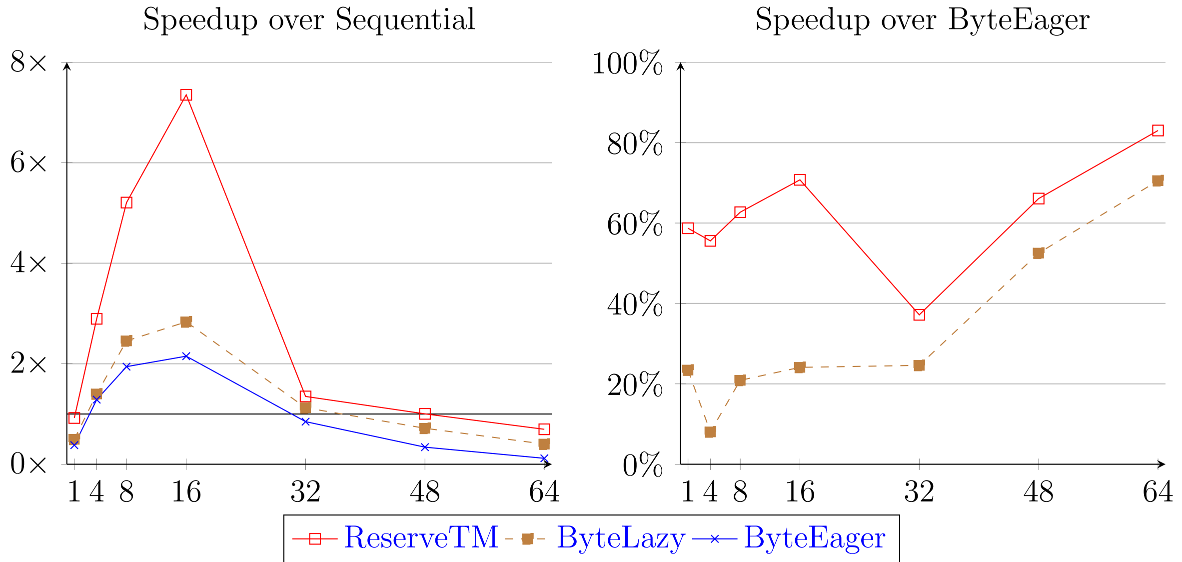


Figure 4.4: Speedup of STM threads for kmeans-low++

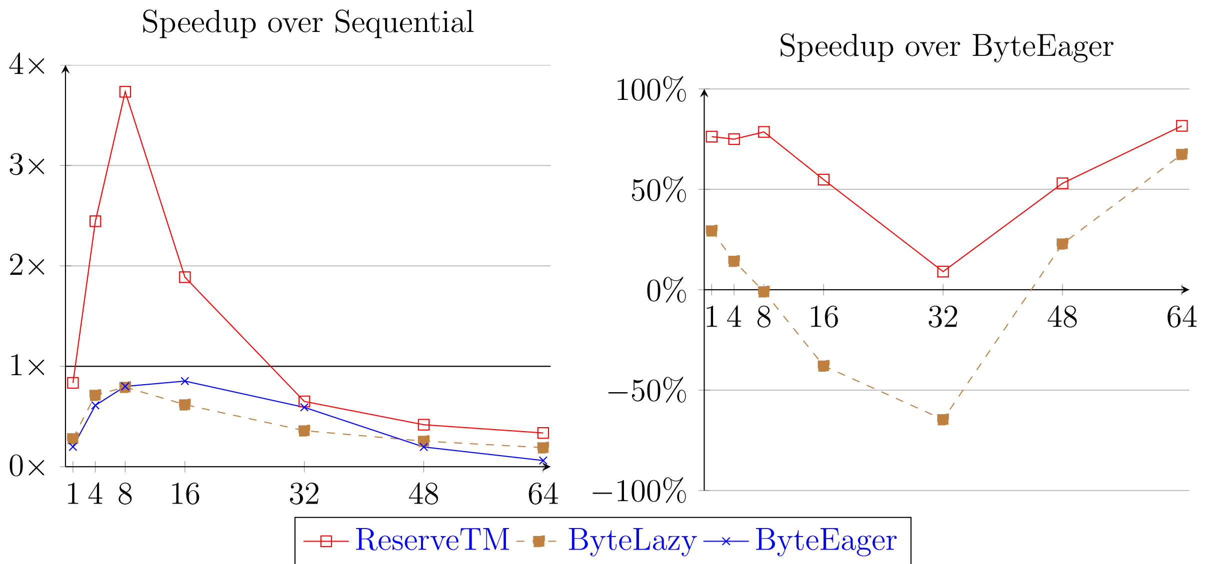


Figure 4.5: Speedup of STM threads for kmeans-high++

For kmeans, any access to an address consists of a `TM_READ` immediately followed by a `TM_WRITE`. Since all addresses that are read, are written to, releases cannot be used as it is not possible to release an address in the write set of a transaction. However, compression can collapse the 2 instrumentation points into a single `TM_WRITE_RESERVE` call. The dependency analysis in section 3.3.2 removes all instrumentation from the for loop in Figure 4.3, since each loop iteration depends on the previous one. Further, since a write to `new_centers[index][0]` is dependent on a write to `*new_centers_len[index]` a single write reservation is sufficient for the entire transaction. Hence, each transaction only requires a single `TM_WRITE_RESERVE` instrumented call.

