The Dissertation Committee for Jeffrey Michael Napper
certifies that this is the approved version of the following dissertation:

# Robust Multithreaded Applications

Committee:

Lorenzo Alvisi, Supervisor

Michael Dahlin

Keith Marzullo

Harrick Vin

Emmett Witchel

# Robust Multithreaded Applications

by

**Jeffrey Michael Napper, B.S.; M.S.**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**

May 2008

To my Mother and Father.

# Acknowledgments

All good stories start somewhere. Mine begins with a happy job where I was glad to have a friend for my boss. Pat Pitt has always been someone with a much more intelligent view and a great lunch partner. And yet, I still wanted more research in my job. So, I talked to one of my old teachers, and that's when things got a bit out of hand. Little did I know then that I would gain a good friend and advisor in the course of such a long journey.

I would like to thank sincerely Lorenzo Alvisi for taking the advisor role seriously. Not only has he helped me along the path to becoming a better scientist and writer, but also, I believe, taught me to appreciate elegance and good design in all things.

Mike Dahlin has also been a source of productive inspiration and instruction for me. Although I can't say that I ever beat him in an argument, I can say that I had fun trying. The rest of my committee: Emmett Witchel, Harrick Vin, and Keith Marzullo also gave me helpful feedback, both on the beginning of this thesis and on career advice afterwards.

What a long, strange trip it's been at UT Austin. After many, many years (and three degrees!) I can't possibly thank all the people that have made it so much fun. In the beginning of graduate school there were many students: Jayaram, Ravi, Stefano, Paolo, Arun, Alison, Suvrit, and Sergei, but by the end they were almost all new: Taylor, Allen, Harry, and Brian. You have all helped me survive. So long, and thanks for all the Thai (food). Good luck to those still on the way!

Not to be missed, I must thank Sara Strandtman, without whom our group certainly would not function, for saving me from many tough situations and for acquiring our beloved Gaggia. Further thanks also go to Gloria and Katherine who always provided help and advice when trapped by bureaucracy.

There have been many friends elsewhere too. Every internship (on both coasts and in the middle!) was a chance to meet new, great people. Although most were wonderful, a few have become good friends. I will especially look forward to further ski trips with Apu and Phoebe (and next time including Ana). Then, there are my fellow Austinites who make the city so hard to leave. I will always miss the food, music, people, and city attitude: especially, my long friends Stephen, Maggie, and Mat. I will not, however, ever miss the August temperatures.

I would like to thank my wonderful wife for helping me through the end times and for waiting through all the previous. I would definitely still be coasting along without you. Although you have the degree in math, I can still prove that one plus one is greater than two.

I would like to close by thanking my parents. Without their unending support and concern, I could not have finished. To my Mother, who showed me how to work hard by always working harder, I would like to say thanks for all the unconditional love and support. To my Father, who taught me patience by listening to me (and others) rant, I would like to say thanks for ignoring my faults and for always giving me good advice. This thesis is dedicated to you both. Further, I am happy to say that my family has grown (before and after graduate school), and I welcome(d) both Barbara and Ken as parents. I love you all and wish you all the happiness you've brought to me.

Jeffrey Michael Napper

*The University of Texas at Austin*
*May 2008*

# Robust Multithreaded Applications

Publication No. ____________________

Jeffrey Michael Napper, Ph.D.
The University of Texas at Austin, 2008

Supervisor: Lorenzo Alvisi

This thesis discusses techniques for improving the fault tolerance of multithreaded applications. We consider the impact on fault tolerance methods of sharing address space and resources. We develop techniques in two broad categories: conservative multithreaded fault-tolerance (C-MTFT), which recovers an entire application on the failure of a single thread, and optimistic multithreaded fault-tolerance (O-MTFT), which recovers threads independently as necessary. In the latter category, we provide a novel approach to recover hung threads while improving recovery time by managing access to shared resources so that hung threads can be restarted while other threads continue execution.

# Contents

**Acknowledgments** **v**

**Abstract** **viii**

**Chapter 1 Introduction** **1**

1.1 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

1.2 Outline of the Thesis . . . . . . . . . . . . . . . . . . . . . . . . . 6

**Chapter 2 Modeling Multithreaded Applications** **8**

2.1 Capricious Failure Model . . . . . . . . . . . . . . . . . . . . . . . 9

2.2 Non-determinism . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.3 The JVM as a State Machine . . . . . . . . . . . . . . . . . . . . . 11

**Chapter 3 Conservative MT Fault-Tolerance** **13**

3.0.1 Asynchronous Commands . . . . . . . . . . . . . . . . . . . 14

3.0.2 Non-deterministic Commands . . . . . . . . . . . . . . . . . 16

3.0.3 Non-deterministic Read Sets . . . . . . . . . . . . . . . . . 17

3.0.4 Output to the Environment . . . . . . . . . . . . . . . . . . 19

3.0.5 A Note on Restrictions R0–R6 . . . . . . . . . . . . . . . . 20

3.1 Implementation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

3.1.1 Nondeterministic Commands . . . . . . . . . . . . . . . . . 22

3.1.2 Nondeterministic Read Sets . . . . . . . . . . . . . . . . . . 22

3.1.3 Garbage Collection . . . . . . . . . . . . . . . . . . . . . . . 27

3.1.4 Environment Output . . . . . . . . . . . . . . . . . . . . . . 28

3.2 Experiments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

3.3 Related Work: JVM . . . . . . . . . . . . . . . . . . . . . . . . . . 35

3.4 Summary . . . . . . . . . . 37

**Chapter 4 Optimistic MT Fault-Tolerance** **38**

4.1 Related Work: Software Data Isolation . . . . . . . . . . 39

4.2 Related Work: Detecting Faults . . . . . . . . . . 41

4.3 Reducing the Effective Recovery Unit . . . . . . . . . . 42

4.4 Summary . . . . . . . . . . 43

**Chapter 5 Software Transactional Memory** **45**

5.1 Related Work: STM . . . . . . . . . . 46

5.2 System Model . . . . . . . . . . 48

5.2.1 Serializability vs. Linearizability . . . . . . . . . . 50

5.3 Lock-Free Transactions . . . . . . . . . . 51

5.3.1 Ensuring One-Copy Serializability . . . . . . . . . . 57

5.3.2 Ensuring Lock-Freedom . . . . . . . . . . 59

5.4 Evaluation . . . . . . . . . . 63

5.5 Using Timestamps . . . . . . . . . . 65

5.5.1 Ensuring One-Copy Serializability . . . . . . . . . . 71

5.5.2 Ensuring Lock-Freedom . . . . . . . . . . 73

5.6 Evaluation of Timestamps . . . . . . . . . . 76

5.7 Summary . . . . . . . . . . 82

**Chapter 6 Mayfly: O-MTFT and Leases** **84**

6.1 Hung Threads . . . . . . . . . . 84

6.2 Design . . . . . . . . . . 86

6.3 Lock-free Resource Access . . . . . . . . . . 87

6.4 Recovery . . . . . . . . . . 88

6.5 When Optimism Fails . . . . . . . . . . 90

**Chapter 7 Conclusions** **93**

**Bibliography** **95**

**Vita** **108**

# Chapter 1

# Introduction

Highly concurrent services, such as internet services or transaction processing, increasingly use servers built on multicore processors that are designed to provide hardware parallelism. Reliability is an extremely important metric for internet services—downtime incurs significant costs per hour in lost revenue [56, 95] and often has unforeseen consequences such as canceled airline flights or ATM network outages [81]. Although much downtime can be attributed to human error [89], software and hardware faults persist.

This thesis details how to design and build highly concurrent services that tolerate both hardware and software errors. Highly concurrent services are constructed using application processes (we also use the simplified *application*) that maintain pools of several threads to exploit increasing available hardware concurrency in order to maximize throughput [90, 110, 114]. Although highly concurrent services could be implemented using several single-threaded application processes, using threads yields multiple advantages. First, because all threads share an address space within an application process, context switching is faster (cooperative user-level threads can achieve context switches on the order of nanoseconds [13]) and cached data and allocated resources (files, sockets, etc.) are easier to share. Second, thread creation requires less time than application process creation. Third, threads maintain less state, making them scalable; for example, Capriccio [111] supports up to 100K threads in a single application process.

While the multithreaded (MT) applications composing highly concurrent services are increasingly important, concerns for reliability remain. Higher chip densities, lower voltage requirements, and combinatorial logic susceptible to single event upset (SEU) radiation errors due to cosmic rays are predicted to increase transient hardware errors in the performance-oriented server architectures used in data centers to support high-concurrency services [25, 82, 102, 118]. Indeed, transient hardware errors have been seen to outnumber hard errors by an order of magnitude [103].

Fault-tolerance techniques can improve the reliability of highly concurrent services. For example, the reliability of a service can be improved by moving its implementation from a single, centralized server to a collection of coordinated identical replicas through a technique known as State Machine Replication (SMR) [63, 99]. The idea is that, as long as faulty replicas are eventually recovered, one replica will always be available to provide the service. The unit of recovery in fault-tolerance techniques—called the *recovery unit*—such as SMR consists traditionally of the entire application process running on a replica (perhaps containing several threads). We call the resulting application *conservative multithreaded fault-tolerant* (C-MTFT) because an error requires recovery of the entire process.

**Conservative multithreaded fault-tolerance.** We discuss in this thesis several techniques for building C-MTFT multithreaded services, and we describe an implementation of these techniques for the Java Virtual Machine. Our techniques achieve C-MTFT without significant modifications to the application. C-MTFT addresses the non-determinism introduced by multithreading: access to shared data is made deterministic so that the entire application can be reliably replicated as a single recovery unit.

SMR tolerates both software and hardware failures, as replicas of components can execute on physically distinct machines. However, hardware is relatively reliable; in fact, software errors dominate the downtime of applications [37, 60, 68, 108]. Significant reductions in hardware costs can hence be achieved by using an approach that protects only against software faults. Rollback recovery [28] leverages the availability of correct hardware by replicating processes in time rather than space: the application is restarted on the same hardware (that is assumed to be correct) and the entire process is recovered to a pre-failure state. Simply restarting the application often restores service [18], while using the same hardware reduces the financial costs of fault-tolerance.

Current MT applications must rely on C-MTFT for reliability because the failure of even a single thread can leave the data shared among threads in an inconsistent state. Consequently, all threads must restart, and the application can lose all pending requests although correct threads could continue if provided consistent data. Subsets of the threads in an application that share data can be restarted independently to ensure data consistency, and this technique has been shown effective [18]. However, this approach easily degenerates into restarting the entire application as it does not address the problem of shared memory access. For example, in event-driven programming, thread pools share input and output event queues, implying that transitively all application threads share data.

To reduce the time spent on recovery, this thesis explores scaling down the size of the recovery unit to a single thread so that an MT service may contain many recoverable components tolerant to transient errors. We call this approach *optimistic multithreaded fault-tolerance* (O-MTFT) because it attempts to improve the performance of recovery by reducing the state recovered, but does not succeed in all cases.

**Optimistic multithreaded fault-tolerance.** O-MTFT approaches maintain some availability during a thread failure because application-specific data consistency checks (for example, range value checks) can often limit errors to a single thread, and correct threads need not participate in recovery. Access to consistent shared memory is available to correct threads with O-MTFT recovery techniques, allowing cheaper protocols for some failures because correct threads may continue execution. Hence, O-MTFT overcomes a significant limitation of the conservative approach, as C-MTFT requires the entire application to recover on a single thread fault.

O-MTFT enables threads in an application to fail independently without (in most cases) stopping or requiring a restart of other threads, changing the granularity of recovery unit from the entire application to threads. Many software faults (such as segmentation faults or race conditions) can be addressed at this level. Our approach can provide fine-grained robustness to transient errors (or, *heisenbugs* [38]) by restarting faulty threads. Further, allowing correct threads to continue execution rather than forcing them to restart can achieve graceful degradation to persistent errors (or, *bohrbugs* [38]). For example, an approach to error handling in a webserver with continuous service might be to learn which requests activate faults (that is, those that trigger a bohrbug). The system could refuse these requests while continuing to provide service to requests that don't activate faults or lead to occasional heisenbugs.

We enable independent thread recovery by regulating resource sharing through a transactional, non-blocking communication channel, providing faster thread recovery and tolerance to hung threads. Transactions [36] can provide simple fault recovery for shared data, requiring only an abort of the faulty thread's transaction so that shared data is always consistent, and only the faulty thread must be recovered. Though transactions provide the appearance of atomicity, they are traditionally implemented using data locking primitives that require complex system support and are generally conservative. If a thread fails while holding a crucial system lock, the full application must restart because the system lock will not be obtainable by correct threads.

Software Transactional Memory (STM) [101] provides a transactional interface to shared data that also has progress guarantees, ensuring the desirable properties of O-MTFT: 1) shared memory is always consistent for correct threads, and 2) correct threads can always access shared memory. These two properties allow correct threads to continue without recovering shared memory when a thread fails. We describe a novel STM design [83] that implements these two properties using serializability (transactions appear to occur in some serial order) and lock-freedom (some transaction by a correct thread eventually will commit).

Using an STM to access shared memory enables nonblocking communication between threads to facilitate independent recovery even in the presence of *hung threads*. Hung threads perform no work—for example, in the case of deadlock if a lock is unobtainable—or useless work—for example, when a thread is caught in an infinite loop because of inconsistent data. These errors can result from transient conditions (such as race conditions or SEU errors due to cosmic rays) or persistent conditions (such as software bugs).

In addition to tolerating hung threads by guaranteeing access to shared memory for correct threads, we attempt to recover them to ensure overall progress. We detect hung threads caused by an infinite loop or deadlock by leasing [35] cpu time and wall-clock time, respectively, to threads. By enabling the recovery of single hung threads, we 1) increase the fault coverage of multithreaded systems to include hung threads, 2) increase the application's performance by keeping correct threads performing useful work where possible, and 3) reduce recovery time by bounding downtime due to hung threads.

Finally, by managing access to shared memory, O-MTFT techniques enable faster and cheaper thread-specific recovery so that an application can recover each thread using application semantics rather than recovering a large group of threads as in C-MTFT. For example, knowing that a client can retry the request, a webserver can simply drop the request processed by a failed thread and start a new worker thread to maintain concurrency without worry that the correct threads will deadlock waiting for the faulty thread to perform an action.

## 1.1 Contributions

The main contribution of this thesis is to explore techniques for enabling and improving the recovery of multithreaded services. We developed two broad approaches that target recovery units at different scales. Our conservative multithreaded fault-tolerant techniques recover the entire application to ensure consistency even if the failed thread corrupts shared data, but do so at the expense of availability: a single fault in any thread can bring down the entire application. Our optimistic multithreaded fault-tolerant techniques instead provide focused recovery of small groups or even individual threads, allowing application-specific recovery strategies. Further, optimistic multithreaded fault-tolerance approaches enable recovery from hung threads that perform no useful work by leasing execution resources, bounding the time lost to such failures.

Specific contributions are:

- *Replicating MT Applications.* We demonstrate an implementation of the State Machine Replication technique applied to MT applications. We modify the Java Virtual Machine (JVM) according to our multithreaded C-MTFT model to provide primary-backup fault-tolerance transparently to applications [85] (though the system can be modified for other forms of hardware replication). No modifications to the application are required, though we provide different replication mechanisms depending upon the application's use of shared data. Assuming access to all shared data is guarded by critical sections (and execution is otherwise piecewise deterministic [106]), we replicate the entry order to critical sections between the primary and backup to ensure that shared data is consistent between the replicas (similar to [10, 34, 61, 66]). In a different approach, we replicate thread scheduling, ensuring piecewise determinism even in the presence of double-checked locking (similar to [1, 104]).

- *Software Transactional Memory.* We describe the first lock-free, serializable [91] STM. Our multiversion STM provides for more concurrency than linearizable [53] STM while maintaining a strong progress guarantee: eventually some transaction will commit. Under a model in which threads can fail by halting arbitrarily, some correct thread will always eventually commit a transaction (although starvation can still occur where a particular correct thread is never able to commit a transaction). Maintaining multiple versions provides both more opportunities for concurrency and previous data values that can be used for recovery of individual threads.

- *MT-specific Recovery Techniques.* We develop novel optimistic fault-tolerant recovery techniques to improve the availability of MT services. Our techniques involve: 1) leasing thread execution to bound downtime due to hung threads, 2) using transactions to ensure consistency of data shared between threads in the presence of failures, and 3) using obstruction-free data structures to improve the availability of shared data in the presence of failures. These three techniques allow our implementation, called Mayfly, of O-MTFT for the Java Virtual Machine to provide good throughput in the presence of a stream of faulty requests that cause errors in a webserver. Mayfly enables the developer to use application semantics to recover individual threads.

## 1.2 Outline of the Thesis

The rest of this thesis is organized as follows.

Chapter 2 provides the system model and background on which we base discussion of our multithreaded recovery techniques.

Chapter 3 discusses our approach to conservative multithreaded fault-tolerance. We describe our primary-backup implementation of the Java Virtual Machine and several techniques for implementing C-MTFT in the Java language.

Chapter 4 describes a novel optimistic approach to multithreaded fault-tolerance, reducing the size of the recovery unit to improve the availability of MT services.

Chapter 5 gives the details of our implementation of a lock-free, serializable, multiversion STM. Our STM provides progress guarantees to ensure that correct threads can continue to access shared data, while the serializability correctness criterion provides for safe and concurrent execution of transactions. Multiple versions compose a history that can be used during recovery of single threads for replay of shared data accesses.

Chapter 6 discusses our implementation of O-MTFT called Mayfly. Mayfly implements leased execution for threads in the Java Virtual Machine. In this way, hung threads can be recovered using lease expiration as a failure detector. We provide evaluation results to show the low cost of recovery in the system.

Finally, Chapter 7 summarizes the thesis and proposes directions for future research.

# Chapter 2

# Modeling Multithreaded Applications

We begin by describing a model of multithreaded applications. This model provides a foundation for our discussion of fault-tolerance. A *system* provides a service to clients and is built as a set of recovery units. Although we will be more precise shortly, roughly speaking, a recovery unit contains the data required to recover from a fault and may contain any number of threads and their state. Each *thread* is an execution context that is modeled by a state machine (we will use the terms thread and state machine interchangeably). Threads may execute concurrently (as on a multiprocessor). The relative speed between threads is unbounded, and correct threads may stop temporarily at any time for a finite, but unbounded, period (for example, when a page fault occurs).

A state machine is a set of *state variables* and *commands*, which respectively encode and modify the machine's state. A command reads a subset of the state variables, called the *read set*, plus, possibly, other inputs obtained from the *environment* (a set of state variables whose values are unpredictable and are not necessarily determined by the state machine); it then modifies a subset of state variables called the *write set*, and potentially produces some output to the environment. For a given command, the read and write sets are fixed. However, the values that these variables assume at each invocation of the command can change and depend upon the current state of the system and the precise memory consistency condition (for example, sequential consistency [64]) in effect for variables accessed by multiple threads.

Henceforth, we refer to the values of the variables as *read-set values* and *write-set values*. The sequence of commands, along with the corresponding read and write sets and their values, executed by a thread is called an *execution.* We call the set of all read-set (write-set) (values) of all the commands in an execution simply the read-set (write-set) (values) of the execution. A variable is *shared* between threads if it is present in either the read or write sets of at least two threads; otherwise, if a variable is not shared between any threads, it is *local* to a specific thread.

## 2.1 Capricious Failure Model

Faults cause a *faulty thread* to deviate from a *failure-free execution,* which is simply an execution in which no failures occur. A fault can cause a thread to commit two kinds of *errors:* 1) an *erroneous* write occurs when the write-set values of a command in an execution with a failure contain values different from those produced in a failure-free execution with the same sequence of commands (for example, when a bit in a value is flipped by an SEU error) and 2) an *errant* write occurs when the write-set contains variables different from that of a failure-free execution (for example, when a bit in an address is flipped by an SEU error). Further, the faulty thread may *crash* by halting unexpectedly. After a thread commits an error but has not yet crashed, the thread performs an *erroneous* execution.

Our model is similar to the Byzantine failure model [65], but lacks the allowance for intentional misbehavior. Threads in our model may exhibit arbitrary behavior (for example, by writing arbitrary values at arbitrary locations) but we explicitly do not consider the possibility that these faults may be caused by a malicious adversary who may alter the a priori probability of an arbitrary fault. We call this new model Capricious to distinguish it from the weaker Byzantine model.

While a faulty thread can always be determined by an omniscient observer, an erroneous execution may not be immediately identifiable by an application, allowing an erroneous execution to affect other threads. To recover from an error, both the faulty thread and the threads affected by the failure must be rolled back to a *correct state* corresponding to some pre-failure state. To determine the set of threads to roll back, we define which threads are affected by a failure:

**Capricious orphan:** a thread whose read set includes a variable whose value has been set by a faulty thread.

Now, we simply add the faulty thread to obtain the set of threads that must be rolled back after a failure:

**Recovery unit:** a set of threads containing the thread that executes the error causing the failure and all resulting Capricious orphans.

We discuss the correct state to which a recovery unit rolls back in later chapters as we describe different MT fault-tolerance techniques.

Determining the members of the recovery unit may incur significant overhead. Fault-tolerance techniques may, in a performance tradeoff, include more threads than necessary to reduce the need to track all Capricious orphans. We call the set recovered by a fault-tolerance technique the *effective recovery unit.* To ensure correctness, the effective recovery unit must be a superset of the recovery unit. A fault-tolerance technique must identify Capricious orphans in order to determine the effective recovery unit. In this thesis, we will consider two different techniques for determining such orphans—a conservative approach in Chapter 3 that considers all threads to be in the effective recovery unit, and an optimistic approach in Chapter 4 that attempts to minimize the recovery unit by reducing the length of erroneous execution.

## 2.2 Non-determinism

Fault-tolerance is concerned with replication used to recover or mask faults and thus with the read- and write-set values associated with commands to ensure that an execution of a state machine can be replicated. A *deterministic command* produces the same output and write-set values when given the same read-set values. Hence, two state machines started from the same initial state and executing identical sequences of deterministic commands with identical read-set values undergo the same sequence of state transitions and produce the same outputs. However, not all commands or read-sets are deterministic.

We identify three types of non-determinism in our model: 1) *non-deterministic commands* whose write-set values or output to the environment are not uniquely determined by their read-set values (for example, if the command produces an unpredictable sequence of events on the environment), 2) *asynchronous commands* that can appear anywhere in the sequence of commands executed by a state machine (for example, if the command is the result of a non-deterministic interrupt), and 3) *non-deterministic read sets* where any of the read-set values of the command are not uniquely determined by the previous sequence of commands executed by the state machine (for example, when the read-set contains input from the environment).