Figure 4.4 and 4.5 demonstrate how the use of the ReserveTM primitives give consistent improvements over both eager and lazy versioning. We show significant improvements over sequential performance, up to 16 cores with low contention and 8 cores with high contention. In both configurations, as we add more threads the contention from writer acquisition dominates the benchmark. Though contention is restricted to a single write reservation in a transaction, expensive cache line invalidations across NUMA nodes degrade the performance of the benchmark with more threads.

Our results show an important trade-off between eager and lazy versioning. Lazy versioning has slightly better performance than eager versioning under low contention. This is due to the lazy commit sequence applying all updates to the `new_centers` in bulk at commit time. The sequential array access benefits from processor caching logic. On the other hand, eager versioning is updating the array in-place in an expensive loop that includes acquiring metadata locks. Under high contention, lazy versioning performs poorer than eager versioning since transactions are more likely to abort during commit-time validation. ReserveTM enjoys the benefits of in-place updates, while performing efficient sequential array access as the for loop does not have any runtime instrumentation.

4.2.2 Yada

The yada benchmark implements Delaunay mesh refinement. Yada transactions operate on a mesh data structure which utilizes vectors and trees to represent connections between mesh elements. Mesh traversal incurs significant contention, as transaction working sets are large and likely to conflict. Releases enable the benchmark to avoid livelock due to the contention with more threads.

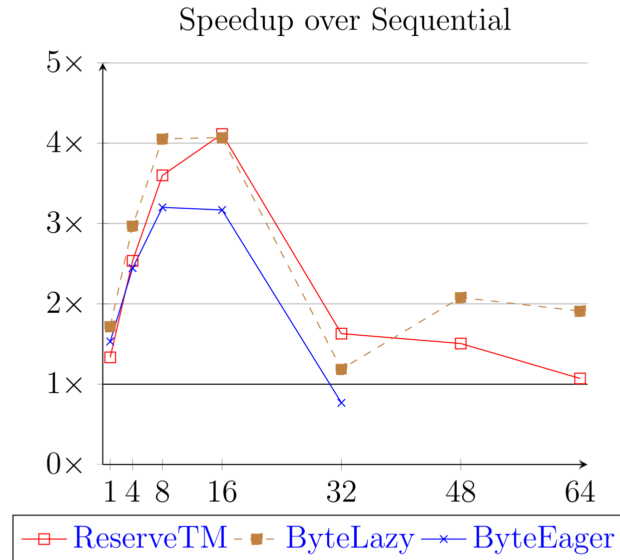


Figure 4.6: Speedup of STM threads for yada++

In Figure 4.6, increasing the running threads to 32 threads degrades the performance of all algorithms. Beyond 32 threads, eager conflict detection in ByteEager causes livelock, while ByteLazy and ReserveTM continue to make forward progress. For ByteLazy, lazy conflict resolution favored transactions that committed early. As a result, ByteLazy always committed at least one of the conflicting transaction and continued to operate beyond 32 threads. Eager conflict detection favored transactions with smaller reserved read sets. However, the large transaction working sets caused transactions to repeatedly be in conflict, causing livelock for ByteEager. Releases reduced the reserved read set of a transaction, reducing the opportunity for conflicts and avoided livelock. Thus, the performance of ReserveTM degrades more gracefully than ByteEager. We expect ReserveTM to livelock with sufficient concurrent threads.

A shared work queue in yada also increases conflicts between transactions. The queue elements are stored in an array-backed heap. Whenever yada removed an element from the heap, it removed the root node and replaced it with the last element in the heap. Modifying the root node caused a write conflict between concurrent dequeue operations. This caused concurrent dequeues to always have conflicting write sets. None of the algorithms were able to exploit concurrency during concurrent dequeues.