The *piecewise-deterministic assumption* [106] states that all non-deterministic events in the state machine can be identified. We make this assumption that the three types of non-determinism we have discussed encompass the sources of non-determinism in the MT applications we consider. These non-deterministic events can be characterized by a *determinant* [2] that can be used to determine the read-set values used during recovery. For example, if a command includes the current time as input from the environment (that is, it has a non-deterministic read set), the determinant might contain the precise time read as input so that during recovery the same value can be used regardless of the current time. Generally, from the previous description of the types of non-determinism, we characterize the corresponding determinants, respectively, to contain: 1) the write-set values or output to the environment that is not uniquely determined by the read-set values, 2) the order in which the asynchronous command appears in the sequence of commands executed by the state machine, and 3) the subset of non-deterministic read set values that are not uniquely determined by the sequence of commands executed previously by the state machine.

## 2.3 The JVM as a State Machine

We use the Java Virtual Machine (JVM) to explore our fault-tolerance techniques. Java programs are compiled into an architecture-independent bytecode instruction set. The compiled code is organized into classfiles, containing class definitions and methods according to the Java Virtual Machine Specification [72]. The JVM also defines standard libraries that provide supporting classes for various tasks (e.g., data containers, I/O, and windowing components). The JVM and standard libraries comprise the Java Runtime Environment (JRE). Java provides language-level support for multithreading, mutual exclusion (synchronized methods) and conditional synchronization (wait and notify methods).

We model the JVM as a set of cooperating state machines, each roughly corresponding to a Java thread. In particular, we choose as our state machines a set of *bytecode execution engines* (BEE) inside the JVM. Although BEEs do not explicitly exist as components of the JVM (for example, the bytecodes might be compiled into processor-specific machine code), we can conceptually associate a BEE with the set of functions that perform bytecode execution and track the state of each thread.

The commands of the BEE state machine are bytecodes, and the state variables are the values of memory locations accessible to the BEE. Each BEE has exclusive access to its own *local variables* and may share with other BEEs access to *shared variables.* The rest of this thesis discusses methods to ensure that each BEE replica processes the same sequence of commands, whether the replicas execute simultaneously as with a primary-backup system (Chapter 3) or later in time during recovery (Chapter 4).

# Chapter 3

# Conservative MT Fault-Tolerance

Conservative MT fault-tolerance (C-MTFT) describes techniques for multithreaded applications wherein the unit used for recovery or replication contains many threads that share data. In this chapter, we discuss C-MTFT while considering the entire application as a single fault-tolerant recovery unit. The salient characteristic that distinguishes C-MTFT from the optimistic techniques discussed later in Chapter 6 is that C-MTFT recovery units do not communicate through shared data. Threads that access the same shared variables belong to the same recovery unit in our conservative approach.

This chapter demonstrates the C-MTFT approach through the design and implementation of a fault-tolerant Java Runtime Environment that tolerates fail-stop failures. Our technique is based on the well-known work on State Machine Replication [63, 99]. This approach involves 1) defining a deterministic state machine as the unit of replication, 2) implementing independently failing *replicas* of the state machine, 3) ensuring that all replicas start from identical states and perform the same sequence of state transitions (replica coordination), and 4) guaranteeing the replication is transparent: each output-producing transition results in a single output to the environment, rather than a collection of outputs, one for each replica.

Our approach to C-MTFT is inspired by, and extends, the work of Bressoud and Schneider on *Hypervisor-based fault-tolerance* [17], which presents a strong case for achieving transparent fault-tolerance by 1) building a software layer (the hypervisor) that implements a virtual state machine over the underlying hardware and 2) implementing replica coordination in the hypervisor. To demonstrate their approach, Bressoud and Schneider had to build an hypervisor for (a subset of) the HP PA-RISC architecture because the architecture itself was not modifiable. Our focus on the Java Virtual Machine allows us to implement replica coordination directly into the JVM rather than building a hypervisor over unmodifiable hardware.

State machines must be deterministic for replication to work. Multithreaded applications are not typically deterministic. The focus of this thesis is how to manage the added non-determinism present in MT applications. We therefore systematically identify and eliminate the effects of non-determinism within the JVM in order to apply State Machine Replication. In doing so, we face the same issues (asynchronous exceptions, output to the environment, etc.) identified in [17]. However, we identify additional sources of non-determinism in our model as they pertain to C-MTFT: *asynchronous commands, non-deterministic commands* and *non-deterministic read sets* (see Section 2.2).

After we discuss these different sources of non-determinism, we present and evaluate two techniques for eliminating the non-determinism. The first technique forces each replica to perform the same sequence of monitor acquisitions; the second technique guarantees the same sequence of thread scheduling decisions. Both techniques ensure identical accesses to shared data at all replicas in order to eliminate non-determinism.

### 3.0.1 Asynchronous Commands

A command is *asynchronous* if it can appear anywhere in the sequence of commands processed by a BEE. Replicas of the same BEE might encounter a given asynchronous command at different points in their command sequences. In [17], hardware interrupts are asynchronous commands. Although there are interrupts in the JVM, they do not give rise to asynchronous commands. For example, our JVM performs I/O synchronously,[1] and any I/O completion interrupt that corresponds to a given bytecode is delivered before the execution of that bytecode completes. Programmers can use Java's multithreading to perform asynchronous I/O or events, and subsequent to our work, Sun has done exactly that. The New I/O [59] (NIO)

[1]This work was performed on Java 1.2 before the introduction of asynchronous I/O.

**R0:** Fatal environment and JVM implementation exceptions are not raised at all replicas.

**R1:** A thread must not invoke `java.lang.Thread.stop`.

**R2:** Native methods must produce only deterministic output to the environment.

**R3:** Native methods must invoke other methods deterministically.

**R4A:** All access to shared data is protected by a monitor (i.e., Java's `synchronized` keyword).

**R4B:** A thread has exclusive access to all shared variables while scheduled.

**R5:** All native method output to the environment is either *idempotent* or *testable*.

**R6:** If a native method produces volatile state in the environment, then a side effect handler is provided to recover the state.

Table 3.1: Restrictions placed on applications and execution environment.

package released by Sun implements asynchronous I/O, but still does not give rise to asynchronous commands; instead, NIO can be addressed using the techniques for non-deterministic commands discussed in the next section.

Asynchronous commands in the JVM correspond to asynchronous Java exceptions that are not interesting sources of non-determinism. All but one of these exceptions are raised by fatal errors in the run-time environment (e.g., resource exhaustion) or in the implementation of the JVM (e.g., locks in inconsistent states). Such errors are intrinsic to the run-time environment of the application and would repeat themselves if all replica environments were identical. Our implementation must not replicate these exceptions, or all replicas will deterministically fail. Replication is effective only if we assume that either such errors never occur or that the replicas' run-time environments are sufficiently different. We assume the latter in R0 in Table 3.1.

The stand-out non-fatal asynchronous exception is delivered to a thread when it is killed by another thread. However, beginning with the Java Development Kit version 1.2, use of this exception is deprecated. Applications that use this method might not work on future releases of the JVM and should be rewritten using condition variables. We therefore place restriction R1 in Table 3.1 upon applications prohibiting the use of the deprecated exception.

### 3.0.2 Non-deterministic Commands

A command is *non-deterministic* if its write-set values or its output to the environment are not uniquely determined by its read-set values. The only non-deterministic bytecode executed by the JVM invokes a *native method*. Java includes the Java Native Interface (JNI) [70] to invoke methods that execute platform-specific code written in languages other than Java. Native methods have direct access to the underlying operating system and other libraries. By accessing the operating system, for instance, native methods implement windowing components, I/O, and read the hardware clock.

Native methods therefore may take input values from the environment as well as from the read set. In conventional State Machine Replication, replicas run an agreement protocol to make their read sets and the input from the environment identical. It is generally impossible to have the BEEs agree on input values from the environment, since input is performed outside the control of the JVM. Instead, we make sure that differences in input values (e.g., different local clock values) do not result in different write-set values for the command. By simply forcing the backup to adopt the write-set values produced by the primary. However, since native methods execute beyond the purview of the JVM, an agreement protocol cannot ensure that replicas executing a native method will behave identically. We therefore restrict the behavior of native methods as given in R2 and R3 of Table 3.1 to achieve identical results at all replicas.

R2 restricts the native method behavior visible to the environment; however, it is often possible to relax this restriction and still obtain the same functionality provided by the offending method. For example, a method that reads the current time and then prints it could be split into two methods. The first method reads the local time and writes it to a local variable *lc*, which constitutes the method's write set. Our agreement protocol ensures that executing the first method at the primary and the backup results in the same value for *lc*. The second method, which prints the value of *lc*, now produces deterministic output to the environment.

R3 restricts the ways in which a native method invokes other methods. While executing outside of the state machine, a native method can invoke Java methods,

```
class Example {
  // Accessible from all threads.
  static Formatter shared_data = null;
  String toString() {
    // Guard not protected by monitor!
    if(null == shared_data) {
      shared_data = new Formatter();
      synchronized_method();
      // code continues...
```

Figure 3.1: A common data race in Java. If the Formatter constructor and synchronized_method are idempotent the data race has no semantic effect.

causing the BEE to execute commands. If a native method calls a Java method non-deterministically (e.g., if the native method decides to acquire a lock depending on the value of the local clock) then the sequence of commands processed by a BEE could be different at each replica. We rule out this possibility by forbidding native methods from making non-deterministic calls to Java methods.

We do not consider R3 a significant restriction, but rather a better programming paradigm: to avoid debugging nightmares, it is wise to restrict non-determinism in native methods to input methods. Just as R2, R3 might be upheld by splitting an offending method into a non-deterministic input method and a deterministic method. For instance, the clock example would be handled by placing the clock read in a different method and allowing our replicas to agree on the local clock values before invoking the (now deterministic) method that acquires a lock. Native methods must use the JNI interface to invoke other Java methods; thus, a program can be inspected for compliance with R3 by checking native methods that use the JNI interface.

### 3.0.3 Non-deterministic Read Sets

Shared memory among threads creates the possibility of deterministic commands reading different read-set values at different replicas of a given BEE. We call a read set *non-deterministic* if it contains at least one shared variable. Java allows data to be shared both explicitly, by invoking methods on a shared object, and implicitly, through static data references. We could keep track of all shared data or perform data race detection as in Eraser [98]. Generally the bookkeeping necessary to determine which objects are actually shared can result in a significant source of overhead: for example, an order of magnitude in time for Eraser.

Note that with C-MTFT, two fault-tolerant recovery units may not share variables so that read sets cannot overlap across recovery unit boundaries. The non-deterministic read-set values are solely determined by accesses from BEEs within a single recovery unit. We address data shared between recovery units using thread-fault tolerance in Chapter 4.

We explore two restrictions to make this problem manageable. One is to assume R4A in Table 3.1, which requires every access to a shared variable to be protected by a monitor (i.e., that the program is free of data races). Another way to achieve the same result is to assume R4B, which requires a run-time environment that enforces exclusive access to shared variables while a thread is scheduled (e.g., on a uniprocessor). Relaxing both restrictions for the general case might require a combination of the approaches above and agreement on the shared data values.

A Java monitor guarantees exclusive access to shared variables. In practice, the monitor allows the invoking BEE to transform temporarily a shared variable into a local variable. To a BEE that invokes a monitor and acquires its associated lock, however, the values stored in these temporary local variables appear to be non-deterministic since they have been last modified by some arbitrary BEE. One way to eliminate this non-determinism would be for the replicas to agree on the values of the variables associated with every lock they acquire. This approach is hard to implement, however, because Java does not express or enforce the association between a lock and the variables it protects, leaving this responsibility to the programmer through annotations or the use of statistical measures.

Our solution is instead to achieve agreement on the sequence of BEEs that acquire each lock. Reaching agreement on a lock acquisition sequence ensures that the corresponding BEEs at the primary and the backup access the variables associated with the lock in identical order. Combined with identical initial values, identical lock acquisition sequences guarantee all commands executed by corresponding BEEs have identical read-set values.

Unfortunately, many real programs do not satisfy R4A: even the JRE provided by Sun does not meet this restriction for all shared data. In particular, static data members are often shared between threads without explicit shared method invocations. As BEE replicas reach agreement on the sequence of lock acquisitions, these data races can cause the state of the primary and backup to diverge, even when the races do not affect the semantics of the program.

Figure 3.1 shows a use of static data members without acquiring a lock. Object `shared_data`, a static data member, is shared by all `Example` objects. The guard on line 4 is not protected by a monitor, which allows different thread schedules at the primary and the backup to invoke `synchronized_method` a different number of times, preventing agreement on the sequence of lock acquisitions. Testing our implementation of replicated lock acquisitions required removing these race conditions in the JRE by hand! Although the code in Figure 3.1 contains a data race, we wanted to find a less labor-intensive way to handle this common (mal)practice.

We also consider an approach for handling shared data that does not rely on R4A, but assumes R4B instead. It eliminates non-deterministic read sets by replicating at the backup the order in which threads are scheduled at the primary. When R4B holds on a uniprocessor, the BEE whose thread is being scheduled effectively changes *all* its shared variables to local variables because no other BEE is allowed to execute commands. By replicating the order in which threads are scheduled, our implementation ensures that the order of access to shared data is replicated regardless of whether data races exist.

### 3.0.4 Output to the Environment

State Machine Replication strives to hide replication from the environment by requiring output to the environment to be indistinguishable from what a single correct state machine would produce. To meet this requirement, we distinguish between output to the environment that affects *volatile state* (i.e., state that does not survive failure of the state machine) and *stable state* (i.e., state that does). A particular command can produce multiple outputs to the environment, each of which is either volatile or stable. We discuss each of the types in turn.

Hiding replication of output to stable state is easy if the output is either *idempotent* or *testable*. In the former case, the output is independent of the number of times the corresponding command is executed, while in the latter the environment can be tested to ascertain whether the output occurred prior to failure. For example, seeking to an absolute offset in a file is an idempotent operation, while seeking to a relative offset is not. If the current offset can be read, a relative seek becomes a testable operation. Except for these cases, it is impossible to maintain the "single correct machine" abstraction in the presence of failures. For instance, in a primary-backup system a backup cannot in general determine whether the primary failed before or after performing an output command, and executing the command again could produce different results. This impossibility result forces us to introduce a further restriction R5 in Table 3.1 that requires all native method output to the environment be either *idempotent* or *testable*.

Replication of output to volatile state might be necessary for correct operation. For example, the OS underneath the JVM is considered part of the environment. Opening a file at the primary creates OS state that disappears when the primary fails and that the backup must replicate if it is to execute correctly. Some volatile state could be restored simply by replaying the output (i.e., if the methods are idempotent), but volatile state generally requires special treatment. For instance, replaying messages on a socket would not recover the state at the backup because sending messages is in general not an idempotent operation. An extra layer must be added to make sending messages either an idempotent or testable operation as in [3].

Our protocol uses a novel interface, called *side effect handlers*, to replicate the lost volatile state of the primary. Native methods can create volatile state as an effect of producing output to the environment. Using JNI, any application may call native methods supplied by the application. Our interface allows an application programmer to include methods that replicate the volatile state of the primary created by the additional native methods. For example, through the interface we have included methods to handle file I/O in the standard JRE libraries. Restriction R6 in Table 3.1 requires applications to use this interface whenever they invoke a native method that creates volatile state.

### 3.0.5 A Note on Restrictions R0–R6

The restrictions that we impose on applications are manageable. R0, R2, and R5 must hold for any replicated application. R1 is for convenience as we could replicate the special case for backwards compatibility. We introduce R4A/B to provide a natural foundation to handle multithreading. R4B does not require any programmer effort, while R4A requires good programming practice. R5 is inevitable if we limit ourselves only to modifying the JVM because native methods execute outside of the JVM, but as discussed previously it is not difficult to follow. R6 allows us to work around the limited control over native methods, requiring some extra knowledge and programming effort.

Though we assume the restrictions are followed, we could easily enforce R1 and R6 at runtime. We could not enforce R6 directly as the JVM cannot distinguish which native method will output to the environment. Instead we would enforce a stronger restriction that holds for all native methods (instead of only those that output to the environment). Clearly R3 cannot be enforced automatically, but requires programmer assistance. The areas of code to inspect are well-defined: native methods that read input from the environment and invoke other methods through the JNI interface. R4A could be enforced using techniques as in Eraser [98]. The system could enforce R4B by adding an extra shared lock at the cost of reduced concurrency.

It is interesting to note other methods to work around our restrictions. We discuss approaches other than R4A or R4B for handling replication of multithreaded applications in Section 3.3. We could circumvent R6 using approaches outside of the JVM (e.g., TFT [16]). However, the JVM alone cannot identify native methods with output to volatile state. As such, we would need some form of annotation from the programmer.

## 3.1 Implementation

Sun's JVM 1.2 provides two implementations of multithreading. The *native threads* version provides thread scheduling in the underlying OS, while the *green threads* version implements a user-level thread library for a uniprocessor inside the JVM. Since R4A depends upon the application's use of locks and not the low-level thread implementation, both libraries can take advantage of techniques that achieve replica coordination by replicating the sequence of lock acquisitions. Indeed, multiprocessor applications running with native threads on an SMP can take immediate advantage of the technique described in Section 3.1.2.

Enforcing R4B, however, requires changes in the thread library. Since our first goal is to maximize portability, we have focused on implementing a replicated thread scheduler for green threads. Our approach could be extended to native threads (see [104])—we leave this as future work.

We add two system threads to the JVM. One performs failure detection to allow the backup to initiate recovery. The other is concerned with the transfer of logging information, either by sending it (at the primary) or by receiving it (at the backup). These additional threads join the several system threads that perform tasks such as garbage collection and finalizing objects. We next discuss how our implementation addresses the challenges (non-deterministic commands, non-deterministic read sets, and output to the environment) that we identified in Section 2.3.

### 3.1.1 Nondeterministic Commands

We checked by direct inspection and categorized all native methods in the standard libraries of the JRE: fewer that 100 native methods are non-deterministic. We store the *signature* of these methods, composed of their class name, method name, and argument types, in a hash table. Generally, every time a native method is invoked at the primary, its signature is checked against those stored in the hash table. If there is a match, then the method's return values (including arguments, if they are modified) and the exceptions raised are sent to the backup, which keeps an identical hash table. Before executing a method during recovery, the backup checks if it is stored in the hash table. If so, the backup always uses the corresponding return values and exceptions, whether or not it actually invokes the method. If the method is indeed invoked in order to reproduce volatile output, the backup discards the generated return values and exceptions. The side effect handlers discussed later provide an extra layer to handle specific cases where the return value may reflect volatile environment state (e.g., returning a file descriptor from a file open command).

### 3.1.2 Nondeterministic Read Sets

Data races and scheduling differences among the JVM's threads can make read sets containing shared variables return different values at the primary and the backup. We use two different approaches to make read sets deterministic.

**Replicated Lock Synchronization.** The first approach relies on assumption R4A: all shared data is protected by locks that, if correctly acquired and released, ensure mutual exclusion. Under this assumption, we create a mechanism that guarantees that threads acquire locks in the same order at the primary and at the backup.

Replicating the order in which threads acquire locks requires identifying the locking thread, the lock, and the relative order of each lock acquisition. We store this information in a *lock acquisition record*, which is a tuple of the form (t_id, t_asn, l_id, l_asn) where:

**t_id** is the *thread id* of the locking thread.

**t_asn** is the *thread acquire sequence number* recording the number of locks acquired so far by thread *t_id*.

**l_id** is the *lock id*.

**l_asn** is the *lock acquire sequence number* recording the number of times lock *l_id* has been acquired so far.

These records are created by the primary, but they are used during recovery by the backup. Therefore, for each thread and lock, the primary needs to generate virtual *t_id*s and *l_id*s that are unambiguous across replicas. For instance, although in the JVM each lock is uniquely associated with an object, the primary cannot simply use the object's address as the lock's *l_id*, because this address is meaningless at the backup. Further, any scheme that assigns ids according to the order in which events—such as thread and object creation—occur at the primary is dangerous, since these events might be scheduled differently at the primary and the backup.

We then define recursively the id of a thread $t$ as consisting of two values: 1) the id of the parent thread of $t$ (the parent of the first thread has by convention $t\_id = 0$) and 2) an integer that represents the relative order in which $t$ is created with respect to its siblings. This definition is well founded because, although the absolute order in which $t$ is created does depend on the order in which threads are scheduled, $t$'s parent spawns its descendants in the same relative order at the primary and the backup, independent of scheduling.

To assign a lock its *l_id*, we observe that threads execute deterministic programs. Hence, the sequence of locks acquired by a thread with a given virtual *t_id* is identical at the primary and the backup. We can then uniquely identify a lock by specifying the *t_id* and the *t_asn* of the first thread that acquires the lock at the primary. We get an even simpler *l_id* as follows. When the primary acquires a lock for the first time, it assigns to the lock a locally unique value (our *l_id* is simply a counter); it then creates an *id map*, which is a tuple of the form (*l_id, t_id, t_asn*) that associates the *l_id* with the appropriate *t_id* and *t_asn*. Each map is then logged at the backup.

During failure-free execution, whenever the primary acquires lock *l_id*, it generates a corresponding lock acquisition record and logs it at the backup. If the primary fails, then the backup's threads use the logged id maps and acquisition records to reproduce the sequence of lock acquisitions performed by the corresponding threads at the primary.

When a backup thread $t$ tries to acquire a lock with id $l$, it checks if the log contains a lock acquisition record with $t\_id = t$ and $l\_id = l$, and $t\_asn$ equal to the current value of $t$'s acquire sequence number. If such a record $r$ exists, then $t$ waits for its turn for acquiring lock $l$—that is, $t$ waits until $l$'s acquire sequence number is equal to the value of $l\_asn$ stored in $r$, acquires the lock, and removes $r$ from the log. If the log contains no such record, then $t$ waits until the log contains no more lock acquisition records (indicating the end of recovery at the backup) before it acquires lock $l$.

The case in which a backup thread $t$ attempts to acquire a lock that still has no *l_id* requires special treatment. First, $t$ checks if it is its responsibility to assign the id to the lock. The thread looks for an id map with $t\_id = t$ and matching $t\_asn$; a match implies that, before the primary failed, thread $t$ at the primary assigned to that lock the *l_id* stored in the id map. If a match is found, the corresponding map is removed from the log and the id of the lock is set to *l_id*.

If a match is not found, then either 1) the lock was assigned its *l_id* at the primary by a different thread $t'$, or 2) no primary thread logged an id map for the lock before the primary failed. Thread $t$ handles these two cases by waiting, respectively, until either $t'$ assigns the *l_id* at the backup or until the log contains no more maps, in which case $t$ can safely assign a new *l_id* to the lock.

This approach only replicates the lock acquisition sequence, which may require extra synchronization when ordering is important. If multiple threads are interacting with the environment (e.g., reading or writing a log) and the interleaved order is important, then synchronization is required to ensure an identical order between the primary and the backup even if the synchronization is not required for correctness at the primary.

**Replicated Thread Scheduling.** The second approach relies on assumption R4B: the scheduling lock protects all shared data. Whenever the primary interrupts the execution of a thread $t$ to schedule a new thread, it sends a *thread scheduling record* to the backup, which uses it during recovery to enforce the primary's schedule. A record is comprised of *(br_cnt, pc_off, mon_cnt, l_asn, t_id),* where:

**br_cnt** counts the control flow changes (e.g., branches, jumps, and method invocations) executed.

**pc_off** records the bytecode offset of the PC within the method currently executed by $t$.

**mon_cnt** counts the monitor acquisitions and releases performed by $t$.

**l_asn** records the lock acquisition sequence number when $t$ is rescheduled while waiting on a lock.

**t_id** is the thread id of the next scheduled thread.

The basic scheme for tracking how much Java code $t$ executed before being rescheduled is simple, and it is implemented by the first two entries in the schedule record. Rather than counting the number of bytecodes, which would add overhead to every instruction, we instrumented the JVM to increment *br_cnt* for each branch, jump, and method invocation. Further, since the program counter address is meaningless across replicas, we store in *pc_off* the last bytecode executed by $t$ as an offset within the last method executed by $t$. Unfortunately, in our implementation this requires an update to the thread object after executing every bytecode because it is hard to determine, when $t$ is rescheduled, where the JVM is storing its program counter, whose value is needed to calculate *pc_off.* However, storing the offset is still simpler updating a counter. Future implementations may by able to remove the *pc_off* member if it can be determined from the state of $t$ saved when $t$ is rescheduled.

A first complication over this simple scheme arises when $t$ is rescheduled while executing a native method. Native methods are opaque to the JVM: we have no way of determining precisely when $t$ is rescheduled. Often this is not a problem: when repeating $t$'s schedule during recovery, the backup reschedules $t$ right before the native method is invoked. This is unacceptable, however, if $t$, while executing within the native method, acquires one or more locks: reproducing the lock acquisition sequence is necessary for correct recovery, because it is this sequence that determines the value of shared variables. Fortunately, whenever a lock is acquired or released, control is transferred back inside the JVM. Our implementation intercepts all such events, independent of their origin, allowing us to correctly update the value stored in *mon_cnt*. In this case, instead of rescheduling $t$ during recovery before invoking the native method, we allow $t$ to execute within the native method until it performs the number of lock acquisitions stored by the primary in *mon_cnt*.

Further complications come from the interaction of application threads and system threads. System threads do not correspond to a BEE executing application code, and several do not execute Java code at all (e.g., the garbage collector). As was the case for native threads, we cannot reproduce scheduling events that involve system threads. [2] Ignoring system thread scheduling creates problems when application and system threads share resources, such as the heap, because both types of threads can contend for the same locks.

In particular, interaction with system threads might result in either of two events occurring during the recovery of an application thread $t$:

1. **$t$ is forced to wait at the backup for a lock that was acquired without contention at the primary.** In this case, $t$ runs the risk of being rescheduled by the backup before it can complete the sequence of instructions executed by its counterpart at the primary. We solve this problem by adding a separate *scheduler thread* and a private runnable queue (as in user-level thread libraries) to guarantee that $t$ will continue to be scheduled, without being interleaved with other application threads, until necessary.

2. **$t$ acquires without contention at the backup a lock for which it was forced to wait at the primary.** So, while $t$ was rescheduled at the primary, it might not be rescheduled at the backup. It is easy to use *mon_cnt* to enforce

[2]Replicating thread scheduling at the OS level in the native threads library would allow us to handle all threads, but at the cost of reduced portability. Further, we would still have to modify the JVM to handle other sources of non-determinism.

the correct scheduling.

Threads can also perform *wait* operations on a monitor, blocking the thread until a corresponding *notify* or *notifyAll* is performed. If multiple threads are awakened, we need to guarantee that they will acquire the monitor in the same order at the primary and the backup. To do so, we store the *l_asn* of the monitor lock as part of the thread scheduling record.

A final subtle point arises when the backup completes recovery, i.e. when it finishes processing the sequence of thread scheduling records logged by the primary before failing. The last scheduling record in this sequence contains the *t_id* $t'$ of the next thread that the primary intended to schedule—the primary failed before recording at the backup the scheduling record for $t'$. Nevertheless, the backup must schedule $t'$ because at the primary $t'$ might have interacted with the environment. $t'$ will execute at the backup until these interactions are reproduced.

### 3.1.3 Garbage Collection

Garbage collection in Sun's JVM is both asynchronous and synchronous. Any thread can synchronously collect garbage by invoking a JRE native method. Asynchronous garbage collection is performed periodically by a garbage collector thread and during memory allocation when memory pressure indicates collection is needed. Since garbage is unused memory by definition, we initially avoided replicating the behavior of the asynchronous collector thread. However, asynchronous garbage collection can be a source of non-deterministic read sets. Indeed, both *soft references* and *finalizer methods* create paths for non-deterministic input to application threads.

Soft references are used to implement caches. By fudging the definition of garbage, the referenced objects are guaranteed to be garbage collected before an out-of-memory error is returned to the application. Because R0 prevents such an error from being raised at all replicas, collection of soft references might occur at different times at different replicas. For instance, the primary might find an object in its cache, while the backup might not, leading the execution of primary and backup to diverge. [3] Although we could replicate the behavior of the asynchronous garbage collector by recording when it locks the heap, we use a much simpler solution: all

[3]Similar arguments also apply to *weak references* [19], which we treat similarly.

soft references are simply treated as strong references, which represent active objects and are therefore never collected. This shortcut has no effect on our experiments because there is never enough memory pressure to dictate the collection of soft references.

Another possible source of non-determinism is improper use of finalizer methods. These methods are intended to allow objects to reclaim resources that cannot be freed automatically by the garbage collector (e.g., if memory was allocated in a native method). The Java language specification states that finalizer methods are invoked on objects before the memory allocated to the object is reused, but does not specify exactly when, allowing different behaviors at the primary and the backup. Our current implementation assumes that finalizer methods only free unused memory or perform other deterministic actions on local memory. Since no data is shared between the thread that runs the finalizer on dead objects and any threads that previously used those objects, no new source of non-determinism is introduced. However, it is possible to write improper finalizer methods that do more than free unused memory: in fact, they can perform arbitrary actions, possibly with non-deterministic side effects. Although we don't currently replicate the invocation of finalizers, it would be easy to do so using one of the approaches discussed previously.

### 3.1.4 Environment Output

We deal with output commands in native method through a novel approach based on what we call *side effect handlers* (SE handlers). SE handlers are used to store and recover volatile state of the environment and to ensure exactly-once semantics for output commands. A handler consists of five separate methods that are called at various stages of execution at each replica.

***register*** This method registers with the JVM information about the native methods that the handler will manage, including the signature of the method, whether the method is a non-deterministic command and/or an output command, and whether its arguments should be logged (i.e., if they are also output arguments).

***test*** The backup calls this method to test during recovery whether an output command succeeded. For example, the first output command after recovery is terminated is *uncertain*—we cannot in general decide whether the command has completed. *test* is called on an uncertain command to determine whether a *testable* output completed before failure, guaranteeing exactly-once semantics. Commands for which the *test* method is not defined are considered idempotent and are simply replayed.

***log*** The primary calls this method after executing an output command. The system provides *log* with the arguments to the native method that performed the output (including the class instance object), the return value from the native method, and extra information about the internal state of the JVM. *log* saves and returns in a message all state necessary to recover the output of the command. For example, on a file write this message might store the file descriptor and the amount written (or the current file pointer offset).

***receive*** The backup calls this method to receive the state stored by the primary through the *log* method. Before saving the state, *receive* can compress it: for example, *receive* could compress the results of several file writes into one offset for the file pointer.

***restore*** The backup calls this method during recovery. It is invoked only once. *restore* recovers the volatile state affected by output commands. If *receive* has compressed the results of multiple commands, *restore* might be able to recover the appropriate state directly instead of replaying the commands. For example, to recover an open file *restore* would open the file and set the file pointer to the appropriate offset.

Each SE handler can manage a set of related native methods. For example, we have one handler for all native file I/O methods. The handlers we have written for the standard libraries are automatically added to the system during startup. Applications can incorporate their own handlers using the same functions. Using SE handlers allowed us to add support for file I/O in the standard libraries. The same approach can be used by application writers to incorporate user-supplied output commands.

## 3.2 Experiments

Our experimental setup consists of two Sun E5000 servers, each with 15 400MHz UltraSPARC II cpus and 2GB memory running SunOS 5.8 connected by a 100 Mbps Ethernet. The primary runs on one machine and logs events at the backup running on the other. On performing an output, the primary waits until the backup acknowledges having logged all events up to the output event. The backup keeps its log in volatile memory.

Sun's JVM did not include the source code for Just-In-Time (JIT) bytecode compilation for Java 1.2. JIT compilation dynamically converts methods from bytecodes into native machine code instructions. Without the source we cannot use JIT compilation because we cannot include our modifications to some bytecode executions (e.g., interception of native method invocations). Hence, all of our experiments are performed in *interpreted mode* (i.e., without JIT compilation). JIT compilation reduces the execution time of CPU intensive code but has little effect on communication, which is our primary source of overhead. Hence, although the overhead on a JVM using JIT compilation is hard to predict, we believe that probably it wouldn't change significantly except for compress, which is two times faster on Sun's HotSpot JVM with JIT compilation. The other benchmarks vary from 20% faster to 20% slower execution time, probably resulting in comparable changes to the overhead.

To estimate the costs of adding fault-tolerance to the JVM, we run the SPEC JVM98 benchmark on the replicated lock acquisition implementation, the replicated thread scheduling implementation, and the original Sun JVM. The programs in the benchmark vary widely in their characteristics. Compress is a CPU-intensive Lempel-Ziv compression application. Jack is a parser generator which is run on input to generate a parser for itself. Db contains a memory-resident database that is queried multiple times. Jess is an expert shell system that computes on a set of common puzzles with progressively larger rule sets. Mpegaudio decompresses MPEG-Layer 3 audio files. Mtrt is the only multithreaded application in the benchmark and consists of a ray-tracer rendering a scene of a dinosaur. Though mtrt is the only multithreaded application, several other applications (notably, db) contain much synchronized code. We did not include results for javac (included in SPEC JVM98) because we could not get the application to run on Sun's original JVM.

Table 3.2 summarizes the properties of the benchmark applications with

| Implementation | Event | jess | jack | compress | db | mpegaudio | mtrt |
|---|---|---|---|---|---|---|---|
| Both | Intercepted NM | 64088 | 631295 | 419 | 96011 | 10031 | 1473 |
|  | NM Output Commits | 763 | 34 | 102 | 703 | 10 | 133 |
| Replicated Lock Acquisition | Logged Messages | 4873592 | 12833046 | 2355 | 53492759 | 14717 | 701738 |
|  | Locks Acquired | 4809503 | 12201750 | 1935 | 53396747 | 4685 | 700264 |
|  | Objects Locked | 4515 | 505223 | 102 | 15612 | 21 | 161 |
|  | Largest $l\_asn$ | 1410798 | 746136 | 633 | 5286641 | 1955 | 34738 |
| Replicated Thread Scheduling | Logged Messages | 64089 | 631296 | 420 | 96012 | 10032 | 30638 |
|  | Avg. Reschedules | 0 | 0 | 0 | 0 | 0 | 29163 |

Table 3.2: Properties of benchmarks pertinent to our implementation.

respect to our implementation. Database queries in db result in the most lock acquisitions by far, while jack locks more unique objects. All applications have few intercepted native methods and even fewer output commits. The largest $l\_asn$ shows that the lock acquisitions are skewed—few locks are responsible for most acquisitions. The average number of reschedules in the last row shows that though many locks are acquired in all of the benchmarks, only mtrt actually requires them for multithreading.

We implemented replicated lock acquisition for both the green threads library supporting user-level threads on uniprocessors and the native threads library supporting multithreading on an SMP. We only implemented thread scheduling for green threads. We found the overheads exhibited by the two implementations of replicated lock acquisitions to be qualitatively similar. We thus only report results from our implementation using green threads. All experiments are performed on lightly loaded machines running in multi-user mode; experiments were repeated until 95% confidence intervals were within 1% of the mean.

Figure 3.2 shows the overall execution times of the benchmark applications using each of our replication approaches normalized to the corresponding times without any replication. The primary columns are the execution times of the primary logging events to the backup, while the backup columns give the times for the backup to replay events from the log. Although our implementation was not tuned aggressively (we only optimized some in the replicated thread scheduler), we observed under 100% overhead for most applications. Replicating lock acquisitions has an average of 140% overhead (skewed by db) for green threads, well above the replicated thread scheduling's 60% average.

The overhead for replicated lock acquisitions (Figure 3.3) ranges from 5% (mpegaudio) to 375% (db). The large overhead in db is a result of processing its more than 53 million lock acquisitions. In Figure 3.3, Communication Overhead rep-

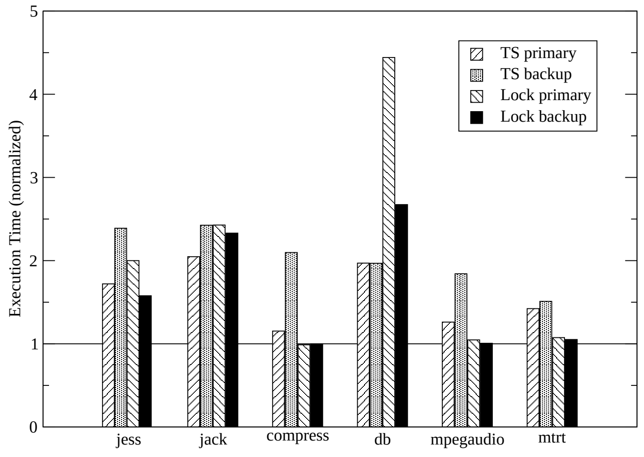


Figure 3.2: Comparison of our implementations using green threads normalized to our JVM without replication. The TS columns represent our replicated thread scheduler implementation, and the Lock columns represent the replicated lock acquisition implementation. The execution times of each benchmark are (in seconds): jess (167), jack (182), compress(541), db (354), mpegaudio (419), mtrt (163).

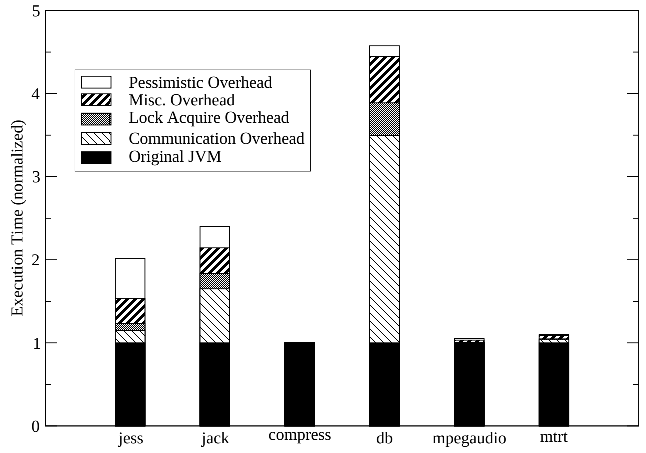


Figure 3.3: Normalized overhead for replicated lock acquisition implementation using green threads library.

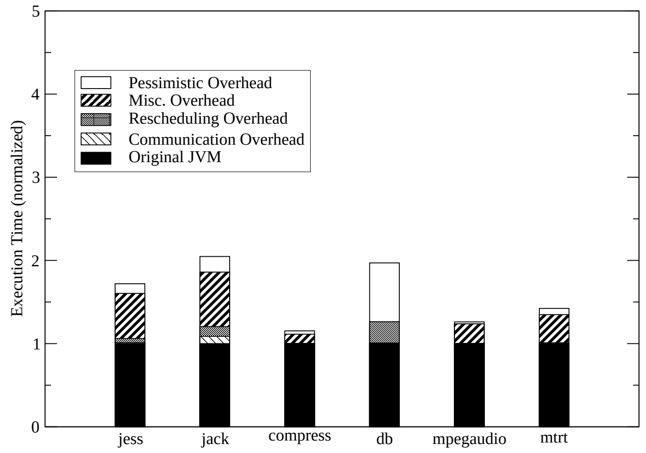


Figure 3.4: Normalized overhead for replicated thread scheduling implementation using green threads library.

resents the time spent sending messages to the backup, and Lock Acquire Overhead measures the time spent storing information on lock acquire. Pessimistic Overhead represents the time spent waiting for acknowledgments from the backup on output commit events.

In our implementation lock acquisition messages are very small (36 bytes). The primary buffers such messages and sends them to the backup either periodically or on an output commit; in the latter case, the primary sends the buffered messages and waits for an acknowledgment. Similarly, the backup only sends an acknowledgment message after processing a burst of incoming logging messages.

The sources of overhead for the replicated thread scheduling implementation are detailed in Figure 3.4. Communication Overhead and Pessimistic Overhead are as in Figure 3.3, while Rescheduling Overhead measures time spent updating counters and storing scheduling decisions. The overhead varies from 100% (jack) to 15% (compress).

Replicating thread scheduling yields a lower communication overhead than replicating lock acquisition: only mtrt logs any thread schedule records to the backup. Further, to reduce the number of records created, a record is sent only when a new thread is scheduled. All other benchmarks are single-threaded; hence, they do not involve transmission of any records. The replicated lock acquisition implementation does not take advantage of this single-threaded case, sending many unnecessary messages.

For such applications, we expect replicated thread scheduling to incur smaller overhead than replicated lock acquisition. In practice, however, we observe that this is not always the case (see Figure 3.2), because storing thread progress incurs significant overhead. As seen in Figure 3.4, the overhead of replicated thread scheduling is dominated by the Misc. Overhead, which captures the overhead resulting from extra bookkeeping. In an earlier version of our implementation, the bookkeeping overhead for the replicated thread scheduler overwhelmed any communication advantages. To reduce these costs, we were forced to add about 12 instructions that update counters and keep track of the virtual machine's PC to the hand-written optimized assembly loop that executes bytecodes at the heart of the JVM. We believe significant additional reductions could be achieved by optimizing the code further. Also, using a deterministic scheduler as in the Jikes RVM [33, 55] or Jalapeño [22] might result in lower overhead substantially because the progress indicators would be simplified.

The two approaches to handling multithreading present different tradeoffs. Replicating lock acquisitions may be less effective if a thread acquires or releases objects several times before being rescheduled. Further, replicating thread scheduling handles automatically the single-threaded case as no extra messages are sent. Nonetheless, replicating lock acquisitions is still a compelling approach because it works on multiprocessor systems, and may provide better performance, as in the case of mtrt.

As communication overhead is the dominant source of overhead in our experiments, the amount of communication for a given application created by each technique is an effective predictor of their performance.

## 3.3 Related Work: JVM

Replica coordination [63, 99] can be implemented at any level of a system's architecture, from the application level [12] all the way down to the hardware [7]. Systems that implement replica coordination at intermediate levels include TFT [16] (at the interface above the operating system) and Hypervisor-based fault tolerance [17], in which replica coordination is implemented above a *virtual machine* that exports the same instruction set architecture as HP's PA-RISC.

We first reported on our fault-tolerant JVM in [84]. Since then, we have become aware of other concurrent and independent efforts that address some of the same issues discussed in this paper. Basile and others report on replicating multithreaded applications in [9]. They develop a leader-follower replicated lock acquisition algorithm that assumes R4A and a Byzantine failure model for a web-server application. Their algorithm for replicated lock acquisition is similar to ours; however, they do not explore scenarios where R4A doesn't hold.

Friedman and Kama have also explored the idea of modifying the JVM (in their case, the Jikes RVM [55]) to achieve transparent fault-tolerance [33] using semi-active replication. Although we share the same goals, our approaches differ in three fundamental ways. First, their approach only applies to systems where R4A holds, while we explore multiple ways to handle the non-determinism introduced by multithreading. Second, they do not address applications with non-deterministic native methods, though they do address I/O within the JRE. Finally, they report experiments using JIT, while all our experiments are performed in interpreted mode because we require access to the source code for JIT.

Prior work on debugging multithreaded applications addressed non-determinism. LeBlanc and Mellor-Crummey first introduced recording lock synchronization and shared memory accesses for debugging replay [66]. More recently, Choi and Srinivasan apply this approach to Java in the DejaVu tool for debugging assuming R4A in [24] and R4B in [22]. DejaVu records *logical thread intervals* wherein a thread performs non-deterministic events such as monitor entry/exit and shared variable accesses. The intervals include thread schedules for the underlying deterministic thread scheduler of the Jalpeño JVM.

As our focus is fault-tolerance, our implementation differs in several ways. First, we include a general approach to handling application-provided native methods. Second, DejaVu does not address output to the environment. Third, the Jalapeño scheduler reschedules at deterministic *yield points*, simplifying thread execution progress tracking.

Their trace sizes are much smaller than ours by clever use of intervals, but the overhead incurred is still 40%-80%, comparable to ours without pessimism. Our implementation could benefit from the use of intervals. For the multithreaded Mtrt application there would only be 56 intervals instead of 700258 lock acquisitions—four orders of magnitude fewer events, resulting in a significant saving in space and probably also time.

To the best of our knowledge, replicating lock acquisitions for handling multithreading was first proposed by Goldberg, et al., for Mach applications in [34]. When replicating lock acquisitions, correctness depends on the absence of data races. By augmenting the type system, Boyapati and Rinard developed race-free Java programs which meet R4A [15]. Data race detection mechanisms [21, 98] could also be used to verify R4A holds for a given program.

Our implementation of replicated thread scheduling is based on Slye and Elnozahy [104]. They record thread progress during normal execution using a count of control flow changes (branches, jumps, function calls). Our solution differs in two ways: 1) the JVM cannot track all control flow changes (for example, while executing a native method) and 2) we do not recover all threads (for example, the garbage collector).

## 3.4 Summary

We built a process-fault tolerant JVM using the State Machine Approach. We implement and evaluate two techniques for eliminating the non-determinism introduced by multithreading in C-MTFT. The first technique allows the threads at the backup to reproduce the exact sequence of monitor acquisitions performed by the threads at the primary. The second technique replicates at the backup the thread scheduling decisions performed at the primary. Our results suggest that this is a viable solution for providing process-fault tolerance to Java applications.

Highlights to take away from this chapter:

- We identified two sources of non-determinism in the JVM, including non-deterministic commands from native methods and non-deterministic read sets from multithreaded access to shared data.

- Access to shared data (that is, non-deterministic read sets) can be determined either by lock acquisition order [34] or by thread scheduling order.