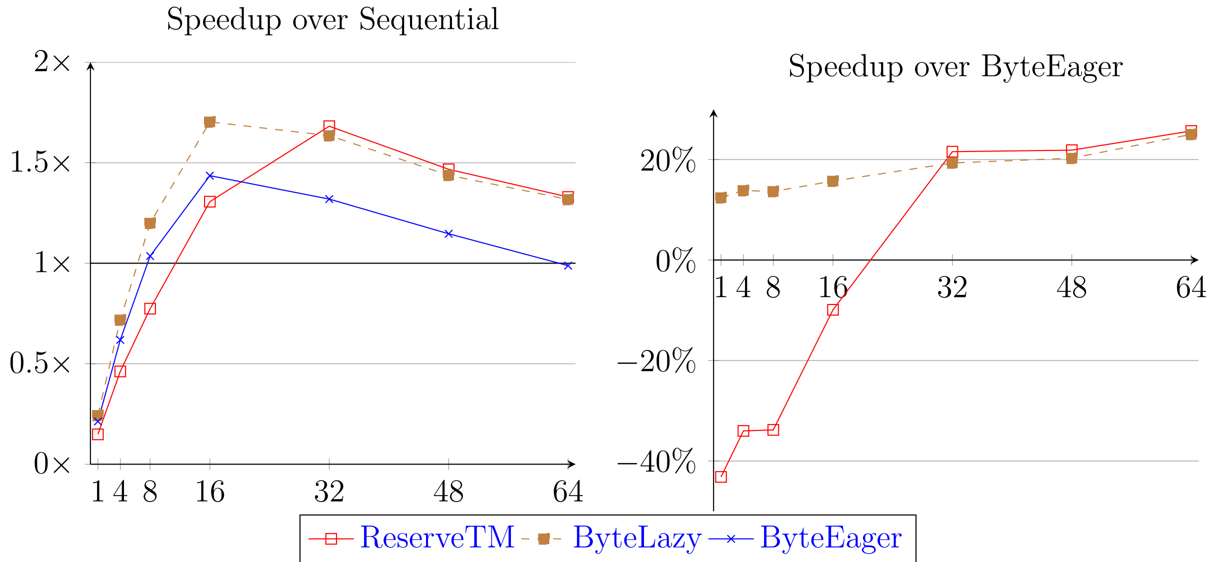


Figure 4.7: Speedup of STM threads for vacation-low++

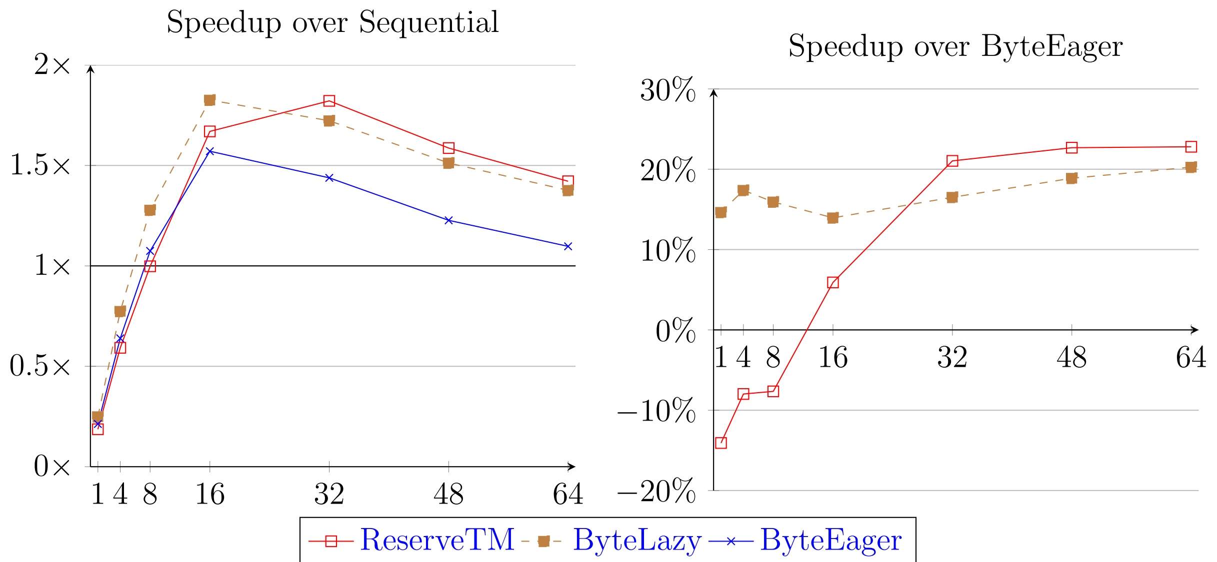


Figure 4.8: Speedup of STM threads for optimized vacation-low++

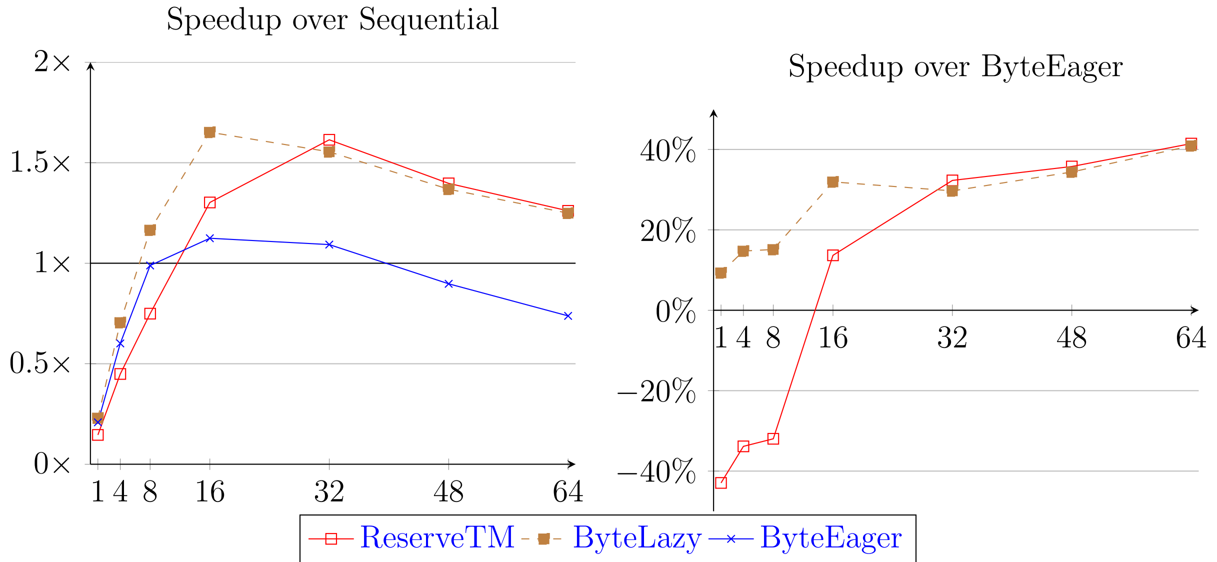


Figure 4.9: Speedup of STM threads for vacation-high++

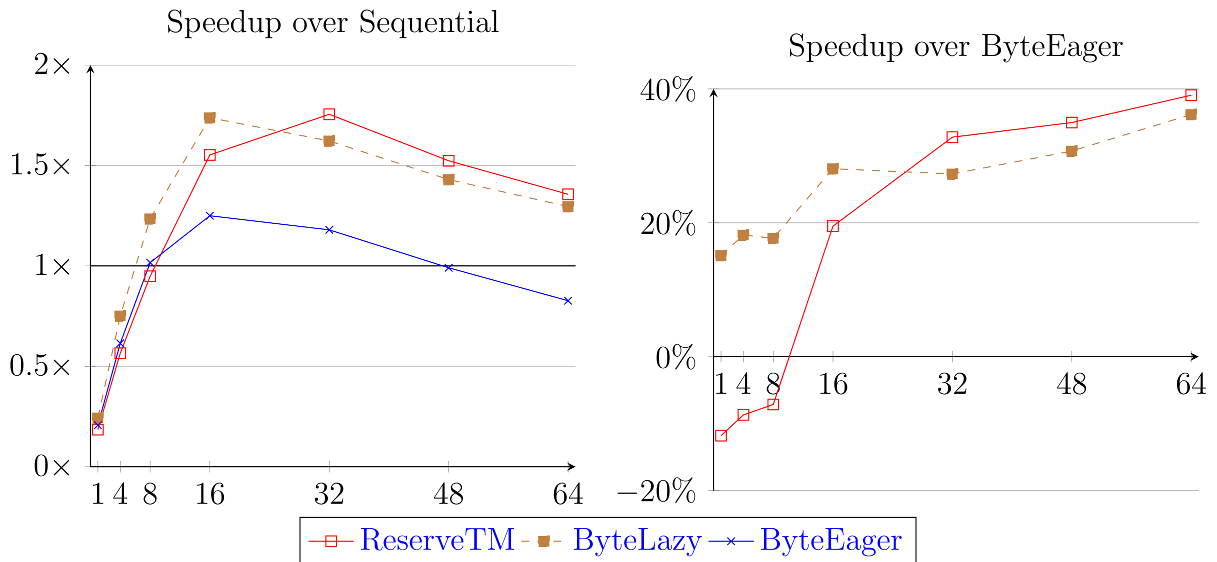


Figure 4.10: Speedup of STM threads for optimized vacation-high++

```
TM_BEGIN();
long bill = MANAGER_QUERY_CUSTOMER_BILL(managerPtr, customerId);
if (bill >= 0) {
  MANAGER_DELETE_CUSTOMER(managerPtr, customerId);
}
TM_END();
```

Figure 4.11: Vacation cancellation

4.2.3 Vacation

Vacation simulates a travel reservation system, modifying reservations and customers as elements in a red-black tree. There are 3 types of transactions that occur: reservations, cancellations and updates. The reservation transaction finds the item to reserve, checks the price, adds a new customer to the database and finally creates a reservation record. The cancellation transaction searches for a customer based on an *id* and then deletes the customer from the tree if their bill is 0. The update transaction is the simplest, which adds or removes items from the inventory by looking up the item in the inventory tree and updating its associated record.