- Native methods can be replicated using a framework of side-effect handlers that characterize the non-determinism of each native method for purposes of recovery.

# Chapter 4

# Optimistic MT Fault-Tolerance

The previous chapter describes techniques to improve the reliability of multithreaded applications, but these improvements require all threads to be in the effective recovery unit. In this chapter, we advocate a new approach for optimistic MT fault-tolerance (O-MTFT) that increases availability by reducing the size of the effective recovery unit. The key insight of this approach is to manage access to shared memory to reduce the likelihood of erroneous writes by identifying them early. Combined with the rarity of errant writes, managing shared memory using consistency checks limits the erroneous execution of a faulty thread, enabling the application to use the minimal effective recovery unit. However, errant writes may still occur in the Capricious model so that the complete approach 1) uses O-MTFT to optimistically produce a small effective recovery unit (Chapters 5–6), 2) detects if the effective recovery unit is too small to complete recovery (Section 4.2), and 3) triggers C-MTFT if necessary in order to finish recovery (Section 6.5).

Before we address techniques to manage shared memory for O-MTFT, we discuss the feasibility of reducing the effective recovery set size in multithreaded applications. We first consider evidence of errant writes between threads because these writes are by their nature difficult to detect (leading us to fall back on the conservative approach discussed in the previous chapter). In our Capricious model, errant writes may occur between threads, but the frequency of such writes can be determined experimentally rather than arbitrarily often (as would occur in the Byzantine model).

We refer to field studies of software faults in existing systems to judge the likelihood and impact of errant writes. The Tandem studies [37, 67] evaluate the source of software faults in the Tandem OS itself and how the faults are tolerated. These studies show that errant writes to memory due to software failures are fairly rare, and erroneous writes typically do not cause further errors. Of interest is the categorization of *first errors*, which are the first effects of a fault. Though 23% of first errors resulted in bad addresses, only 7% could be classified as errant writes unrelated to the data structure under operation. Further, only 18% of all first errors propagated to cause further errors, implying corruption is typically contained by the operating system. Half of the first errors studied were detected by observing references to illegal addresses; otherwise, consistency checks performed by the OS itself detected the errors.

These results suggest that for many applications errant writes are rare and that consistency checks can effectively prevent most erroneous and errant writes. As we will show in later chapters, the O-MTFT technique imposes little overhead to the application so that the improvements in availability provided by the reduced effective recovery unit can be favorably weighed against the unlikely probability of errant writes between threads. Indeed, some consistency checks may even be automatically generated, similar to the method for generating program invariants in Daikon [31]. Since most processors do not guarantee memory protection between threads, our O-MTFT technique is only optimistic and may underestimate the size of the effective recovery unit. If an errant write affects another thread, it will be necessary to fall back on the C-MTFT techniques discussed in the previous chapter to ensure correct recovery. In the next section, we discuss related work that can further reduce the likelihood of errant writes to increase the applicability of O-MTFT.

## 4.1 Related Work: Software Data Isolation

We focus on techniques for protecting memory regions from errant writes, which we call *data isolation*, that are implemented in software and do not require special hardware support. All of the following techniques can reduce the possibility of errant writes either by extending the hardware memory protection mechanisms available on general-purpose commodity processors or by sandboxing execution. Reducing the likelihood of errant writes increases the chance that O-MTFT techniques will be able to use a smaller effective recovery unit than the C-MTFT techniques discussed previously.

Sullivan and Stonebraker study several techniques to protect a database manager's buffer pools [109]. *Guarding* shared pages marks them read-only until an update operation is executed. This technique incurs only a few percent overhead for update-intensive workloads. Write-protecting pages improves the consistency of memory after failures even though other objects on the same page remain vulnerable during an update because memory access is protected only at the page granularity.

The Rio project resulted in several papers related to software data isolation [20, 74, 86]. Rio provides persistent memory that survives operating system crashes using virtual memory protection. Although Rio relies on special hardware to survive power outages, the memory protection techniques do not require special hardware. Initially, Rio was used as a replacement for the operating system's file cache [20]. A memory segment is kept read-only until it is updated in a manner similar to the guarding technique of Sullivan and Stonebraker. A special API marks the page writeable before update and read-only afterwards. Further, the operating system is modified to ensure the buffer pages are protected during initialization. Without memory protection from writes (that is, with the file cache always writeable), corruption is rare (1.5%) but higher than with a simple write-through to disk system (1.1%). With write protection, Rio achieves improvements in reliability (0.6%), while performance is significantly better than a write-through system (4–22 times as fast as UFS).

Ng and Chen show that using Rio to hold database buffer caches provides similar reliability to using disk for persistent storage, while providing much faster access times [86]. In the experiments, as previously with the file cache, buffer cache pages are kept read-only until a transaction locks an object contained in the page for update. However, fault injection experiments show that protecting memory does not significantly increase reliability: errors due to memory corruption decreased from 2.7% to 2.5% of errors. The authors attribute the limited benefit of protection to the fact that few writes intentionally go to the buffer cache and errant writes are unlikely to hit randomly the buffer cache in the 64-bit address space used.

The Recovery Box [6] uses a kernel memory segment to backup important OS data in order to protect that data during failure and recovery, but provides no access protection from errors that occur in the kernel.

Wahbe et al. [112] designed *fault domains* to contain the erroneous and errant writes of the executable code that comprises a domain from causing memory faults in other domains. Fault domains are implemented in software by sandboxing code to ensure that it cannot access memory outside of its fault domain. Executable code is scanned (largely statically) to verify this property before execution. Similarly, the Java Runtime Environment (JRE) provides security policies to limit loading of trusted code and access to different objects. These approaches can limit the corruption caused by errant writes due to bugs in software, but static checks cannot prevent code mutated by SEU errors from affecting memory. It remains necessary to check for consistency of shared data during execution to minimize the damage that may occur from errant writes.

## 4.2 Related Work: Detecting Faults

While consistency checks can find corruption of shared data, other techniques can detect errors before data is corrupted, reducing the length of erroneous execution and the likelihood of Capricious orphans. Memory protection hardware can detect some errant writes to bad locations. Datarace detection [23, 29, 88, 98, 117] can detect unprotected accesses to shared memory that may be the result of errant writes or software bugs.

Several projects attempt to correlate invariants with particular failures in an effort to help fix software bugs. We can instead apply these approaches to predict imminent failures by assuming anomalous behavior is an early indicator of a failure. Depending on the false positive rate, these techniques can be used to reduce erroneous execution by triggering O-MTFT recovery before a failure is observed by memory protection hardware or other consistency checks. Further, if O-MTFT recovery has already been attempted, these techniques can signal a failure of the optimistic approach and trigger fall back on C-MTFT as discussed in the next section.

Using sampling of different runs, Liblit et al. use statistical bug isolation [71] to find predicates that correlate with failures. DIDUCE [44] dynamically develops invariants to detect anomalous program behavior to assist developers in finding the root cause of a failure. The Multivariate State Estimation Technique (MSET) used in [39] can be used reliably to predict failure conditions due to overload using domain-specific telemetry such as the average time to commit a transaction in an OLTP system. Other approaches [14, 43, 73, 93] use machine learning techniques to automatically generate a statistical model of the application to classify program behavior using such metrics as counting control flow or basic block execution counts.

Finally, the replication used in C-MTFT can also be used to detect faults as replicas diverge. For example, when replicating thread scheduling, threads at two replicas might acquire locks with different ids or at different points of execution (as measured by control flow changes) signaling a divergent execution due to an error.

## 4.3 Reducing the Effective Recovery Unit

Although errant writes are rare, we must still address erroneous writes to shared memory both to reduce the number of Capricious orphans and to identify the orphans. In many MT applications, synchronization primitives (for example, mutexes and semaphores [27]) maintain the consistency of shared data by forcing threads to wait for atomic access to shared data. Failures exacerbate the problem of waiting as some threads may be forced to wait indefinitely for a faulty thread that will never release the lock. Further, the shared data protected by a lock can be left in an inconsistent state when the thread holding the lock fails. Transactions [36] have long been used in fault-tolerance to provide safety guarantees in the presence of failures. Under concurrent access, a transaction provides atomic access to data that can be *committed* or *aborted* so that all updates are visible to other threads or not, respectively. Consistency checks can be performed on the transaction to reduce the likelihood of committing erroneous values. Further, the transactional system can track the propagation of erroneous values. The precise view of memory provided by a transactional system—that is, the collection of safety properties that it guarantees—depends on the particular correctness condition used. We discuss two such correctness conditions, serializability [91] and linearizability [53], in the next chapter when we present our own solution to managing shared data for O-MTFT.

While transactions and consistency checks ensure the safety of shared memory in the presence of failed threads, correct threads still need access to shared memory in order to make progress. Without providing progress guarantees to the threads outside the effective recovery unit, O-MTFT could only reduce the recovery time, but if the threads outside the effective recovery unit can make progress, O-MTFT can increase availability *during* recovery. *Software transactional memory* (STM) [101] combines a transactional interface to shared memory with the robustness of nonblocking data structures [49, 51]. There are several nonblocking progress guarantees: *obstruction-freedom* guarantees progress only when a single thread attempts to commit a transaction (admitting livelock); *lock-freedom* guarantees some thread will eventually be able to commit a transaction (admitting starvation, but not livelock); and, *wait-freedom* guarantees every correct thread eventually commits a transaction. Although all of these progress guarantees tolerate halted threads, we restrict our use of STM in O-MTFT to lock-free and wait-free STM because we will also use the STM to tolerate hung threads (see Chapter 6), which can occur in the case of livelock and deadlock. In the following chapter, we present an STM that meets our criteria to manage shared memory between fault-tolerant components: it provides both a lock-free progress guarantee and ensures a consistent serializable view of memory for all threads. Our STM also maintains a history of changes to shared memory that can be used during recovery.

## 4.4 Summary

Our optimistic MT fault-tolerance approach provides for independent thread recovery. Unlike C-MTFT, O-MTFT applications can utilize a minimal effective recovery unit containing as few as a single thread, but can fall back on C-MTFT techniques to guarantee consistency as we will discuss in Chapter 6.

Highlights to take away from this chapter include:

- O-MTFT differs from C-MTFT in that the effective recovery unit may contain fewer threads (possibly only one) because O-MTFT limits erroneous writes using consistency checks on shared memory.

- Errant writes overwrite data not accessed by a thread during fault-free execution, but occur rarely and can be mitigated by consistency checks during access to shared memory.

- Management of shared data for O-MTFT must guarantee consistency—provide a reasonable view of memory in the presence of concurrent access and thread failures—and ensure progress—provide access guarantees to correct threads.

Both provisions enable threads outside the effective recovery unit to make progress during recovery.

# Chapter 5

# Software Transactional Memory

*Software transactional memory* (STM) combines transactions with the robustness of nonblocking data structures [49, 51, 78, 101]. An STM frees users from concerns about the atomicity of multi-object operations, and in turn its implementation is freed from the disadvantages of using locks for concurrency control: deadlock, priority inversion, difficult fault management, and the undesirable tradeoff between complexity and achievable concurrency. Recently, STM using mutual exclusion locks have been designed that are guaranteed to be deadlock-free, but they tradeoff nonblocking access for performance gains [30]. While concurrent programmers are primarily concerned with deadlock [30], this thesis addresses the fault-tolerance of multithreaded applications. We focus on non-blocking STM to demonstrate their use in O-MTFT.

While most transactional systems use serializability [91] as their correctness criterion for handling concurrent transactions, most STM implementations [32, 46, 52, 96] provide linearizability [53]. Extending linearizability to transactions with multi-object operations requires these transactions to appear to take effect instantaneously, or in "one-at-a-time" order respecting the real-time order in which the operations occur [52]. Transactions must then operate simultaneously on the most recently written versions of all objects to ensure a consistent view of objects under concurrent modification. To ensure this in nonblocking STM, transactions acquire exclusive ownership (albeit using no physical locks) of all objects they operate upon, limiting concurrency.

To increase the opportunity for concurrent execution of transactions, we demonstrate our stm, SSTM, that provides serializable, rather than linearizable, executions. An earlier version can be found in [83]. Serializability does not impose a real-time ordering on committed transactions; thus, transactions need not acquire ownership of all objects simultaneously. For example, a long-running read-only transaction may be serializable using previously written versions, which is not possible in a linearizable system where the real-time ordering of events requires operations to access current values of data, forcing any update transactions to wait for the long-running transaction.

Our SSTM has many of the desirable properties of modern STM. First, *lock-freedom*, which guarantees that as long as at least one correct thread continuously attempts to commit transactions, an infinite sequence of transactions will commit: in other words, livelock or deadlock can never occur even in an environment with crash failures, although starvation is still possible for a single thread. Second, support for *dynamic transactions*, which do not need to know in advance the set of objects they will access. Third, *isolation*, where transactions are guaranteed to see serializable data, eliminating the spurious errors generated by some STM [47]. Fourth, *a single read and write object interface for all transactions*: read-only transactions need no special treatment. Fifth, *disjoint-access parallelism* [57], which ensures transactions on disjoint sets of objects do not compete. Finally, *increased concurrency* through the support of multiversioned objects: specifically, SSTM enforces *one-copy serializability* [11]. We use abstract operations encoded in blind writes to exploit the more liberal consistency constraints on blind writes under serializability over those under linearizability. Blind writes reduce the frequency of transactions aborted due to consistency conflicts with other transactions. In the rest of this document, we describe our algorithm, the implementation, and performance evaluation. In the performance evaluation we contrast Fraser's OSTM [32] and a mutual-exclusion lock-based STM (both implementing linearizability) with our timestamped STM that implements serializability. Using a mix of different transactions on a concurrent object, we demonstrate the advantages and disadvantages of our approach.

## 5.1 Related Work: STM

Shavit and Touitou propose the first example of nonblocking software transactional memory [101]. Their STM is lock-free, but requires static transactions: a transaction must declare before executing the set of objects on which it will operate. Building on their work, Herlihy, et al., present the first dynamic software transactional memory (DSTM) [52]. DSTM however relaxes the progress guarantee: instead of lock-freedom, it offers *obstruction freedom* [51], which guarantees progress only in the absence of contention. Fraser [32] combines the best characteristics of these earlier works by proposing an object-based STM that provides dynamic, lock-free transactions [32]. Many other STM explore different tradeoffs: access or commit time acquisition of objects (ASTM) [77], different validation techniques (RSTM) [105], and contention management for conflicting transactions (SXM) [41, 115].

Our STM provides the same properties as Fraser's (it is object-based and supports dynamic lock-free transactions) but departs from nearly all existing STM in choosing serializability, rather that linearizability, as its correctness criterion.

Lock-free transactional objects have been extensively studied in real-time environments [4, 92] where threads have priorities that constrain preemption, simplifying shared data synchronization. Our work assumes a different computational model wherein threads do not have priorities and no bounds are placed on the relative speed of threads.

*Transactional monitors* [113] have been proposed as an alternative to traditional monitors in Java. While Java monitors regulate access to shared data using mutual exclusion, transactional monitors do so using lightweight transactions. Transactional monitors provide serializability, but do not address nonblocking access—for example, a thread is not allowed to halt inside a monitor. We instead guarantee serializability while ensuring nonblocking access to shared data—in our STM, threads may halt arbitrarily without preventing the progress of other threads.

Conditional critical regions (CCR) [54] are an elegant approach to manage shared data. Recent work [45, 46] explores implementing nonblocking access to shared data using an obstruction-free, multi-word compare-and-swap primitive, while providing programmers with the familiar CCR interface. Since this interface is independent of the mechanism used to regulate access to shared data, we believe our approach could be used underneath a CCR interface.

Finally, our system is similar to optimistic concurrency control [62] as it does not use locks to regulate which transactions should be executing concurrently, but rather aborts transactions to enforce serializability. However, while in optimistic concurrency control locks are used to enforce serializability at commit time,

```
atomic procedure CAS ( addr, expected_value, new_value ) returns $\mathcal{V}$
   let old_value := value_of(addr)
   // Store new value only if old values matches expected
   if ( old_value = expected_value )
      value_of(addr) := new_value
   return old_value
```

Figure 5.1: Pseudocode implementing Compare-And-Swap operation. The set $\mathcal{V}$ encompasses the possible values of a variable that is modifiable using CAS (typically both 32-bit and 64-bit CAS are supported).

our system guarantees serializability without locks while providing strong progress guarantees. Further, our optimistic system guarantees serializable reads during execution so that errors are not introduced by the use of the STM.

## 5.2 System Model

We assume only a finite number of threads exist in the system at any time, though new threads may be created at any time. In this chapter, we aggregate the shared data associated with read and write sets into objects that support read and write operations. As discussed in Section 2.1, threads may run simultaneously (as on a multiprocessor) and be arbitrarily delayed (for example, by a page fault). We also assume no bounds on the relative speed between threads. In contrast to the Capricious presented in Chapter 2, in this chapter we restrict threads to fail by halting; a thread that does not halt is considered correct. Chapter 6 addresses the full Capricious failure model.

We assume that shared data is modified by threads only through the STM and that the Compare-And-Swap (CAS) primitive is available to operate on shared data—CAS can be implemented on all major modern processor architectures. CAS atomically and conditionally modifies a shared memory location (as described in Figure 5.1) and has been shown by Herlihy [49] to be sufficient to achieve nonblocking synchronization. We assume that a correct thread may complete a CAS operation in a finite number of steps, though the CAS may fail if the expected value is not present.

A transaction is an ordered sequence of read and/or write operations on a set of objects that may be determined dynamically. Read and write operations on different objects may be interleaved within a transaction. Once all operations are complete, a thread either attempts to *commit* or *abort* the transaction so that all operations are either visible or not, respectively, to other transactions. We do not address nested transactions (that is, a single thread does not interleave or nest operations of different transactions); hence, only operations belonging to transactions performed by different threads may be interleaved. We assume any transaction executed in isolation transfers the system from a correct state to another correct state, and require committed transactions to be *view serializable* [91], meaning that there exists a sequential execution of the set of committed transactions (that may have in reality executed concurrently) such that each operation returns the same value in both executions. Transactions are uniquely identified. A thread $p$ executes the transaction $T_i$ from the set $\mathcal{T}$ of all possible transactions on the set $\mathcal{O}$ of all possible objects. Until a transaction commits or aborts, it is considered undecided (denoted $T_i \in \mathcal{U}$); whereas a decided transaction is either *aborted* ($T_i \in \mathcal{A}$) or *committed* ($T_i \in \mathcal{C}$).

We take special care in allowing read-only transactions to proceed concurrently with writes. Borrowing from multiversion databases, we keep several versions of an object so that reads can proceed in parallel as new versions are written. Maintaining multiple versions allows reads of past versions, permitting read-only transactions to commit concurrently with read/write transactions. A version is uniquely associated with the transaction that creates it, and to distinguish between versions, the corresponding version is noted in the operation. For example, $T_k$ creates version $x_k$ of object $x$ with the write $w_k[x_k]$. A read operation is then denoted $r_j[x_k]$ if $T_j$ reads version $x_k$. There is a total order, called the *history*, to versions of an object such that for all versions $x_j$ and $x_k$, either $x_j \ll x_k$ or $x_k \ll x_j$ where $\ll$ represents the history order. All writes to a specific object within a transaction are considered to be a single write operation. Each write operation on an object could generate a complete copy of the object. Alternatively, in Section 5.4 we explore an implementation where each update holds the abstract operation to be performed on the object or the portion of the object that has changed, requiring a read to perform the operations in the history to generate the relevant version of the object.

In a multiversion system, serializability alone is insufficient to enforce executions consistent with the one-copy semantics assumed by many multithreaded applications. We thus require transactions to be *one-copy serializable* (1-SR), that is, serializable with the added constraint that a read always sees the latest write in a serialize ordering. A *multiversion serialization graph* (MVSG) [11] provides the framework to determine whether a set of concurrent transactions over multiple versions of objects can be equivalent to a set of sequential transactions over a single copy of each object. The MVSG is composed of vertices $T_0, \ldots, T_n$ from $\mathcal{T}$ and edges $T_i \rightarrow T_j$ (where $i \neq j$) if $r_j[x_i]$ is in $T_j$ for some object $x$ previously written by $T_i$. Such *reads-from* edges ensure that a version is written before it can be read. To ensure that the version read corresponds to the latest write, the MVSG contains two other types of edges—for each $r_k[x_j]$ and $w_i[x_i]$ operation ($T_k \neq T_i$), we add a *write-read* edge $T_i \rightarrow T_j$ if $x_i \ll x_j$, and a *read-write* edge $T_k \rightarrow T_i$ if $x_j \ll x_i$. In the MVSG($\mathcal{T}, \ll$) we thus have either $T_i \rightarrow T_j \rightarrow T_k$ or $T_j \rightarrow T_k \rightarrow T_i$ for the operations $r_k[x_j]$ and $w_i[x_i]$, ensuring that no other write operation (in this case, $w_i[x_i]$) is ordered between a write operation ($w_j[x_j]$) and the corresponding read operation ($r_k[x_j]$). In [11], a set of transactions $\mathcal{T}$ is shown to be 1-SR if and only if there exists a history order $\ll$ such that MVSG($\mathcal{T}, \ll$) is acyclic.

To simplify discussion, we shorten MVSG($\mathcal{C}, \ll$) to CMVSG, and we use PMVSG for the MVSG($\mathcal{T}, \ll$) where $\mathcal{T}$ contains all *public* transactions (either committed or undecided, but with all operations enqueued). We say that $T_i \rightsquigarrow T_j$ if there is a path from $T_i$ to $T_j$ in the corresponding MVSG. To achieve 1-SR executions, before committing an undecided transaction $T_i$ our algorithm checks for cycles in the PMVSG containing $T_i$ and aborts transactions to ensure that the CMVSG will be acyclic upon commit of $T_i$.

### 5.2.1 Serializability vs. Linearizability

Here we present an example illustrating the difference between the serializability and linearizability correctness criteria. Linearizability was originally specified as a local correctness criterion for concurrent access to single objects [53]. Extending the condition to transactions that span multiple objects requires concurrent ownership of all objects. A linearizable system of multi-object transactions does not allow stale data whereas a multiversion serializable transactional system allows read-only summary transactions—those that may read many objects, but write none—to read stale data. Figure 5.2 demonstrates a set of transactions that is not linearizable, but is serializable and applies to read-only summary transactions across many objects. Since any linear order is also a serial order, serializability allows more executions, and

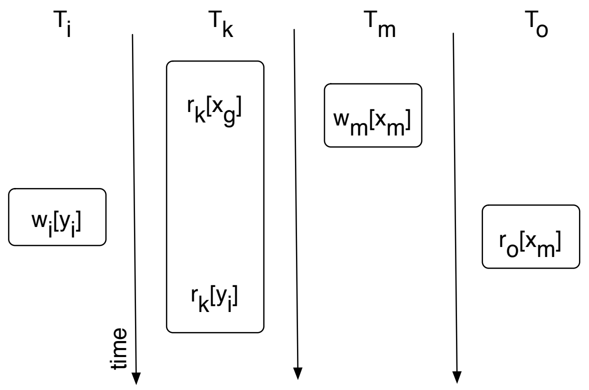


Figure 5.2: Linearizability vs. Serializability. In this example time flows down, and $T_g \in \mathcal{C}$ before these transactions begin. In any linearizable execution, if $T_i, T_m, T_o \in \mathcal{C}$, then $T_k \in \mathcal{A}$ since there is no point in time time at which both $x_g$ and $y_i$ are valid. A valid serialization of the transactions is $T_i, T_k, T_m, T_o$, as shown.

can thus be said to allow more concurrency. However, implementing a serializable system that exploits this extra concurrency is difficult; we show later in this chapter the performance of our serializable STM versus current linearizable STM.