Most vacation transactions perform writes after accumulating a large read working sets from tree traversals. The large read sets prevent transactions from concurrently executing. Even though transactions may operate on different branches of a tree, if a write needs to re-balance the red-black tree, it conflicts with the read set of another in-flight transaction. As a result, Figure 4.7 and 4.9, show ByteEager to perform poorly as we increased threads. ReserveTM performs better than ByteEager, matching the performance of ByteLazy, as it is able to release its read set while doing tree traversals.

Surprisingly, ReserveTM had poorer performance than ByteEager at lower thread counts. Our profiling determined the cause to be re-reads of released addresses. The re-reads were due to redundant lookups in the benchmark implementation. Figure 4.11 lists the cancellation transaction. The transaction performs 2 lookups of `customerId` in the customer tree. The first lookup finds the customer record and gets the balance of the customer's bill and the second lookup is to delete the customer. The reservation transaction suffers in the same manner as it does a 3 separate lookups for an item: the initial lookup for the presence check, then for the price and lastly to make the reservation. We modified the benchmark to avoid the unnecessary lookups and were able to improve the performance of ReserveTM at lower thread counts as compared to ByteEager. Figure 4.8 and 4.10, show results from using the optimized benchmark.

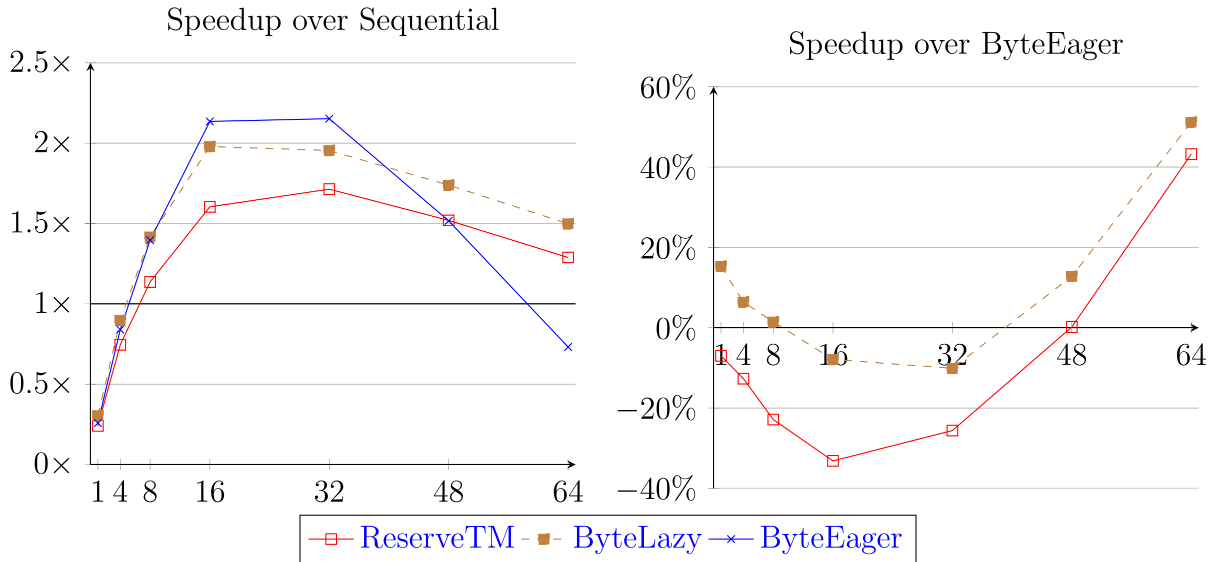


Figure 4.12: Speedup of STM threads for genome++

4.2.4 Genome

The genome benchmark takes a number of DNA subsequences and constructs the original sequence from them. This benchmark, is notable for containing 2 phases. The first phase populates a shared hash table with unique sequences from the subsequences, while the second phase builds a set of unmatched segments and dequeues from it to construct the original sequence. Most contention came from hash table operations. The hash table buckets consisted of a linked list of nodes sorted by their key. To insert an element into the hash table, the key was hashed to find the hash bucket and then the list was iterated over to see if an element for the node already existed, before inserting it.

With ByteEager, most of the transaction aborts were from writers that blocked on readers. ReserveTM reduced the number of aborts, but at the cost of more reader aborts. Since the workload was very read heavy, figure 4.12 illustrates ReserveTM to generally have poorer performance than byteEager. However, ByteEager performance degraded sharply when increasing the contention with more threads. Thus ReserveTM had better performance with 64 threads.

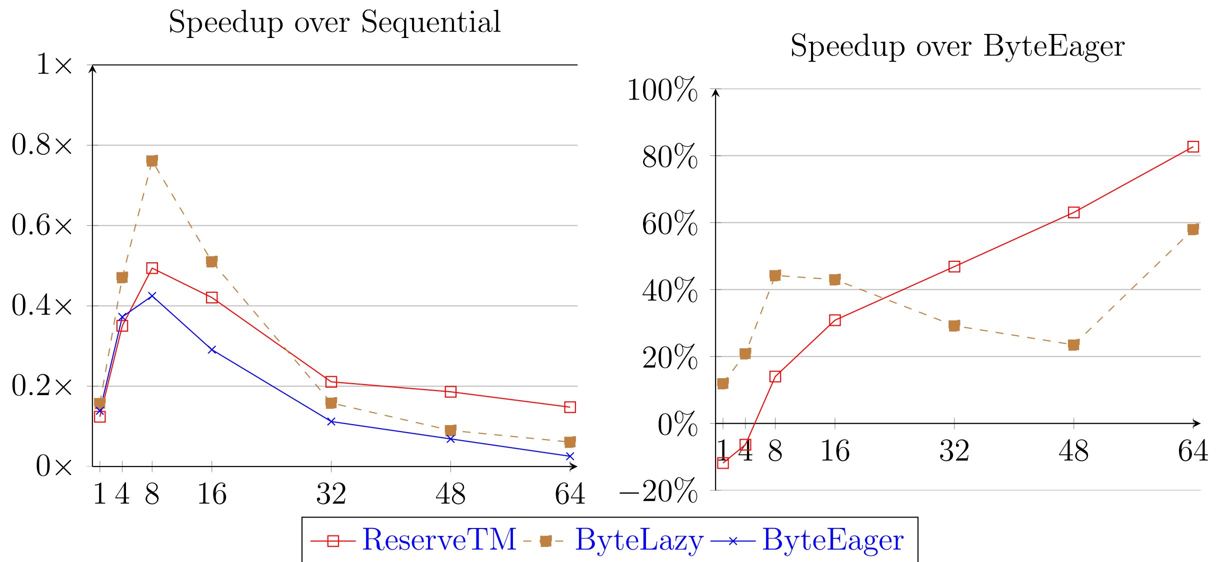


Figure 4.13: Speedup of STM threads for intruder++

4.2.5 Intruder

Intruder models a transactional network intrusion detection system. It consists of 3 transactions. 2 of these transactions pop items from a queue, while the 3rd processed trees and linked list data structures. We observed from Figure 4.13, that none of the algorithms offered any scalability improvements, as they all were slower to execute than the sequential version of the benchmark. The poor performance was attributed to early writes that would consistently cause conflicts. As a result, more of the processing time was spent resolving conflicts and rolling back changes, rather than doing actual work.

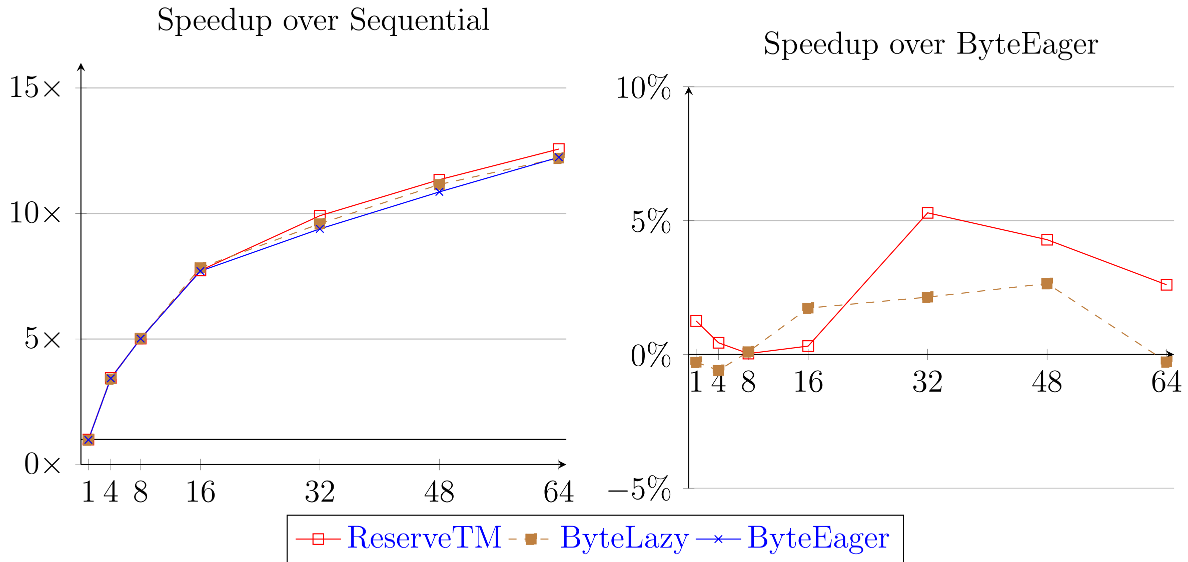


Figure 4.14: Speedup of STM threads for labyrinth++

4.2.6 Labyrinth

The labyrinth benchmark applied Lee's algorithm [16] to solve maze routing. The benchmark makes a private copy of the global three-dimensional grid, without runtime instrumentation, and works off the copy to find a path in the maze. Once the benchmark finds a path, it attempts to add the path back into the global grid while revalidating points along the path. This approach avoids conflicts while performing path finding, as working off the grid copy does not require runtime instrumentation.