## 5.3 Lock-Free Transactions

Figure 5.3 shows the data structures used by our algorithm. Our STM's API, shown in Figure 5.4, provides methods to read, write, and create objects as well as to commit, abort, and validate transactions.[1] Only the thread that initiates a transaction can invoke on it the methods in the API; however, the helper functions called by some of these methods (shown in Figure 5.6) can also be called by other threads—we will discuss the helper functions and how they are invoked shortly.

A thread $p$ begins a transaction by calling `begin-transaction`, which returns a data structure representing the transaction. This data structure is private to $p$ until $p$ calls `commit-transaction`: we call the transaction *private* until then. If $p$ calls `abort-transaction`, the data structure never becomes shared, and the transaction is aborted privately. As a transaction executes, this data structure records the versions read and written by the transaction and its status.

[1]Validation returns a boolean indicating whether the transaction will be required to abort because of conflicting transactions.

```
// Global timestamp of transactions enqueued
int trans_tstamp

// Transaction data
transaction T_i
   int ts  // Timestamp of enqueue
   transaction status  // 2 LSB indicate status
   obj_version * reads  // List of versions read
   obj_version * writes  // Cache of writes

// Data object
object x
   obj_version * history  // Sequence of versions

// Version of object written by transaction
obj_version x_k
   obj_version * next  // Next version in history
   trans_list * read_set  // Trans.s reading x_k
   transaction * creator // Trans. writing x_k
   void * data  // Object representation
```

Figure 5.3: Overview of data structures used by TS-STM.

To read an object $o$, $p$ calls `read-object`, which searches the history of $o$ for the latest version that is consistent with the previous operations of the transaction. Our algorithm checks at read time that the version being read maintains 1-SR. The procedure `is-readable` checks whether the current PMVSG would remain acyclic if $p$'s transaction reading the given version were allowed to be published. Since objects are modified concurrently, that version could later become inconsistent with the other operations. If a cycle is detected in the corresponding MVSG at commit time, transactions in the cycle are aborted as needed to ensure acyclicity. Each version stores in its *reading transaction set* the identifiers of the transactions that have read that particular version—the identifier of a transaction is added to the appropriate reading transaction sets when the transaction attempts to commit.

To perform an update, $p$ calls `write-object`, which keeps all writes local until commit to reduce cache contention and support fast aborts. During the commit procedure new versions are added to objects' histories to generate the history order of versions. To create an object, $p$ calls `create-object` then `write-object` to create the initial version.

A thread attempts to commit a transaction $T$ by invoking `commit-transaction`. In Step 1, for each read operations performed by $T$, the algorithm attempts to add $T$ to the reading transaction set of the version being read. Our implementation of the reading transaction set uses an ordered lock-free list [48], ensuring that $T$ is added no more than once for every read operation and that the loop in Step 1 eventually terminates successfully. Step 2 adds the write operations in $T$ to the

```
// Record version written in private transaction data.
procedure write-object ( $T_i$ , $x_i$ )
    $x_i$.read_set := $\emptyset$
    $x_i$.creator := $T_i$
    < Add $w_i[x_i]$ to $T_i$.writes >

// Read latest version of object that doesn't create a
// cycle in PMVSG and record version in transaction.
procedure read-object ( $T_i$ , $x$ ) returns $x_k \in x.\text{history}$
    // Loop backwards through history for latest version.
    foreach $\{x_k : x_k \in x.\text{history} : T_k \in \mathcal{C}\}$
        if ( is-readable($x_k$) )
            < Add $r_i[x_k]$ to $T_i$.reads >
            return $x_k$

// Create new object.
procedure create-object ( ) returns $x \in \mathcal{O}$
    < Create and init. object data structure $x$ >
    return $x$

// Mark object for deletion if transaction commits.
procedure destroy-object ( )
    < Mark object for deletion >

// Abort transaction by marking status. This trans's
// operations will not be visible to any other trans.
procedure abort-transaction ( $T_i$ )
    CAS(&($T_i$.status), undecided, aborted | mark-aborter($\perp$))

// Assign a unique transaction number and initialize transaction data.
procedure begin-transaction ( ) returns $T \in \mathcal{T}$
    < Create and init. transaction data structure $T_i$ >
    $T$.published := false
    $T_i$.ts := $\perp$
    return $T_i$

// Check whether trans. is part of a cycle in PMVSG.
procedure validate-transaction ( $T_i$ ) returns boolean
    return $T_i \not\rightsquigarrow T_i$

// Use helper function to commit transaction.
procedure commit-transaction ( $T_i$ )
    // Step 1. Enqueue read operations
    foreach $\{x : x \in \mathcal{O} : r_i[x_k] \in T_i\}$
        let $L$ := $w_k[x_k]$.read_list
        do
            list-insert-operation($L$, $T_i$)
        until ( $T_i \in L$ )
        // Step 2. Enqueue write operations
        let $S$ := $< T_i >$ // LIFO queue containing $T_i$
        while ( $S$ not empty )
            let $T_j$ := top of $S$
            let $T_q$ := help-enqueue-transaction($T_j$)
            if ( $T_q \neq T_j$ )
                push($S$, $T_q$)
            if ( $\exists T_k \mid T_k \in S : T_k$.published )
                < pop stack until $T_k$ removed >
                // Step 3. Attempt to commit published transaction.
                help-commit-transaction($T_k$)
```

Figure 5.4: API procedures to read and write versions of objects and commit, abort, and validate transactions. The procedure `list-insert-operation` adds the operation to the ordered list ([48] can be used provided each operation can be added only once). The `enqueue` procedure must return the element enqueued successfully or the argument if the element is already enqueued ([80] can be modified for this purpose). Finally, the &() operator returns the address of the argument.

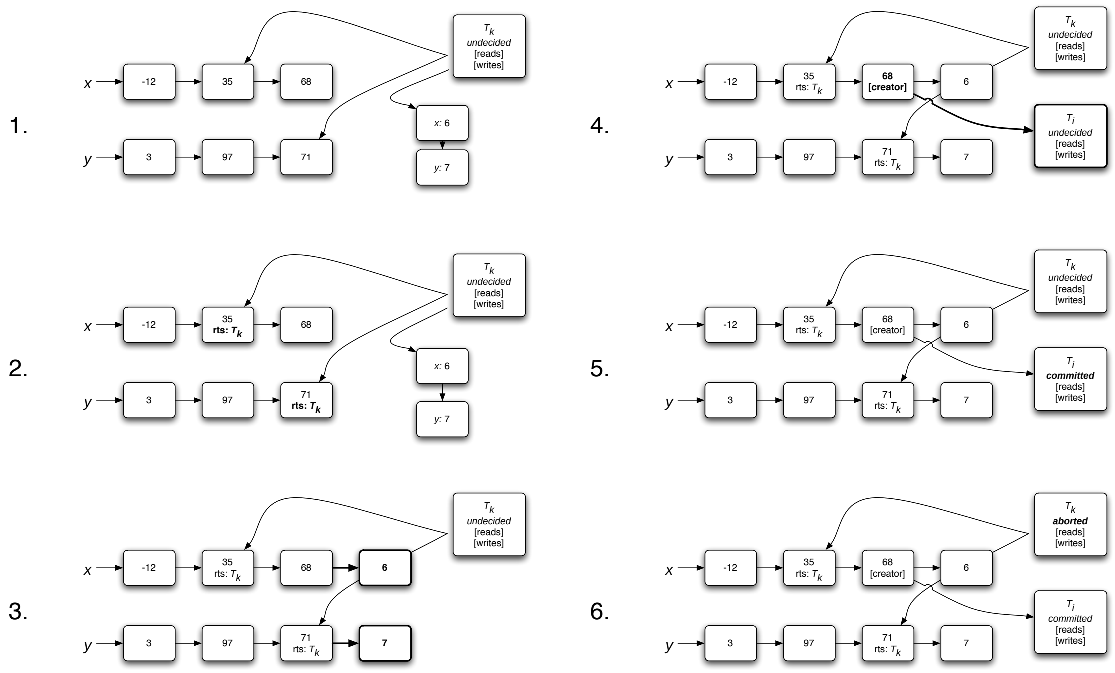


Figure 5.5: The lifetime of a transaction. 1. Transaction $T_k$ after execution has read and written both $x$ and $y$. 2. $T_k$ adds read operations to the relevant versions' reading transaction sets (rts) This step is guaranteed to complete eventually because the set is implemented as an ordered list where eventually $T_k$ can be added without contention. 3. $T_k$ enqueues write operations to the relevant histories. This step requires help from other threads in the presence of contention to guarantee that some transaction will be published successfully. After 3, $T_k$ has been published and can be committed or aborted by any thread. 4. Transaction $T_k$ finds a write by $T_i$ enqueued that would create a cycle when it examines the PMVSG. 5. If $T_i$ is earlier, the thread commits $T_i$. 6. Finally, $T_k$ must be aborted to prevent the cycle since $T_i$ has already been committed.

```
// Help commit transactions begun by any thread.
procedure help-commit-transaction ( $T_i$ )
    // Step 1. Ensure there are no cycles with $T_i$ in CMVSG.
    if ( $T_i \in \mathcal{U}$ )
        decide-mvsg-reachable($T_i$)
    // Step 2. Ensure there are no cycles from write-read edges.
    foreach $\{T_j : T_j \in \mathcal{C} : r_i[x_j] \in T_i\}$
        if( ¬ is-readable($x_j$, $T_i$.ts) )
            CAS(&($T_i$.status), undecided, aborted | mark-aborter($T_i$))
            return
    // Step 3. Commit with Compare-and-Swap status.
    CAS(&($T_i$.status), undecided, committed)
    // Step 4. Help aborter, as needed.
    if ( $T_i \in \mathcal{A}$ )
        let $T_j$ := get-aborter($T_i$)
        // Recurse only if new transaction
        if ( $T_j \neq T_i$ )
            help-commit-transaction($T_j$)

// Help undecided transactions begun by any thread.
procedure help-decide-transaction ( $T_i$, $T_h$ )
    // Help commit only earlier transactions.
    if ( $T_i$.ts < $T_h$.ts )
        help-commit-transaction($T_i$)
    else
        CAS(&($T_i$.status), undecided, aborted | mark-aborter($T_h$))

// Help decide earlier, reachable transactions in PMVG.
procedure decide-mvsg-reachable ( $T_h$ )
   foreach $\{T_i : T_i \in \mathcal{T} : (T_h \rightsquigarrow T_i) \wedge (T_i.\mathrm{ts} \leq T_h.\mathrm{ts})\}$
      if ( $T_i \in \mathcal{U}$ )
         help-decide-transaction($T_i$, $T_h$)

// Help transaction to enqueue operations
procedure help-enqueue-transaction ( $T_m$ ) returns $T \in \mathcal{T}$
    foreach $\{x : x \in \mathcal{O} : w_i[x_i] \in T_i\}$
        let $Q$ := $x$.history
        // enqueue returns transaction corresponding to successful enqueue.
        let $T_q$ := enqueue($Q$, $w_i[x_i]$)
        if ( $T_q \neq T_i$ )
            return $T_q$
    $T_i$.published := true
    CAS(&($T_i$.ts), ⊥, trans_tstamp)
    inc-timestamp( )
    return $T_m$

// Get finite set of transactions that have read version $x_i$
procedure get-readers ( $x_i$, ts ) returns $\mathcal{T}$
    return $\{T_j : T_j \in \mathcal{T} : (r_j[x_i] \in T_j) \wedge (T_j.\mathrm{ts} \leq \mathbf{ts})\}$

// Return aligned pointer to transaction.
procedure mark-aborter ( $T_h$ ) returns &( $T \in \mathcal{T}$ )
    return < aligned &($T_h$) >

// Return transaction pointer masked from status field.
procedure get-aborter ( $T_i$ ) returns $T \in \mathcal{T}$
    return < $T_i$.status with cleared 2 least significant bits >

// Determine whether reading version creates a cycle.
procedure is-readable ( $x_m$, $T_i$ ) returns boolean
    return $T_m \not\rightsquigarrow T_m \in \mathrm{CMVSG} \cup \{T_i\}$

// Increment global transaction timestamp for TPO.
procedure inc-timestamp ( )
    let ts := trans_tstamp
    CAS(&(trans_tstamp), ts, ts+1)
```

Figure 5.6: Procedures to help commit transactions. Aborted transactions are implicitly ignored. The procedures `decide-mvsg-reachable` and `is-readable` are simplified for presentation.

history of the corresponding objects to determine version order. At this point, we say that the transaction is *published*, and as such can be aborted or committed by any thread.[2] The loop in Step 2 uses a lock-free queue to hold object histories: the stack records threads competing to update object histories, and the size of the stack is limited by the finite number of threads in the system. Every time a correct thread invokes `help-enqueue-transaction`, an update is successfully enqueued, and updates can be enqueued only once—this ensures that eventually all updates of *some* transaction will be enqueued. Finally, in Step 3 the thread invokes `help-commit-transaction` on the public transaction at the top of the stack.[3]

The procedure `help-commit-transaction` in Figure 5.6 checks for cycles in the current PMVSG in Steps 1 and 2, and in Step 3 attempts to commit $T$ using CAS to change the transaction's status field if the observed PMVSG is acyclic. If the commit of $T$ fails, Step 4 of the procedure recursively helps to commit any transaction responsible for aborting $T$. No special API is needed for committing read-only transactions: their only distinction is that they can do away with Step 2 of `commit-transaction`.

We say that a transaction $T_i$ is *committed* if the CAS of the status of $T_i$ to *committed* succeeds. Conversely, $T_i$ is aborted if the CAS of the status of $T_i$ to *aborted* succeeds. To allow transactions to abort while guaranteeing progress, we save the id of the transaction responsible for the abort (using the `mark-aborter` function of Figure 5.6) in the status field of the aborted transaction. If a transaction $T_i$ initiated by thread $p_i$ is aborted by another thread $p_j$ attempting to commit $T_j$, $p_i$ can identify $p_j$ by using `get-aborter` (see Figure 5.6) and can help $p_j$ commit $T_j$. Helping ensures progress even if new threads perpetually abort $p_i$'s transactions and then promptly fail before ever successfully committing their own transactions.

To ensure that marking the aborted transaction occurs atomically with the successful abort, we use only the two least significant bits of the status field to show the committed, aborted, or undecided status.[4] If the transaction is aborted, the remaining bits represent an aligned pointer to the transaction responsible for the

[2]There is no guarantee that a given transaction will become public. The lock-free progress property only requires a sequence of transactions to become public.

[3]Because we implement object histories using lock-free rather than wait-free [49] queues, the transaction at the top of the stack may not be the one that the thread has finished executing.

[4]If we may reasonably assume that 0 is an invalid pointer and that pointers are aligned, we can use a single bit. If the status is 0, the transaction is undecided; 1 implies committed; any other value represents the pointer to the aborting transaction.

abort.[5]

Our STM is consistent with current approaches to garbage collection for objects, versions, or transaction data structures [50, 79]. A prefix of the object history can be pruned provided the history contains a committed version for future reads. Pruning risks causing slow transactions to abort because a suitable version cannot be found to read. If all operations of a transaction have been garbage collected, the transaction itself can be collected. Objects can only be garbage-collected after a committed transaction has invoked `destroy-object`.

Because threads may help commit transactions that they have not initiated, different threads may end up performing identical operations on the data structures that represent object histories, e.g. adding the same object version multiple times to an object's history. Fortunately, it is easy to modify existing data structures, such as ordered lists [48] or FIFO-queues [80], to return success if the update has already been added to the list or return an error if, because of garbage collection, it cannot be determined whether the update has been added to the list.

### 5.3.1 Ensuring One-Copy Serializability

Threads help commit or abort public transactions not only to guarantee progress, but also to ensure one-copy serializability. We prevent cycles in the CMVSG despite concurrent modifications of object histories in two steps. First, whenever $p$ calls `commit-transaction`$(T_i)$, we let the undecided $T_i$ add its operations to the object histories, making them visible to other threads—this results in new edges being added to the PMVSG. Second, we only attempt to commit an undecided transaction after we can be sure that the CMVSG that would result from committing $T_i$ is acyclic. The procedure responsible for enforcing this check is `decide-mvsg-reach-able`, shown in Figure 5.6 and invoked in Step 1 of `help-commit-transaction`.

Intuitively, `decide-mvsg-reachable` identifies all transactions that are reachable from $T_i$ along a path of committed transactions in the PMVSG. If $T_i$ is reachable from itself, then it should be aborted. Checking for the presence of such a cycle, however, presents a challenge. It is not sufficient to check for the absence of a cycle involving $T_i$ in the current CMVSG. That check would cover only transactions that are already committed, while the cycle may involve transactions that are going to

[5]We are similar in this to others who have used the least significant bits of pointers to mark, for example, the logical deletion of a member from a list [48].

commit concurrently with $T_i$—when we perform the check, the cycle may not yet have appeared in the CMVSG. On the other hand, it is not necessary to require that the PMVSG contain no cycles involving $T_i$ as a precondition for committing $T_i$. If such a cycle is detected in the PMVSG, the cycle may include undecided transactions other than $T_i$—to prevent a cycle in the CMVSG it is sufficient to abort any one of *those* transactions.

The approach we use in `decide-mvsg-reachable` is to consider, for each object $o$ that $T_i$ reads or writes, the transactions reachable from $T_i$ in the PMVSG by following edges that correspond to operations on $o$. The procedure `decide-mvsg-reachable` performs a breadth-first search of the transactions with earlier timestamps that are transitively reachable from $T_i$. By searching earlier transactions, the procedure is guaranteed to terminate because timestamps are assigned with increasing value. When an undecided transaction $T_U$ is found, the transaction is decided by attempting either to commit or to abort $T_U$ (we are going to discuss the appropriate course of action in a moment). If in so doing we return to $T_i$ on a path that includes only committed transactions, then to guarantee an acyclic CMVSG, we abort $T_i$.

We prove below that the set of transactions committed is 1-SR using the history order as the version order $\ll$. As a shorthand in the proofs, we use `hct` as an abbreviation of `help-commit-transaction`, `hdt` for `help-decide-transaction`, and `dmr` instead of `decide-mvsg-reachable`.

**Theorem 1.** *The set of committed transactions is one-serializable.*

*Proof.* The theorem holds if and only if the CMVSG is acyclic [11]. Suppose, by way of contradiction, that a cycle existed in the CMVSG, and consider the set $\mathcal{T}_{cy}$ of all transactions that appear in the cycle and those that *define* edges in the cycle. Note that this set may be larger than just the set of transactions that *appear* as vertices in the cycle: for instance, a transaction $T_{rd}$ that executes a read operation $r_{rd}[x_j]$ defines, if $x_i \ll x_j$, a write-read edge $T_i \rightarrow T_j$ even though $T_{rd}$ is not an vertex corresponding to the edge.

Assume that each committed transaction has a timestamp, that timestamps are totally ordered,[6] and that $T_l$ is the transaction in $\mathcal{T}_{cy}$ with the highest timestamp.

[6]The timestamps are assigned by `get-timestamp` of Figure 5.6. The total order is discussed in Section 5.3.2.

To establish the contradiction, we show that $T_l$ does not appear as a vertex in the cycle, nor creates a write-read edge.

$T_l$ **is not a vertex:** Consider the time at which a thread $p$ invokes `dmr`$(T_l, T_l.ts)$ from within `hct`$(T_l)$ in the invocation that leads to $T_l$ being committed. By this time, all transactions that will appear as vertices in the CMVSG have already enqueued their operations, because transactions are assigned timestamps only after all of their operations are enqueued, and $T_l$ has the latest timestamp. Therefore, when `dmr`$(T_l, T_l.ts)$ is invoked, the PMVSG already contains a cycle with $T_l$ as a vertex. Since $T_l \rightsquigarrow T_l$ and $T_l$ is not committed until `dmr` returns, $p$ executes `hdt`$(T_l,\ T_l)$, and, since $T_l$.ts $\not< T_l$.ts, $p$ executes the CAS (to abort $T_l$) at the end of `hdt`. Since $T_l$ is still undecided, the CAS will succeed and abort $T_l$. Hence, assuming a cycle of committed transactions including $T_l$ implies that $T_l$ is aborted—a contradiction.

$T_l$ **does not create a write-read edge:** Suppose for contradiction that a read $r_l[x_j]$ by $T_l$ defines a write-read edge $T_i \rightarrow T_j$ in the cycle in the CMVSG. Again, consider the execution of `hct`$(T_l, T.ts)$ by the thread $p$ that successfully commits $T_l$ in Step 3 of `hct`. In Step 2, for each version read by $T_l$, $p$ invokes `is-readable` on the version; we focus on the invocation `is-readable(` $x_j$, $T_l.ts$ `)`. The invocation finds the cycle $T_j \rightsquigarrow T_j$, which includes the edge $T_i \rightarrow T_j$, because all operations causing the cycle have already become public by the time $T_l$ is assigned its timestamp. Thus, `is-readable` returns false, and the thread executes the CAS following. By construction $T_l$ is not yet decided—indeed, not yet committed—until Step 3 of `hct`. Hence, the CAS will succeed, aborting $T_l$ and providing the contradiction. □

### 5.3.2 Ensuring Lock-Freedom

We can now go back to the problem of deciding what the thread $p$ that initiated transaction $T_i$ should do when, as it visits the transactions reachable from $T_i$ in the PMVSG, reaches another undecided transaction, $T_j$. It is tempting, in the interest of ensuring progress, to always have $p$ try to commit $T_j$. Unfortunately, helping to commit *all* undecided transactions reachable in the PMVSG does not work. Suppose two transactions $T_i$ and $T_j$ by threads $p_i$ and $p_j$ respectively, both add updates to the histories of objects $x$ and $y$ in the opposite order—without loss of generality, consider $x_j \ll x_i$ and $y_i \ll y_j$. Thread $p_i$ may then help commit $T_j$ as $T_i \rightarrow T_j$ from object $y$, while $p_j$ helps to commit $T_i$ because $T_j \rightarrow T_i$ from object $x$. Clearly, one of the transactions must abort to guarantee no cycles in the CMVSG.