As a result of this design, privatization is built into the algorithm and aborts are scarce. Figure 4.14 shows little additional speed up that a specific STM algorithm can offer. All the algorithms have very similar performance whereby the speedup tapers off at higher thread counts. ReserveTM offers a very slight improvement over the other algorithms, as it is able to use compression to reduce instrumentation when reading the grid for validation and adding a path.

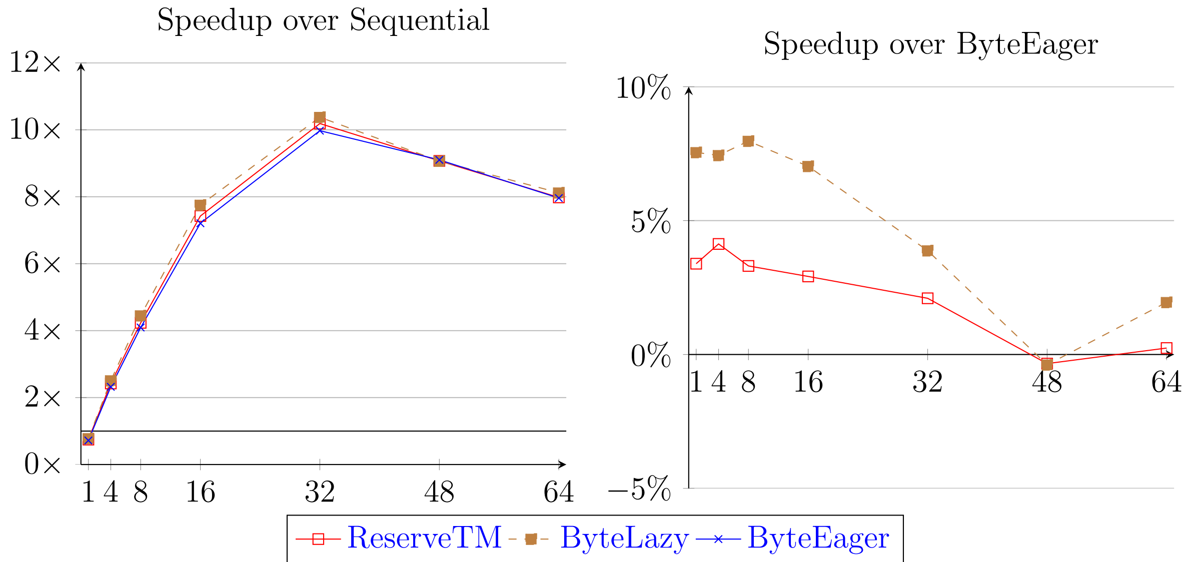


Figure 4.15: Speedup of STM threads for ssca2++

4.2.7 SSCA2

The Scalable Synthetic Compact Applications 2 (SSCA2) [3] benchmark executes mostly small transactions on a large multi-graph. There is hardly any contention, as very little time is spent executing transactional code, and the read and write sets of transactions are quite small. As a result, similar to labyrinth, Figure 4.15 show the performance of STM algorithms do not differ much as most of the benchmark operates outside of a transaction.

Chapter 5

Conclusions and Future Work

ReserveTM improves the performance and scalability of eager versioned STMs by strongly embracing privatization and eager releases. Reservations enable instrumentation-free access to shared memory while releases enable additional concurrency between readers and writers.

For a runtime to fully realize the benefits of ReserveTM, it must support a new set of primitives that explicitly communicate when an address has been privatized. The primitives reduce instrumentation overhead while still enabling interleaving of transactions.

Our evaluation showed significant performance gains which either surpassed the performance of lazy versioning or narrowed the difference between eager and lazy runtimes. The exception was with genome where the performance gains were only visible at higher thread counts.

5.1 Future Work

Through experimentation with manual instrumentation we found that the ReserveTM provides a highly effective means to communicate with the STM runtimes. Manual instrumentation, as opposed to compiler instrumentation, is brittle and can easily be incorrectly applied. We plan on implementing ReserveTM within a compiler to transparently instrument transactions. As compilers often suffer from over-instrumentation, we believe that reservations can be use to reduce the impact of over-instrumented addresses.

We were able to demonstrate that modifications could be made to byte-locks to support reservations and releases. In order to validate our claim that ReserveTM can be applied to

other eager versioned algorithms, we plan on adapting TinySTM to support reservations and releases.

The STAMP benchmark suite also did not offer enough variance in workloads, as some of the benchmarks were impossible to improve upon. We intend to evaluate ReserveTM on additional benchmarks such as STMBench7 [10].

References

[1] Intel® Transactional Memory Compiler and Runtime Application Binary Interface. Intel® Corporation, November 2008.

[2] Intel Architecture Instruction Set Extensions Programming Reference, 2012. http://software.intel.com/file/41604.

[3] David A Bader and Kamesh Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th international conference on High Performance Computing*, pages 465–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[4] Luke Dalessandro, Dave Dice, Michael L. Scott, Nir Shavit, and Michael F. Spear. Transactional Mutex Locks. In *16th International Euro-Par Conference on Parallel Processing: Part II*, pages 2–13, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[6] Dave Dice and Nir Shavit. TLRW: Return of the Read-Write Lock. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 284–293, New York, NY, USA, 2010. ACM.

[7] Aleksandar Dragojevic, Yang Ni, and Ali-Reza Adl-Tabatabai. Optimizing Transactions for Captured Memory. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, pages 214–222, New York, NY, USA, 2009. ACM.

[8] Pascal Felber, Christof Fetzer, Ulrich Müller, Torvald Riegel, Martin Süßkraut, and Heiko Sturzrehm. Transactifying Applications using an Open Compiler Framework. In *2nd ACM SIGPLAN Workshop on Transactional Computing*, August 2007.

[9] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic Performance Tuning of Word-Based Software Transactional Memory. In *13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246, New York, NY, USA, 2008. ACM.

[10] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 315–324, New York, NY, USA, 2007. ACM.

[11] Ruud A Haring, Martin Ohmacht, Thomas W Fox, Michael K Gschwind, David L Satterfield, Krishnan Sugavanam, Paul W Coteus, Philip Heidelberger, Matthias A. Blumrich, Robert W Wisniewski, Alan Gara, George Liang-Tai Chiu, Peter A Boyle, Norman H Chist, and Changhoan Kim. The IBM Blue Gene/Q Compute Chip. *IEEE Micro*, 32(2):48–60, 2012.

[12] Tim Harris, James Larus, and Ravi Rajwar. Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[13] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software Transactional Memory for Dynamic-Sized Data Structures. In *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[14] Maurice Herlihy and J Eliot B Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th annual international symposium on computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[15] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2004 International Symposium on Code Generation and Optimization*, pages 75–86, Palo Alto, CA, USA, March 2004.

[16] C. Y. Lee. An Algorithm for Path Connections and Its Applications. *Electronic Computers, IRE Transactions on*, EC-10(3):346–365, 1961.

[17] V J Marathe, M F Spear, and M L Scott. Scalable Techniques for Transparent Privatization in Software Transactional Memory. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 67–74, 2008.

[18] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer, III, and Michael L. Scott. Lowering the Overhead

of Nonblocking Software Transactional Memory. In *1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, June 2006.

[19] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *2008 IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008.

[20] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[21] William N. Scherer, III and Michael L. Scott. Advanced Contention Management for Dynamic Software Transactional Memory. In *Proceedings of the Twenty-Fourth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[22] Martin Schindewolf, Albert Cohen, Karl Wolfgang, Andrea Marongiu, and Luca Benini. Towards Transactional Memory Support for GCC. In *1st GCC Research Opportunities Workshop (2009)*, January 2009.

[23] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[24] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing Isolation and Ordering in STM. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–88, New York, NY, USA, 2007. ACM.

[25] Michael F. Spear. Lightweight, Robust Adaptivity for Software Transactional Memory. In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 273–283, New York, NY, USA, 2010. ACM.

[26] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization Techniques for Software Transactional Memory. Technical Report 915, February 2007.

[27] Amy Wang, Matthew Gaudet, Peng Wu, José Nelson Amaral, Martin Ohmacht, Christopher Barton, Raul Silvera, and Maged Michael. Evaluation of Blue Gene/Q Hardware Support for Transactional Memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, pages 127–136, New York, NY, USA, 2012. ACM.

[28] Cheng Wang, Wei-Yu Chen, Youfeng Wu, Bratin Saha, and Ali-Reza Adl-Tabatabai. Code Generation and Optimization for Transactional Memory Constructs in an Unmanaged Language. In *2007 International Symposium on Code Generation and Optimization*, pages 34–48, March 2007.

[29] Qingping Wang, Sameer Kulkarni, John Cavazos, and Michael Spear. A Transactional Memory with Automatic Performance Tuning. *ACM Transactions on Architecture and Code Optimization*, 8(4):54:1–54:23, 2012.

[30] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the Tires of Software Transactional Memory: Why the Going Gets Tough. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, pages 265–274, New York, NY, USA, 2008. ACM.