To choose in general which transaction to abort, we introduce a total order on undecided transactions, called the *transaction priority order* (TPO).[7] We say that $T_i < T_j$ (or that $T_i$ has higher priority than $T_j$) if and only if $T_i$ precedes $T_j$ in the TPO; further, we define the *distance* between $T_i$ and $T_j$ as the number of transactions $T_k$ such that $T_i < T_k < T_j$. We require two properties of our total order: first, it must have a minimum; and second, the distance between any two transactions must be finite. A simple way to achieve both properties is to order transactions according to an increasing timestamp at commit as shown in Figure 5.4. Using a single CAS to increment a counter suffices—though multiple transactions by different threads may receive identical counters, ties can be broken using the id of the thread that initiates the transaction.

Note that the TPO is not used to determine the order in which transactions will eventually commit—transactions with higher priority may well commit after transactions with lower priority. Rather, the TPO is used to determine the fate of any undecided transactions $T_j$ that a thread $p$ encounters as it checks the PMVSG for cycles involving the transaction $T_i$, that $p$ is trying to commit. Specifically, $p$ will attempt to commit only those $T_j$ that have higher priority than $T_i$ and attempt to abort the rest. The two properties of the TPO ensure that any recursion triggered while applying this policy will terminate.

We prove that our algorithm is lock-free by showing that all threads agree to help commit the minimum transaction (according to TPO) in any given set of conflicting transactions. Since no thread will abort the transaction, eventually some transaction will commit, providing a lock-free progress guarantee.

**Lemma 2.1.** *If there is at least one correct thread, then there exists an infinite sequence of decided transactions $\mathcal{T}^p$ such that for each transaction $T_i \in \mathcal{T}^p$, some thread executes* `hct`$(T_i)$.

*Proof.* We consider the sequence of transactions $\mathcal{T}^p$ such that a single correct thread $p$ invokes `hct`$(T_i)$ for each transaction $T_i$ in the sequence. The assumption that $p$ is correct implies that $p$ will always attempt a new transaction by calling `commit-transaction`. The execution of `commit-transaction`$(T_i)$ executes `hct`$(T_j)$ on a

[7]A similar order is required in other STM. For example, in FSTM [32] the address of objects is used as a total order to prevent the cycle described.

sequence of transactions until `commit-transaction` terminates, implying the sequence $\mathcal{T}^p$ is infinite as long as $\texttt{hct}(T_j)$ terminates. We note that every invocation of `hct` terminates because all loops are over finite sets and recursion only occurs on earlier transactions in timestamp order, which has a minimum initial timestamp by construction. Finally, since $p$ is correct, each invocation of `hct` ensures that the corresponding transaction is decided in either Step 2 or 3 where the CAS either succeeds in aborting or committing the transaction, respectively, or fails in either case because the transaction is already decided. □

Note that the lemma refers only to *decided* transactions—lock freedom requires us to show that within $\mathcal{T}^p$ there exists an infinite subsequence of committed transaction. Towards this goal, we define two subsets over $\mathcal{T}^p$: 1) $\texttt{helped}(T_i) \equiv \{T_h \mid \texttt{hct}(T_h)$ was recursively called during the execution of $\texttt{hct}(T_i)\}$ and 2) $\texttt{benefactors}(T_i) \equiv \{T_p \mid T_i \in \texttt{helped}(T_p)\}$. Informally, $\texttt{helped}(T_i)$ includes the transactions that $T_i$ helped towards a decision, while $\texttt{benefactors}(T_i)$ includes the transactions that $T_i$ was helped by.

**Lemma 2.2.** *For any decided transaction $T_i$, $\texttt{benefactors}(T_i)$ is finite.*

*Proof.* The proof easily follows from the two observations that there exists a finite number of threads in the system before $T_i$ is decided and that only undecided transactions can be helped. □

Next, we introduce a method to map each transaction in $\mathcal{T}^p$ to a finite number of committed transactions in $\mathcal{T}^p$ using the following lemmas.

**Lemma 2.3.** *Consider an execution of $\texttt{hct}(T_p)$ that terminates, and let $T_i$ be the earliest transaction in $\texttt{helped}(T_p)$ according to the transaction priority order. If $T_i$ is aborted, then there exists a committed transaction $T_j$ such that $T_i \rightsquigarrow T_j$ in the PMVSG, and $T_j$ is committed after $T_i$ begins.*

*Proof.* We first note that $T_i$ cannot be aborted as a result of some thread executing $\texttt{hdt}(T_i, T_k)$ because the Lemma assumes $T_i$ is the earliest transaction in $\texttt{helped}(T_p)$. If an earlier transaction aborts $T_i$, it will also appear in $\texttt{helped}(T_i)$ by Step 4 of `hct`. Therefore, $T_i$ must have been aborted in Step 2 of $\texttt{hct}(T_i)$. The abort occurs if there is a cycle in the graph CMVSG $\cup$ $\{T_i\}$. The graph CMVSG is acyclic by

```
1 procedure find-committed-transaction( $T_p$ ) returns $T \in \mathcal{T}$
2     let $T_b$ := < earliest trans. in helped($T_p$) >
3     if ( $T_b \in \mathcal{C}$ )
4         return $T_b$
5     else
6         return $T_c \in \mathcal{C}$ : ( $T_b \rightsquigarrow T_c$ ) $\wedge$ ( $T_c$ is committed after $T_b$ begins )
```

Figure 5.7: The definition of `fct` that converts an infinite sequence of attempted (and decided) transactions to an infinite sequence of committed transactions.

Theorem 1 implying the cycle must contain $T_i$. As the graph is irreflexive, there must also exist a committed transaction $T_j$ in the cycle.

As argued above, $T_i$ must have been uncommitted after $T_p$ began in real-time. Further, consider the read that results in the cycle (containing $T_j$) for which $T_i$ is aborted. The `read-object` method searches for cycles in the same manner. Since the cycle did not exist at the time $T_i$ performed the `read-object` procedure, some transaction along the path must have committed since $T_i$ began. Let that transaction be $T_j$, completing the proof. □

We are now ready to prove that our algorithm is lock-free.

**Theorem 2.** *There exists an infinite sequence of committed transactions $\mathcal{T}^c$ provided there is at least one correct thread.*

*Proof.* By Lemma 2.1, there exists an infinite sequence $\mathcal{T}^p$ of decided transactions such that for each transaction $T_i$ in the sequence, `hct`$(T_i)$ is executed by some correct thread. We generate from $\mathcal{T}^p$ a (possibly disjoint) sequence of committed transactions $\mathcal{T}^c$ by applying to $\mathcal{T}^p$ the procedure `find-committed-transaction` (abbreviated `fct`) of Figure 5.7. By Lemma 2.3 `fct` is well-defined—that is, for any $T_p$ either $T_i$ (line 1) or $T_j$ (line 5) exists.

A transaction $T_i$ in $\mathcal{T}^p$ is mapped to $T_s$ in $\mathcal{T}^c$ at either line 3 or line 5 of `fct`. A transaction that maps to $T_s \in \mathcal{T}^c$ because of line 3 must belong to `benefactors`$(T_s)$—by Lemma 2.2, `benefactors`$(T_s)$ is finite. Hence the set of possible transactions $T_i$ that can map to $T_s$ is bounded.

A transaction $T_i$ that maps to $T_s \in \mathcal{T}^c$ because of line 5 implies the existence of a $T_b$ such that $T_i \in$ `benefactors`$(T_s)$, $T_b \rightsquigarrow T_s$, $T_s \in \mathcal{C}$, and $T_s$ is committed after $T_b$ begins. Since the transaction $T_b$ must begin before $T_s$ is committed, only a finite number of possible transactions $T_b$ can lead to a particular committed transaction $T_s$.

Further, the number of possible benefactors $T_i$ of $T_b$ is bounded as argued previously. Since only a finite number of transactions in $\mathcal{T}^p$ can map to any transaction in $\mathcal{T}^c$, the sequence $\mathcal{T}^c$ is also infinite. $\square$

## 5.4 Evaluation

Our testbed consists of a Sun E5000 server with 15 400MHz UltraSPARC II cpus and 2GB memory running SunOS 5.8. We tested our serializable STM (SSTM) against the lock-free FSTM [32] and an STM implemented using mutual exclusion locks. The test application (from the FSTM implementation) operates on an integer set stored as a red-black tree implemented in each different STM. Each element in the tree is maintained by a block of memory provided by the STM, and no garbage collection is performed. Elements in the set are found using a lookup in the tree within a read-only transaction, while insert and delete operations require update transactions. Every transaction in our experiments performs a single insert, delete, or lookup operation on the set, but may require multiple reads and/or writes to different transactional memory blocks. SSTM-$n$ is a version of SSTM where complete insert or delete operations are encoded in an update to the root of the tree—here, $n$ represents the maximum number of abstract operations we allow before we coalesce these logical updates and create a new, updated copy of the tree. This bound prevents the tree from becoming flattened into a linear linked-list of set operations.

Figure 5.8 shows the average time to commit a single transaction (insert, delete or lookup on the set) given different numbers of threads under all STM on a tree with 256 keys and 20% read-only transactions. Though our STM implementation is not optimized, the experiments indicate that our STM pays a significant performance price during contention. Figure 5.9 demonstrates the number of aborted conflicting transactions during the attempt to commit 100,000 transactions each by varying numbers of threads. This figure evaluates our SSTM-B$n$ with a modified red-black tree that does not require reading previous values of the memory blocks before updating, allowing blind writes to encode $n$ abstract operations before a comprehensive update. No transactions are aborted unless there is conflict with a concurrent transaction that may violate transactional consistency.

The figures show the tradeoff to encoding abstract operations: when $n$ is large the data structure representing the set degenerates into a linked list and performance

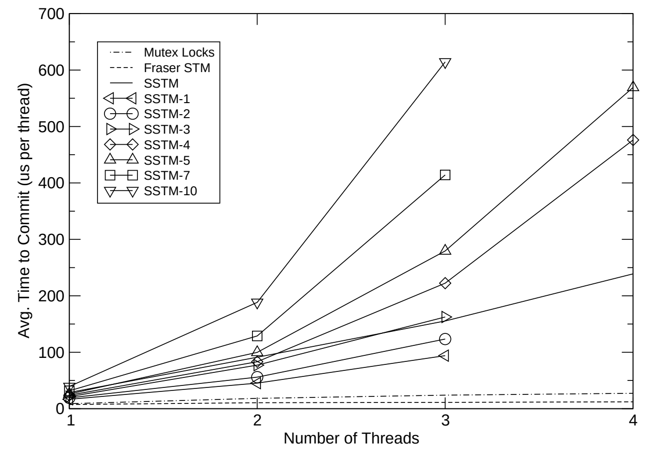


Figure 5.8: Average time to commit a transaction (per thread) with a mix of 20% read-only (lookup) transactions and 40% each of update (insert and delete) transactions.

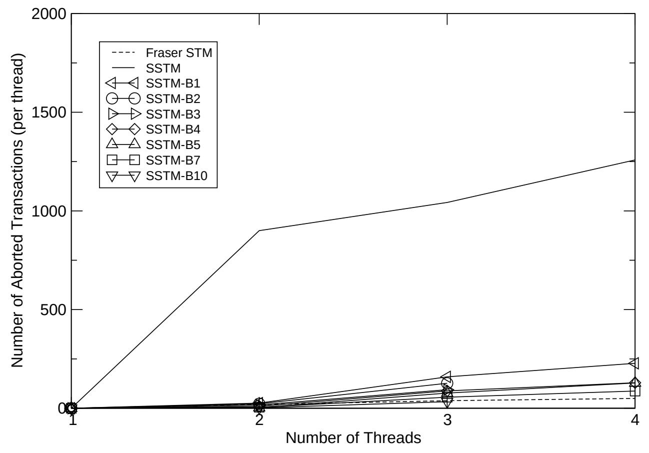


Figure 5.9: Average number of aborts (per thread) while attempting to commit 100,000 transactions (per thread) with a mix of 20% read-only (lookup) transactions and 40% each of update (insert and delete) transactions.

suffers, though fewer aborts occur if we allow write-only updates that do not first read the current state of the set. The tradeoff worsens as the percentage of read-only transactions increases because lookups are not improved by encoded operations, while there are fewer update transactions to cause aborts. SSTM suffers more aborts than FSTM: it is not clear how much this depends on the different overhead experienced by these two STM, although faster performance is likely to generate fewer opportunities for transactions to overlap.

## 5.5 Using Timestamps

Given the performance limitations of TS-STM in Section 5.4, we developed a different STM, called TS-STM, that maintains the lock-free and serializable properties of TS-STM, but uses a timestamp mechanism to provide these properties. In an effort to improve the performance, we add a constraint to eliminate blind writes, resulting in a linearizable STM[8] that achieves significant performance improvements as we show in Section 5.6.

Although LSA-STM by Riegel, et al. [96] use timestamps to provide linearizability and obstruction-freedom, the use of timestamps makes the LSA-STM quite similar to TS-STM that provides serializability and lock-freedom. Indeed, an earlier version SI-STM [97] provides snapshot isolation instead of linearizability, which can be extended to full serializability. Direct comparisons of the implementations are not viable since they are written in different languages.

Figure 5.10 shows the data structures used by our algorithm. A transaction $T_i \in \mathcal{T}$ contains 1) a status field equal to *undecided, committed,* or *aborted* to indicate whether $T_i \in \mathcal{U}$, $T_i \in \mathcal{C}$, or $T_i \in \mathcal{A}$, respectively, 2) the set of read and write operations, 3) a timestamp, TS($i$), used to order transactions, and iv) a unique identifier $i$ composed of the thread's unique id and a count of the thread's transactions.[9]

Timestamps are further composed of 1) a counter, 2) a thread id to differentiate timestamps with identical counters, and 3) local bits that otherwise order read-only and update transactions. Logical time represented by a global counter, GTS, is assigned to the counter field of a timestamp of a transaction before the transaction

[8]The set of linearizable executions is a subset of the view-serializable executions.

[9]The counters can be rolled over during a quiescent period by resetting the counters on the latest committed versions of objects.

```
// Transaction id
xid_t pad : 2 // unused (matches status_t)
      tid : 7 // thread id
      x_count : 23 // transaction counter

// Timestamps are counter and thread id
timestamp_t local : 2, // order read-only and updates
            tid : 7 // id breaks ties with counter
            ts_count : 23 // counter

// Status is counter, thread id, and status bits
status_t local : 2, // in progress, aborted, committed?
         tid : 7, // aborting thread id
         x_count : 23 // aborting thread counter

// Transaction data
transaction $T_i$
   xid_t $i$ // Transaction id
   timestamp_t ts // Timestamp
   status_t status // 2 LSB indicate status
   obj_version * reads // List of versions read
   obj_version * writes // Cache of writes
   timestamp_t cts // Commit tstamp.

// Data object
object $x$
   obj_version * history // Sequence of versions by TS

// Version of object written by transaction
obj_version $x_k$
   obj_version * next // Next version in history
   timestamp_t readcert // Highest TS of reading trans.
   transaction * creator // $T_k$
   void * data // Object representation

// Global counter to assign to timestamps
int GTS := 0
```

Figure 5.10: Data structures used by TS-STM.

```
// Read latest version of object that doesn't create a
// cycle in PMVSG and record version in transaction.
procedure read-object ( $T_i$ , $x$ ) returns $x_k \in x.\text{history}$
   // In reverse $x$.history order
   foreach $\{x_k : x_k \in x.\text{history} : (T_k \notin \mathcal{A}) \wedge (\text{TS}(k) < \text{TS}(i))\}$ do
      if $T_k \in \mathcal{U}$ then
         help-commit-transaction ( $T_k$ )
      if $((T_j \in \mathcal{C}) \wedge (\text{CTS}(k) < \text{TS}(i)))$ then
         $T_i$.reads += $x_k$
         return $x_k$
   return error

// Get new version to hold update
procedure write-object ( $T_i$ , $x$ ) returns $x_i$
   $x_i$ := alloc new version
   $x_i$.readcert := TS($i$)
   // Assign previously read values
   if ( $\exists x_k : x_k \in x.\text{history} : r_i[x_k] \in T_i$.reads ) then
      $x_i$ := $x_k$
   $T_i$.writes += $x_i$
   return $x_i$

// Create a new object
procedure create-object ( $T_i$ ) returns $o \in \mathcal{O}$
   $o$ := alloc new object
   $o$.history := $\perp$

// Mark object for deletion if transaction commits.
procedure destroy-object ( $o$ )
   < Mark object for deletion. >
```

Figure 5.11: API algorithm to access objects.

reads any objects. The global counter GTS is later incremented before an update transaction is committed, ensuring update transactions obtain unique timestamps. In this manner, the algorithm can quickly deduce which versions were created after a transaction began execution because the corresponding update transactions will have later timestamps. It is not necessary to distinguish between read-only transactions because they do not modify any data, but we do use two bits to distinguish between read-only and update transactions. Many read-only transactions by the same thread may share a timestamp.

Our algorithm uses timestamps in a tradeoff between concurrency and performance. Timestamps can summarize progress with a simple counter, allowing easy detection of conflicting transactions. Although this simplification improves performance, it may lead to false conflicts, resulting in transactions that are unnecessarily aborted. We use these timestamps to prove that the set of committed transactions is 1-SR in Section 5.5.1.

In our implementation (see Figure 5.10), an object $o \in \mathcal{O}$ contains only a history, represented by a lock-free list of object versions. Each object version $o_k \in o.\text{history}$ contains 1) pointers to other versions used to implement the history list, 2) a *readcert* timestamp representing the highest timestamp of a committed transaction that has read the version, 3) a pointer to the creating transaction, and iv) the object data interpreted by the application.

The API of TS-STM, shown in Figure 5.12, provides the same methods to read, write, and create objects as well as to commit, abort, and validate transactions. A thread $p$ begins a transaction by calling `begin-transaction`, which returns a data structure representing the transaction, initially undecided, with a timestamp equal to the current value of the global counter GTS.

The read procedure returns the latest committed version $o_k$ that is committed before the reading transaction $T_i$ begins as determined by $\text{TS}(k) < \text{TS}(i)$. Our algorithm prevents inconsistent reads, but does not use incremental validation wherein all previous objects read are checked for consistency with the new object read. To prevent incremental validation, we make a simplifying assumption: there are no blind writes. Every write requires a corresponding read of the object that creates a reads-from dependency:

**Condition 1** (No blind writes.)**.** *If $w_i[x_i] \in T_i$, then $r_i[x_k] \in T_i$.*

```
procedure begin-transaction ( thread_id ) returns $T \in \mathcal{U}$
   $T_i$.version.tid := thread_id
   $T_i$.version.count := $i$
   $T_i$.status.local := undecided
   TS($i$).tid := thread_id
   TS($i$).local := 0
   TS($i$).count := GTS+1
   CTS($i$) := 0
   return $T_i$

// Abort transaction by marking status.
// This trans's operations will not be
// visible to any other trans.
procedure abort-transaction ( $T_i$ )
   status_t newstatus := (aborted, $\perp$, $\perp$)
   CAS($T_i$.status, undecided, newstatus)

// Read-only trans. are locally committed.
procedure commit-transaction ( $T_i$ )
   if ( $\neg\exists x : w_i[x_i] \in T_i$ )
      status_t newstatus := (committed, $\perp$, $\perp$)
      CAS($T_i$.status, undecided, newstatus)
   else
      help-commit-transaction($T_i$)

// Validate transaction.
procedure validate-transaction ( $T_i$ ) returns boolean
   foreach $\{x : x \in \mathcal{O} : (r_i[x_k], w_i[x_i]) \in T_i\}$ do
      if ( $(\exists x_j : (x_j \in x.\text{history}) \wedge (T_j \in \mathcal{C}) \wedge (\text{TS}(j) < \text{TS}(i))) \vee$ ($x_k$.readcert > TS($i$)) ) then
         return false
   return true

// Return transaction taken from aborted
// status field.
procedure get-aborter ( $T_i$ ) returns $T \in \mathcal{T}$
   xid_t $j$ = (0, $T_i$.status.tid, $T_i$.status.x_count)
   return $T_j$

// Abort transaction by marking status.
procedure help-abort-transaction ( $T_i$, $T_h$ )
   status_t newstatus := (aborted, $T_h$.version.tid, $T_h$.version.x_count)
   CAS($T_i$.status, undecided, newstatus)

// Decide all transactions that have
// marked a readcert.
procedure decide-readcert ( $x_k$, $T_i$ )
   let tentative := $x_k$.tentative
   foreach $\{T_l : T_l \in \mathcal{U} : T_l \in \text{tentative}\}$ do
      if $T_i \in \mathcal{U}$ then
         if TS($l$) < TS($i$) then
            help-commit-transaction($T_l$)
         else
            help-abort-transaction($T_l$, $T_i$)
```

Figure 5.12: API procedures to commit, abort, and validate transactions. The `enqueue` procedure must return the element enqueued successfully or the argument if the argument is already enqueued ([80] can be modified for this purpose). Finally, the &() operator returns the address of the argument.

```
// Attempt to help commit transaction.
procedure help-commit-transaction ( $T_i$ )
    // Step 1: Read consistency check
    foreach $\{x : x \in \mathcal{O} : r_i[x_k] \in T_i\}$ do
        $x_k$.tentative+ $= T_i$
        if ( $\exists x_j : x_j \in x$.history : $(\mathrm{TS}(k) < \mathrm{TS}(j)) \wedge (\mathrm{TS}(j) < \mathrm{TS}(i))$ ) then
            help-commit-transaction($T_j$)
            if $T_j \in \mathcal{C}$ then
                abort-transaction($T_i$)
                return

    // Handle write operations
    foreach $\{x : x \in \mathcal{O} : w_i[x_i] \in T_i\}$ do
        // Step 2: Enqueue new versions
        do $x_j :=$ enqueue($x$, $x_i$) while ( $x_j \neq x_i$ )

        // Step 3: Decide previous trans. history
        foreach $\{x_j : x_j \in x$.history : $\mathrm{TS}(j) < \mathrm{TS}(i)\}$ do
            help-commit-transaction($T_j$)

        // Step 4: Check latest write
        if ( $\exists x_j : (x_j \in x$.history$) \wedge (\mathrm{TS}(j) < \mathrm{TS}(i)) : (T_j \in \mathcal{C})$
             $\wedge(\neg\exists x_l : (x_l \in x$.history$) \wedge (\mathrm{TS}(j) < \mathrm{TS}(l)) \wedge (T_l \in \mathcal{C}) : (\mathrm{TS}(l) < \mathrm{TS}(i)))$ ) then
            if ( $\exists x_k : (x_k \in x$.history$) \wedge (r_i[x_k] \in T_i) : \mathrm{TS}(k) < \mathrm{TS}(j)$ ) then
                // Diff. write in between read and write
                abort-transaction($T_i$)
                return
            decide-readcert($x_j$, $T_i$)
            if ( $x_j$.readcert $> \mathrm{TS}(i)$ ) then
                abort-transaction($T_i$)
                return

    // Step 5: Change status and update global timestamp
    FAI(GTS, 1)
    status_t newstatus := (committed, $\perp$, $\perp$)
    CAS($T_i$.status, undecided, newstatus)

    // Step 6: Help aborters
    if $T_i \in \mathcal{A}$ then
        let $T_j :=$ get-aborter($T_i$)
        while $((T_j \neq \perp) \wedge (T_j \notin \mathcal{C}))$ do
            if $(T_j \in \mathcal{U})$ then
                help-commit-transaction($T_j$)
            if $(T_j \in \mathcal{A})$ then
                $T_j :=$ get-aborter($T_j$)

    // Step 7: Update readcerts
    if $T_i \in \mathcal{C}$ then
        timestamp_t newcts := (1, $T_i$.version.tid, GTS)
        CAS($T_i$.cts, 0, newcts)

        foreach $\{x : x \in \mathcal{O} : r_i[x_k] \in T_i\}$ do
            let rc_old := $x_k$.readcert
            while rc_old $< \mathrm{TS}(i)$ do
                rc_old := CAS( &($x_k$.readcert), rc_old, $\mathrm{TS}(i)$ )
            $x_k$.tentative− $= T_i$
```

Figure 5.13: API procedure to help commit a transaction. A thread invokes this procedure on an earlier transaction by a different thread to ensure progress among a set of conflicting transactions.

Condition 1 allows us to implement an optimization using timestamps that eliminates incremental validation.[10] Further, read-only transactions can commit locally because all reads are already shown to be consistent. In addition to the timestamp TS($i$) assigned at transaction begin, an additional timestamp CTS($i$) is assigned at commit. A transaction $T_i$ then reads the latest version $x_k$ of an object such that CTS($k$) < TS($i$). We prove that this approach ensures one-copy serializability in Section 5.5.1.

As previously with SSTM, concurrent modification of objects can create cycles in the MVSG that must be detected at commit time. To simplify detection of conflicts, each version $o_k$ contains a *read certification timestamp* (or *readcert*), $o_k$.readcert, that represents the latest read timestamp among the set of committed update transactions that have read $o_k$ as suggested in [11]. The procedure `decide-readcert` is used by transactions during the commit procedure to ensure the latest readcert value is obtained given concurrent possible updates to the readcert of the version read. The `create-object` and `write-object` procedures have similar purposes as in SSTM.

The `commit-transaction` procedure attempts to commit a transaction $T$. A read-only transaction commits immediately since no versions must be made public. Update transactions proceed through a series of steps in the function `help-commit-transaction`. Step 1 checks for read consistency by searching for a version later than that read but earlier than the transaction timestamp order—this version would conflict with the updated readcert. Step 2 adds the write operations in $T$ to the history of the corresponding objects in timestamp order. The enqueue eventually succeeds because update transactions have increasing timestamps, bounding the number of conflicts that concurrent inserts may encounter by the number of threads. At this point, we say that the transaction is *published* in TS-STM, and hence can be aborted or committed by any thread. Step 3 ensures that the previous versions are all committed so $T$ can check write consistency in the Step 4. In Step 5, the global counter GTS is increased to mark progress, and the status of $T$ is changed to *committed.* Step 6 guarantees progress by attempting to commit another transaction if $T$ is aborted. Finally, if $T$ is committed, Step 7 updates the readcert timestamps of the versions read by $T$.

[10]The assumption of no blind writes changes serializability from an NP-complete problem to an efficiently decidable one [91].

We say that a transaction $T_i$ is *committed* if the CAS of the status of $T_i$ to *committed* succeeds in Step 5. Conversely, $T_i$ is aborted if a CAS of the status of $T_i$ to *aborted* succeeds. We continue to save the id of the transaction responsible for an abort in the same memory location as the status field of the aborted transaction to provide for helping by threads whose transaction was aborted.

### 5.5.1 Ensuring One-Copy Serializability

To prove the set of committed transactions is 1-SR, we must show that there exists a history order "$\ll$" to all versions of an object where the MVSG($\ll, \mathcal{C}$) is acyclic. We define the history order "$\ll$" to be $x_j \ll x_k$ if and only if $\mathrm{TS}(j) < \mathrm{TS}(k)$. This order is well-defined over update transactions because $\mathrm{TS}(j) \neq \mathrm{TS}(k)$ by construction. We prove the MVSG is acyclic by showing that for each type of edge in the graph, the transactions are ordered.

**Lemma 2.4** (reads-from). *If $r_i[x_k]$, then* $\mathrm{TS}(k) < \mathrm{TS}(i)$.

*Proof.* The `read-object` function only returns $x_k$ such that $\mathrm{TS}(k) < \mathrm{TS}(i)$. □

**Lemma 2.5** (write-read). *If $w_j[x_j]$ and $r_i[x_k]$ where $x_j \ll x_k$, then* $\mathrm{TS}(j) < \mathrm{TS}(k)$.

*Proof.* Direct from definition of the history order "$\ll$". □

**Lemma 2.6** (read-write). *Suppose $T_i$ is an update transaction. If $r_i[x_k]$ and $w_j[x_j]$ where $x_k \ll x_j$, then* $\mathrm{TS}(i) < \mathrm{TS}(j)$ *if $i \neq j$.*

*Proof.* Assume the transactions $T_i$, $T_j$, and $T_k$ are committed. During the commit of transaction $T_i$, $x_k$.readcert is increased to at least $\mathrm{TS}(i)$, implying $\mathrm{TS}(i) \leq x_k$.readcert ($x_k$.readcert may be further increased by other transactions). We show that $x_k.\mathrm{readcert} \leq \mathrm{TS}(j)$, providing $\mathrm{TS}(i) < \mathrm{TS}(j)$ if $i \neq j$.

We prove $x_k.\mathrm{readcert} \leq \mathrm{TS}(j)$ holds by induction on the finite number of versions $x_l$ such that $\mathrm{TS}(k) < \mathrm{TS}(l) < \mathrm{TS}(j)$. In the base case, there are no versions in the history of $x$ such that $\mathrm{TS}(k) < \mathrm{TS}(l) < \mathrm{TS}(j)$; that is, $x_k$ is the latest committed version before $x_j$. During Step 3 of `hct`($T_j$), $x_k$ is committed if still undecided. $T_j$ is then aborted in Step 4 (after concurrent transactions are committed by `decide-readcert`($x_k$)) unless $x_k.\mathrm{readcert} \leq \mathrm{TS}(j)$. If another transaction $T_i$ attempts to later increase $x_k$.readcert after $T_j$ completes `decide-readcert`, then $x_j$ is already enqueued. Hence, during Step 1 of `hct`($T_i$), the later enqueued version

$x_j$ is found such that $\text{TS}(j) < \text{TS}(i)$, forcing $T_i$ to abort; that is, $T_i$ cannot change $x_k.\text{readcert} \leq \text{TS}(j)$.

For the induction step, we assume that two transactions $T_k$ and $T_l$ are committed with a finite number of transactions $T_m$ such that $\text{TS}(k) < \text{TS}(m) < \text{TS}(l)$. Then, $x_k \ll x_l$ and $\text{TS}(k) < \text{TS}(l)$ imply that $x_k.\text{readcert} \leq \text{TS}(l)$. We show that the Lemma holds for $T_j$ where $x_l \ll x_j$, and $x_l$ is the latest previous version of $x$ such that $\text{TS}(l) < \text{TS}(j)$. We extend the induction hypothesis, $x_k.\text{readcert} \leq \text{TS}(l)$, with $\text{TS}(l) \leq x_l.\text{readcert}$ since $T_l$ is committed. The argument of the base case applies to show that $x_l.\text{readcert} \leq \text{TS}(j)$, completing the proof. □

The following Lemma provides a useful property of the `read-object` function that states later versions beyond that read by a transaction necessarily commit later in timestamp order.

**Lemma 2.7.** *If $x_k \ll x_j$ and $r_i[x_k]$ (where $T_j, T_k \in \mathcal{C}$), then* $\text{TS}(i) < \text{CTS}(j)$.

*Proof.* When $T_i$ reads $x$, either $x_j$ was not enqueued or $T_i$ must have found $\text{TS}(i) < \text{CTS}(j)$ directly. The former case also implies $\text{TS}(i) < \text{CTS}(j)$. If $x_j$ is enqueued during the read operation but is not yet committed, then $T_i$ helps to decide $T_j$ before checking $\text{TS}(i) < \text{CTS}(j)$ directly. If $x_j$ is not yet enqueued, the global counter GTS is incremented after $x_j$ is enqueued (after $T_i$ reads $x_k$) when $T_j$ is committed, providing $\text{CTS}(j) > \text{TS}(i)$. □

**Theorem 3** (1-SR)**.** *The set of committed transactions $\mathcal{C}$ is one-serializable.*

*Proof.* We first consider the subset $\mathcal{C}_U$ of $\mathcal{C}$ containing only update transactions. Lemmas 2.4–2.6 show that $T_i \rightarrow T_j$ implies $\text{TS}(i) < \text{TS}(j)$. The edges in the $\text{MVSG}(\ll, \mathcal{C}_U)$ are thus in increasing timestamp order. The total order of the corresponding timestamps implies that the MVSG is acyclic and the log is 1-SR by [11].

Next we consider $\mathcal{C} = \mathcal{C}_U \cup \mathcal{C}_{RO}$ that also contains the set of committed, read-only transactions $\mathcal{C}_{RO}$. We provide an algorithm for generating a serialization $S$ to show that the corresponding log is 1-SR. We first form a total order of the set $\mathcal{C}_{RO}$ using timestamp order, breaking ties (which are possible only among read-only transactions) arbitrarily. For each member $T_i$ of $\mathcal{C}_{RO}$ in this order, add the subset $\{T_k : T_k \in \mathcal{C}_U : \text{CTS}(k) \leq \text{TS}(i)\}$ to $S$ in timestamp order, then add $T_i$ to $S$.

$S$ is 1-SR, as follows. First, $T_k$ appears before $T_i$ in the log for each operation $r_i[x_k]$ because CTS($k$) < TS($i$) by definition. Second, a simple induction on the sequence of versions between $x_j$ and $x_k$ implied by TS($j$) < TS($k$) given Condition 1 shows that CTS($j$) < TS($k$) so that $T_j$ must appear before $T_k$ in $S$. Third, $T_i$ appears before any $T_j$ where $x_k \ll x_j$ and $r_i[x_k]$ by construction. If $T_i \in \mathcal{C}_U$, then TS($i$) < TS($j$) by Lemma 2.6 so that $T_i$ appears before $T_j$ by timestamp ordering. If instead $T_i \in \mathcal{C}_{RO}$, we infer TS($i$) < CTS($j$) by Lemma 2.7, so that $T_j$ cannot be included before $T_i$. □

### 5.5.2 Ensuring Lock-Freedom

TS-STM commits read-only transactions locally. If consistent versions are available to read (that is, they have not been garbage-collected), then a read-only transaction will always commit. However, update transactions must schedule writes to objects in such a way that 1-SR is maintained. During concurrent updates to objects several transactions may conflict in their order of reads and writes. To choose in general which transaction to abort, our algorithm uses the total order on update transactions given by the timestamps of the transactions. Our use of timestamps maintain two important properties: first, there is a minimum; and second, *dist*($i$,$j$) is finite for any two update transactions $T_i$ and $T_j$ where *dist*($i$,$j$) is the number of timestamps between TS($i$) and TS($j$). Both properties are evident from the construction and assignment of timestamps.

Note that the timestamp order is not used to determine the order in which transactions will eventually commit—transactions with earlier timestamps may well commit after transactions with later timestamps. Rather, timestamps are used to determine the fate of any undecided transaction $T_j$ that a thread $p$ encounters as it checks the read and write consistency of the transaction $T_i$, that $p$ is trying to commit. Specifically, $p$ will attempt to commit only those $T_j$ that have earlier timestamps than $T_i$ and attempt to abort the rest. The two properties of timestamps ensure that any recursion triggered while applying this policy will terminate.

We prove that our algorithm is lock-free by showing that for any given timestamp, there is always a committed transaction with a later timestamp. The proof relies on all threads agreeing to help commit the minimum update transaction (in timestamp order) in any given set of conflicting transactions. Note that since read-only transactions commit locally, we restrict our attention to the set of update transactions.

**Lemma 3.8.** *If there is at least one correct thread, then there exists an infinite sequence of decided transactions $\mathcal{T}^p$ such that for each transaction $T_i \in \mathcal{T}^p$, some thread executes* `hct`$(T_i)$.

*Proof.* The lemma follows from termination of the `commit-transaction` procedure, allowing a single thread to attempt to commit an infinite sequence of transactions, each of which is decided when `commit-transaction` completes. Before the `commit-transaction` procedure terminates, CAS is used to set the status to *committed* or *aborted* (lines 21, 63, 82, 86, and 91). The procedure terminates because 1) the recursive invocations of `hct` terminate with the transaction with the minimum timestamp, 2) the `enqueue` procedure eventually succeeds because the versions are enqueued in timestamp order and *dist*$(i,j)$ between any conflicting enqueued version $x_j$ is finite, 3) all other invocations are simple, non-recursive functions, and 4) all loops are over the finite number of operations in a transaction. □

Note that the lemma refers only to *decided* transactions—lock freedom requires us to show that within $\mathcal{T}^p$ there exists an infinite subsequence of committed transactions. The focus of the proof is necessarily on aborted transactions. We consider separately two classes: *forced-aborts,* if `get-aborter`$(T_i) \neq T_i$ and *self-aborts,* otherwise. First, we prove a useful lemma over each class.

**Lemma 3.9.** *For any self-aborted transaction $T_i$, there exists a transaction $T_j \in \mathcal{C}$ such that* $\mathrm{TS}(j) < \mathrm{TS}(i)$ *and* $\mathrm{TS}(i) < \mathrm{CTS}(j)$.

*Proof.* There are three points at which a transaction is aborted in `hct` resulting in `get-aborter`$(T_i) = T_i$ (lines 63, 82, and 86). Each occurs precisely when a transaction $T_j$ is observed such that $\mathrm{TS}(j) < \mathrm{TS}(i)$. We further infer $\mathrm{TS}(i) < \mathrm{CTS}(j)$ from the read operation given at line 57 or implied by Condition 1 and the write $w_i[x_i]$. Lemma 2.7 then provides $\mathrm{TS}(i) < \mathrm{CTS}(j)$ given $\mathrm{TS}(k) < \mathrm{TS}(j)$. □

**Lemma 3.10.** *For any forced-aborted transaction $T_i$, there exists a transaction $T_j \in \mathcal{C}$ such that* $\mathrm{TS}(j) < \mathrm{TS}(i)$ *and either 1)* $\mathrm{TS}(i) < \mathrm{CTS}(j)$ *or 2) in addition, there exists an intermediate aborted transaction $T_n$ such that* $\mathrm{TS}(j) < \mathrm{TS}(n) < \mathrm{TS}(i)$, $\mathrm{TS}(n) < \mathrm{CTS}(j)$, *and* $\mathrm{TS}(i) < \mathrm{CTS}(n)$.

*Proof.* An aborted transaction where `get-aborter`$(T_i) \neq T_i$ occurs only in the context of the procedure `decide-readcert` where `get-aborter`$(T_i) < T_i$. Hence, in Step 6 of `hct` when the thread recurses on `get-aborter`$(T_i)$, it only invokes `hct` on an earlier transaction. The minimum transaction $T_m$ guaranteed by our assignment of timestamps bounds the recursion, and the minimum transaction is either committed or self-aborted. If $T_m$ is committed, $\text{TS}(m) < \text{TS}(i)$ by the requirements for recursion. Otherwise, Lemma 3.9 provides for the existence of $T_j$ where transitivity ensures $\text{TS}(j) < \text{TS}(m) < \text{TS}(i)$.

Next, we show $\text{TS}(i) < \text{CTS}(j)$. When the transaction $T_a$ (= `get-aborter`$(T_i)$) aborts $T_i$ during the procedure `decide-readcert`, $T_i$ must already be added to $x_k$.tentative. If $T_a$ is committed, $\text{TS}(i) < \text{CTS}(a)$ because $\text{CTS}(a)$ is set after GTS is incremented. If instead $T_a$ is also aborted by another transaction $T_b$, then we note that $T_i$ must also have been added to $x_k$.tentative before $T_a$ was aborted since $T_a$ must be undecided before aborting $T_i$ in `decide-readcert`. By a simple induction, $\text{TS}(i) < \text{CTS}(n)$ for transaction $T_n$ in the sequence $T_i, T_a, T_b, \ldots, T_n$ such that $T_k$ is aborted by $T_{k+1}$ ($k \in \{i, a, \ldots, n\}$) in the sequence and $T_n$ is not force-aborted. $T_n$ is either committed or self-aborts. In the former case, part 1) of the lemma holds, while the latter implies the more complicated part 2). Lemma 3.9 provides that there exists some committed $T_j$ where $\text{TS}(j) < \text{TS}(n)$ and $\text{TS}(n) < \text{CTS}(j)$. □

Next, we place some bounds on concurrent transactions so that we can infer infinite sequences.

**Lemma 3.11.** *For any decided transaction $T_j$, there is a bound $n - 1$ (where $n$ is the number of threads) on the size of the set of concurrent transactions: $\{T_i : i \in \mathcal{T} : \text{TS}(i) < \text{TS}(j) < \text{CTS}(i)\}$.*

*Proof.* Each transaction $T_i$ ends after $T_j$ begins so that the next transaction by the same thread will necessarily have a timestamp greater than $T_j$. Hence, the set is bounded by the number of concurrent threads. □

Finally, we can prove that there is eventually a new committed transaction if a correct thread continues to execute transactions.

**Theorem 4.** *If there is at least one correct thread $t$, then there exists an infinite sequence of committed transactions.*

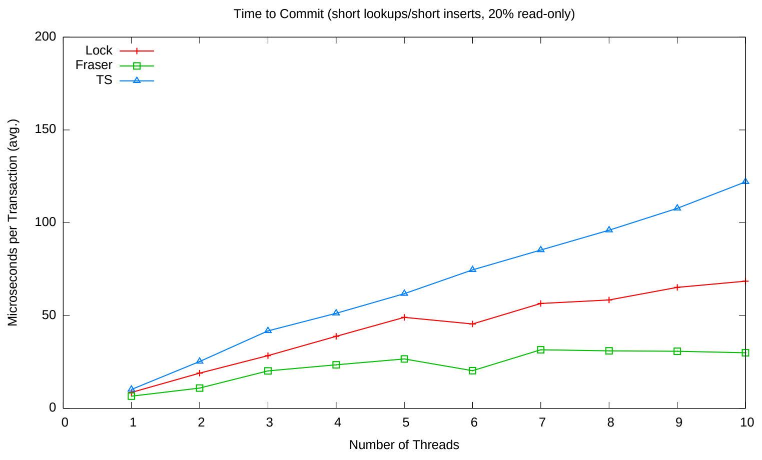
*Proof.* By Lemma 3.8 we infer the existence of an infinite sequence of decided transactions with timestamps greater than some bound *TS*. We consider the subsequence of these decided transactions that are executed by $t$, which is necessarily infinite given that $t$ is correct. We consider an infinite suffix of this sequence consisting entirely of aborted transactions (otherwise, the Theorem is direct).

By Lemmas 3.9 and 3.10 (part 1)), when $t$ attempts to commit each transaction $T_i$ in the infinite aborted sequence beginning after *TS* (that is, *TS*$<$ TS($i$)), there exists a corresponding committed transaction $T_j$ such that TS($j$) $<$ TS($i$) and TS($i$) $<$ CTS($j$). Lemma 3.11 provides that only a finite number of $T_i$ correspond to any specific committed $T_j$ so that there must also be an infinite sequence of committed $T_j$. A similar argument holds for the final case of Lemma 3.10 (part 2)). In that case, there must be an infinite sequence of $T_n$, which further implies an infinite sequence of committed transactions $T_j$ by the concurrency requirement TS($j$) $<$ TS($n$) $<$ CTS($j$). □

## 5.6 Evaluation of Timestamps

Our testbed consists of a Sun E5000 server with 15 400MHz UltraSPARC II cpus and 2GB memory running SunOS 5.8. We tested our serializable STM (TS-STM) against a lock-free OSTM [32] and an STM implemented using mutual exclusion locks (LOCK), all of which are implemented in C. The test application (from the OSTM implementation) operates on an integer set stored as a red-black tree implemented in each different STM. Each element in the tree is maintained by a block of memory provided by the STM, and no garbage collection is performed by TS-STM. Elements in the set are found using a lookup in the tree within a read-only transaction, while insert and delete operations require update transactions. Every transaction in our experiments performs a single insert, delete, or lookup operation on the set, but may require multiple reads and/or writes to different transactional memory blocks. We introduced new longer lookup and insert operations in which every node in the tree is read or written, respectively, up to the value being looked up or inserted. In our experiments with a tree of 256 elements, on average a long operation accesses two orders of magnitude more memory blocks. Longer transactions have been shown to evoke quite different behavior in STMs in STMBench7 [42].

Figures 5.14 and 5.15 demonstrate the scalability of TS-STM. The y-axis



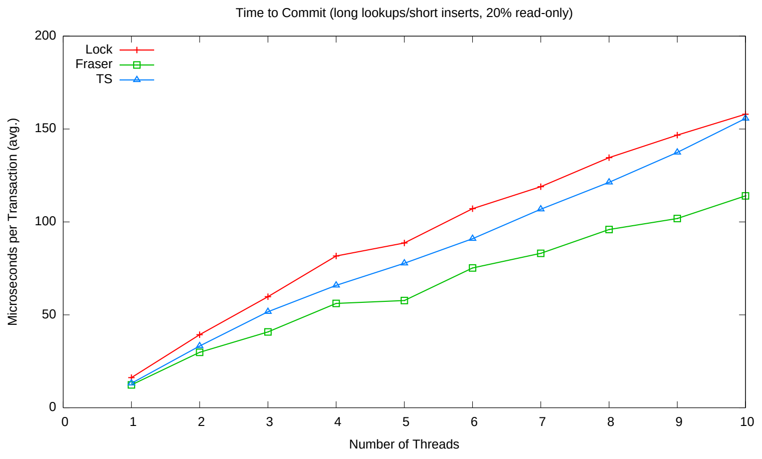
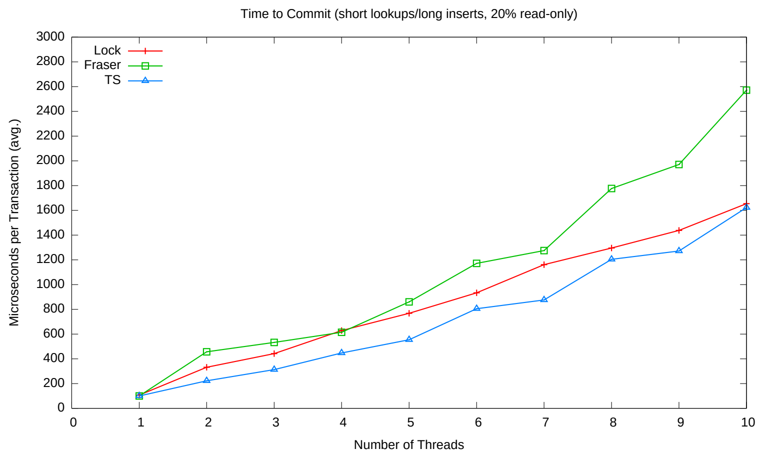


Figure 5.14: Average time to commit a transaction (per thread) of short and long operations with varying number of threads. 20% of the transactions are read-only lookups. The remaining are evenly split update transactions between insert and delete transactions. Long operations typically operate on two orders of magnitude more blocks than short operations. In the first graph all operations are short while the lower graph includes long reads (but short writes). Delete transactions are always short. OSTM is denoted by the primary author, Fraser.



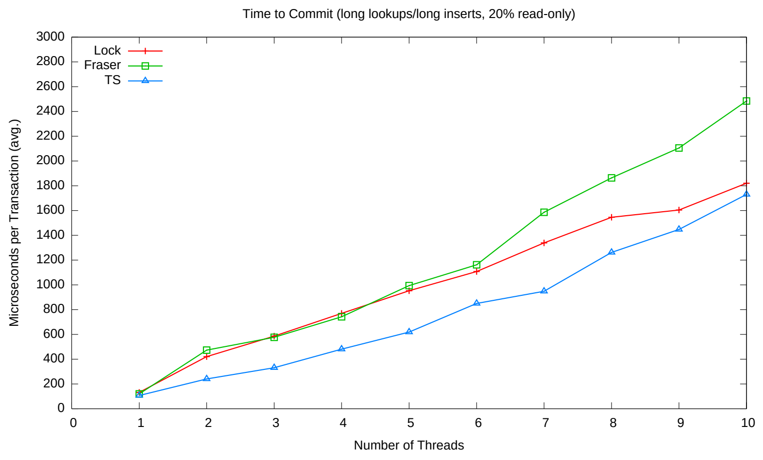
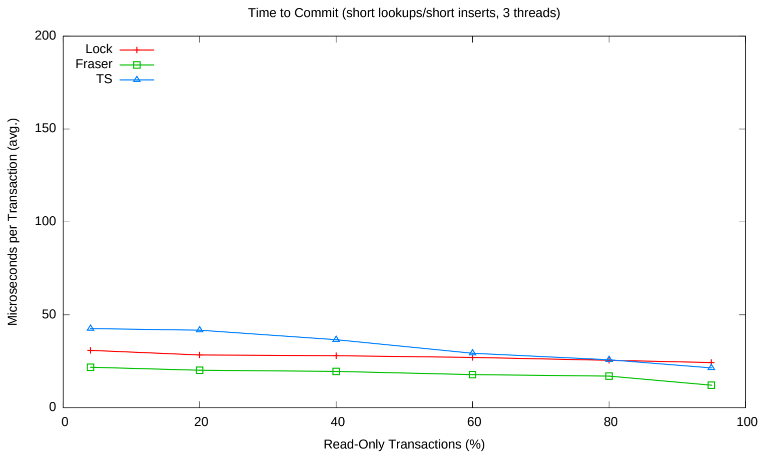


Figure 5.15: Average time to commit a transaction (per thread) of long and short operations with varying number of threads. 20% of the transactions are read-only lookups. The remaining are evenly split update transactions between insert and delete transactions. Long operations typically operate on two orders of magnitude more blocks than short operations. In the first graph reads are short while writes are long while all reads and writes in the lower graph are long. Delete transactions are always short. OSTM is denoted by the primary author, Fraser.



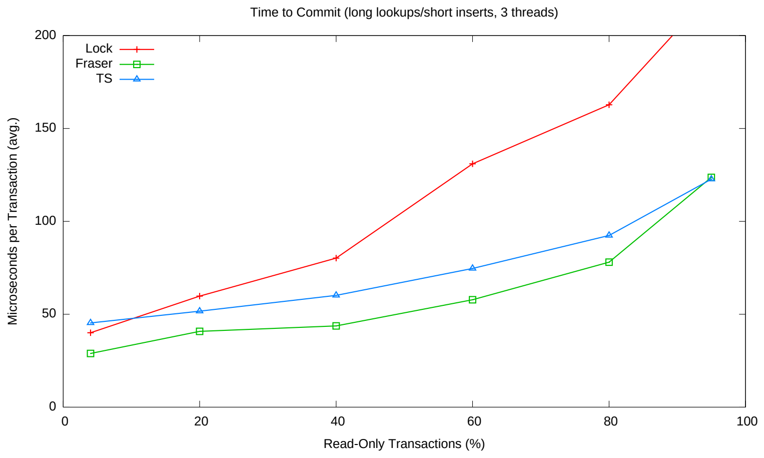
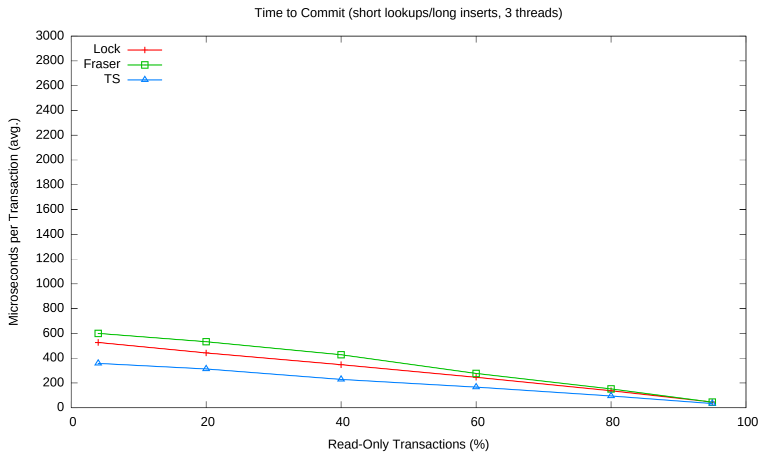


Figure 5.16: Average time to commit a transaction (per thread) of short/long operations with varying percentage of read-only (lookup) transactions. The remaining transactions are update transactions evenly split between insert and delete transactions. Long operations typically operate on two orders of magnitude more blocks than short operations. Delete transactions are always short. There are three threads, and the trends with different numbers of threads are similar. In the first graph both reads and writes are short while in the lower graph includes long reads and short writes. OSTM is denoted by the primary author, Fraser.



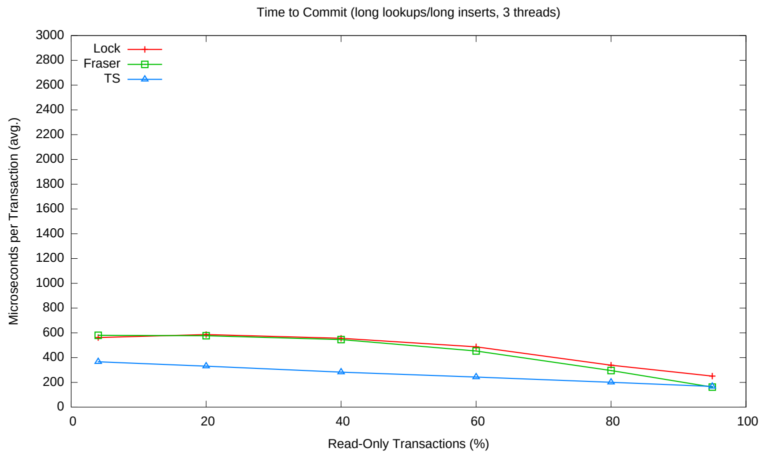


Figure 5.17: Average time to commit a transaction (per thread) of short/long operations with varying percentage of read-only (lookup) transactions. The remaining transactions are update transactions evenly split between insert and delete transactions. Long operations typically operate on two orders of magnitude more blocks than short operations. Delete transactions are always short. There are three threads, and the trends with different numbers of threads are similar. In the first graph reads are short and writes are long while in the lower graph both reads and writes are long. OSTM is denoted by the primary author, Fraser.

shows the average time to commit a single transaction (insert, delete or lookup on the set) given different numbers of threads. All STM operate on a tree with 256 keys and 20% read-only (lookup) transactions. TS-STM clearly has significant overhead when transactions are short because all threads are searched for conflicting transactions. OSTM registers conflicting transactions using logical locks, while LOCK uses wait queues. Both approaches scale much better on short transactions, but begin to suffer as we introduce longer transactions. The long transactions perform a significant number of updates (usually the entire tree is read and/or updated in the transaction) in order to demonstrate the scalability of the transaction size. Using TS-STM, committing long transactions scales better with the number of threads. Both OSTM and LOCK will serialize accesses to the tree by logically locking the entire tree whereas TS-STM provides more opportunity for concurrency by performing consistency checks on transactions that executed concurrently. These extra opportunities to execute a transaction only offset the overhead of the consistency checks when the transactions become long and the logical locking used by OSTM inhibits concurrency.

Figures 5.16 and 5.17 demonstrate the effects of varying the percentage of read-only transactions. In the experiments we report, the number of threads is fixed at three. Both figures show that for workloads with short transactions, the overheads of OSTM and MSTM are smaller than for TS-STM, but as the transactions become longer (that is, operate on more objects), TS-STM processes concurrent transactions faster. As one would expect, the average time to commit a transaction decreases as the overall percentage of read-only transactions increases because there are fewer conflicts. However, in the case of long lookups and short inserts, the average actually increases because the long lookups are significantly longer than the the short update transactions. Two trends are in effect: 1) the average time to commit a read transaction is decreasing because of fewer conflicts with update transactions resulting in fewer aborted transactions and 2) the number of long transactions—where long transactions are significantly longer than the short transactions—is increasing steadily. Because of the large discrepancy in the execution time of long and short transactions, the latter trend dominates, and the average time-to-commit increases for TS-STM.

Although our experiments appear to provide support for the belief that the additional executions admitted by the serializability correctness criteria can result in better throughput, note that TS-STM also provides linearizable executions (if $T_a$ ends before $T_b$ begins, $\text{TS}(a) < \text{TS}(b)$). The logical locking used by linearizable STM provides an efficient mechanism to limit concurrency to those executions that are linearizable. Our implementation of serializability instead allows more concurrent execution that results in more aborted transactions, but admits more concurrency when transactions involve many objects because reads are allowed to return stale data. Without timestamps (or other versioning information), logical locks cannot determine which previous version should be read, requiring instead reads to return the most recent committed version.

## 5.7 Summary

This chapter describes the first one-copy serializable, lock-free software transactional memory (SSTM) and our timestamped, lock-free serializable STM (TS-STM). Both are based on multiversion objects. Multiple versions allow read-only transactions to read previous data values, increasing the likelihood that they will commit under contention. Threads help commit transactions to provide progress and to ensure the one-copy serializability property, although because of concurrent execution some transactions may be aborted to guarantee one-copy serializability. We implemented both STM and evaluated the performance against other STM—the benefit of allowing higher concurrency is only seen for long-running transactions with many objects because our implementation of serializability has significant overhead. Although using timestamps reduces this overhead significantly, the approach also limits concurrency by assuming no blind writes (Condition 1).

Highlights to take away from this chapter:

- The lock-free progress guarantee can be implemented with the serializability correctness criterion using only the CAS primitive.

- Serializability can admit more executions—providing more concurrency—than linearizability, but requires checking concurrent executions for consistency. In our SSTM implementation that is not linearizable, checking incurs high overhead. Our implementation of TS-STM using timestamps reduces the overhead but also admits only the subset of executions that are linearizable.

- Our assumption of no blind writes (Condition 1) allows for simpler consistency checks, eliminating incremental validation of reads. Further performance improvements may be possible. It remains an open question whether Condition 1 can be relaxed without requiring incremental validation of reads.

# Chapter 6

# Mayfly: O-MTFT and Leases

In this chapter, we present our design of O-MTFT fault-tolerance for Java, which we call *Mayfly.* To recover as many threads as possible, we address an overlooked type of error in multithreaded applications: *hung threads.* Hung threads perform no work—for example, in the case of deadlock if a lock is unobtainable—or useless work—for example, when a thread is caught in an infinite loop because of inconsistent data. In this chapter, we propose techniques that we believe will, by enabling the recovery of hung threads, 1) broaden the fault coverage of multithreaded systems (increasing the likelihood that the application can make progress), 2) improve the application's performance by keeping correct threads performing useful work where possible, and 3) reduce recovery time by bounding downtime due to hung threads.

## 6.1 Hung Threads

The incidence of hung threads is difficult to quantify. OS dependability studies often do not consider hangs as a type of failure [5, 8, 40, 60, 75, 100, 103, 108]. However, in [107] Sullivan and Chillarege report 11% of all failures and 49% of a subset of high impact failures (as judged by the users) to be due to hangs. On the other hand, work with Ballista [26], testing the dependability of the OS (and system library) interface, has shown few hangs resulting from OS API errors. Typically, input to this interface is checked for consistency violations better than internal functions in order to maintain consistency of the kernel and supporting libraries. While Jarboui, et al. [58] corroborate the result, they also show that bits flipped using fault injection

```
public class ThreadLease {
  /* Constructors */
  public ThreadLease(long wall_clock_timeout, boolean auto_restart);
  public ThreadLease(long cpu_timeout, long wall_clock_timeout, boolean auto_restart);

   /* Accessors (ms) */
  public long getCPUTimeout();
  public long getClockTimeout();

   /* Will this thread restart at timeout? */
  public boolean isAutoRestarted();

   /* Is this thread leased? */
  public boolean isTimed();

   /* Lease info (default 0 is no timeout). */
  private long cpuTimeout = 0;
  private long clockTimeout = 0;
  private boolean restart = false;
}

class Thread {
  /* Restart thread before timeout (renew lease). */
  public void restart();

  /* Create CPU-leased thread. */
  public Thread(ThreadLease to);
  public Thread(Runnable target, ThreadLease to);
  ...

   /* Period for thread's CPU-based lease. */
  private volatile ThreadLease lease;
}
```

Figure 6.1: Overview of Mayfly Lease API. The library class `java.lang.Thread` is modified to add constructors that take leases. The `auto_restart` flag specifies whether the corresponding thread should be restarted (with identical arguments) upon timeout.

on internal functions arguments (as may occur with SEU errors) manifest 30% hang failures in the Linux kernel and applications, implying that transient errors are likely to result in hung threads since they may occur within the well-checked kernel interface. Madeira, et al. [76] provide further support showing a significant number of hang failures using injected faults.

We detect hung threads caused by an infinite loop or deadlock by leasing [35] cpu time and wall-clock time, respectively, to threads. Leasing execution time and using an STM to manage shared memory go hand-in-hand to help the application make progress. The STM provides access guarantees even if a thread's lease expires early; shared memory remains always accessible. Detecting and recovering hung threads provides more threads to help the application make progress.

```
boolean retry = false;
do {
   ThreadLease lease = thread.lease;

   try {
      if( (null != lease) && lease.isTimed() ) {
         /* Set timeouts based on lease */
         startTimeoutTimers(lease);
      }

      /* Invoke thread-specific execution */
      thread.run();
      retry = false;
   } catch( ThreadTimeoutException tte ) {
      if( null != lease ) {
         retry = lease.getRestart();
      }
   } catch( ThreadRestartException tre ) {
      retry = true;
   } catch( RuntimeException re ) {
      retry = lease.getRestart();
   }
} while( retry );
```

Figure 6.2: Overview of thread exception handling. We replace the invocation of `thread.run` with the above timeout loop. Note that this code example does not include recovery-specific code.

## 6.2 Design

Mayfly supports measuring the lifetime of a thread using two metrics: computing time and wall-clock time. A thread can simultaneously have both deadlines set, expiring when the first is reached. To enable leasing of the CPU, we introduce a `ThreadLease` object and modify the `java.lang.Thread` class that represents Java threads to use our leases. Figure 6.1 shows these changes. A `ThreadLease` object is created by either specifying a wall-clock timeout or a cpu usage timeout, both given in milliseconds. A zero value for a timeout disables that lease so that a thread can be created with any combination of wall-clock and cpu usage leases. The `restart` method added to the `java.lang.Thread` class implements an early renewal of the thread's lease—the thread will be restarted immediately rather than waiting for lease expiration. Using the `restart` method, threads can implement loops for perpetual execution such as occurs in a typical event-processing thread.

Figure 6.2 provides the changes to the algorithm to invoke a thread's `run` method that represents the code that a Java thread executes. We implement timeouts (and early restarts) using Java exceptions that can be caught by the programmer to perform application-specific recovery at timeout.

In order to set the expiration time of threads, the application writer must both design for restartable threads and estimate the lifetime of the thread. In order to provide support for estimating both lifetimes of a thread, we refer to the field of Worst-Case Execution Time analysis (WCET) [69], which provides a measure of the estimated maximum execution time of code segments for the purpose of designing applications with realtime constraints. WCET analysis uses models of processors and a set of possible inputs to deduce the maximum execution time of a code segment by exploring the space of possible executions using the processor model. Although WCET analysis is not mature enough to provide tight bounds on worst-case execution times for Java programs (especially with multithreading), we note that especially conservative bounds should be sufficient. A lease that expires after the true WCET can still provide a useful bound on the time lost to hung threads.

## 6.3 Lock-free Resource Access

O-MTFT specifies lock-free access to shared memory using an STM to ensure that correct threads can continue to make progress where data is available even in the presence of failures. However, threads may need access to other resources besides shared memory in order to make progress. Perhaps more fundamental than shared memory is access to the CPU to perform execution of bytecodes. The Java programming language provides for access to the CPU by guaranteeing multithreaded support. Eventually, a correct thread that is not waiting for a lock will be scheduled. Access to other resources such as the filesystem or network are not guaranteed. To ensure access to these resources, we also utilize the STM to mediate access to shared resources.

Using the side-effect handlers discussed in Chapter 3, we can intercept accesses to shared resources such as files and network connections. We convert the reads and writes to these resources to requests to a pool of system threads that actually perform the reads and writes. The input queue to the pool is implemented using the lock-free STM to guarantee that if any application thread fails, remaining threads will be able to enqueue new requests. Since the system pool only removes requests from the queue after they have been submitted to the operating system, the threads contain no state and are easily recovered by restarting. Java threads that perform synchronous access to shared resources are redirected by the side-effect handler to wait on their own condition variables that are not shared. When the request is completed by the operating system, the Java thread is woken up in order to retrieve the result. Although it would be feasible to use a single condition variable (or one variable for each type of resource), this approach depends heavily on Java's ability to cleanly release locks on failure. By using a condition variable per thread, locks are not shared between application threads. Our architecture uses the STM to distribute access to the shared resources so that the system maintains the goal of independent thread recovery.

## 6.4 Recovery

Figure 6.3 provides an overview of the support API of Mayfly for recovery. Each exception is wrapped into a `FailureEvent` class that attempts to encapsulate information useful for recovery. A log of all failures in the `FailureLog` object is provided to recovering threads to allow for online or offline data mining of failures. For example, a thread can determine that many of its previous failures had identical input, indicating recovery has failed. On the other hand, the inputs may be the same except come from different sources, possibly indicating that the input was recovered successfully, but is a good candidate for filtering due to its high probability of failure.

The `recover` method is invoked upon failure to allow a `Recoverable` object (simply a subclass of `java.lang.Runnable`) to recover from an error in an object-specific manner. For a thread that does not require any recovery of thread-local state to be consistent with shared memory, the `recover` method can simply call the method's `run` to restart execution. Many threads will require more complicated recovery to ensure that the thread-local state, which does not necessarily survive failure because it can be corrupted, is consistent with the shared data that other threads can observe. To this end, we utilize the multiversion nature of our STM implementations. Each transaction has a log of the versions used for reads and produced by writes. Using this log, a recovering thread can re-execute the transactions while recovering the thread-local state as well. Any transactions executed during recovery read from the log, and all writes are discarded. The STM can detect such recovery execution by simply checking a flag in the `Thread` object set by Mayfly during recovery.

As we discussed in Chapter 3, there are events that output to the environ-

```
class FailureEvent
{
   /* Location of failure */
   public Method getMethod();
   public int getLocation();
   public Object getInput();
   public Thread getThread();

   /* */
   public boolean fuzzyEquals(Method method, int location);

   /* Thread that failed */
   private Thread thread;
   /* Input to thread that failed */
   private Object input;

   /* Description of failure */
   Error error;
   RuntimeException runtime_exception;
   Exception exception;
};

class FailureLog {
   /* Insert new failure */
   public void addFailure(FailureEvent event);
   /* Get latest failure */
   public FailureEvent lastFailure();
   /* Get particular thread's last failure */
   public FailureEvent lastFailure(Thread thread);
   /* Get all failures for given thread */
   public Vector getFailures(Thread thread);
   /* Get all failures known to have occurred at location */
   public Vector getFailures(Method method, int location);

   /* Map of info on failures */
   private HashMap internalMap;
};

interface Recoverable
  extends java.lang.Runnable {
    /* Target for recoverable objects */
   public abstract void recover(FailureLog log);
};

class ThreadRecoverable
  extends Thread
  implements Recoverable {
   /* Drop-in replacements for java.lang.Thread constructors */
   public ThreadRecoverable();
   public ThreadRecoverable(String name);
   public ThreadRecoverable(ThreadGroup group, String name);
   ...

   /* Add recovery target to constructors */
   public ThreadRecoverable(Recoverable target);
   public ThreadRecoverable(String name, Recoverable target);
   public ThreadRecoverable(ThreadGroup group, Recoverable target);
   ...

   /* Add recovery target and leases to constructors */
   public ThreadRecoverable(Recoverable target, ThreadLease to);
   public ThreadRecoverable(String name, Recoverable target, ThreadLease to);
   public ThreadRecoverable(ThreadGroup group, Recoverable target, ThreadLease to);
   ...

    /* Invoke recovery target as necessary */
   public void recover(FailureLog log);

   /* Pointer to recovery target */
   private Recoverable recoverTarget;
};
```

Figure 6.3: Overview of Mayfly object support for recovery. Note that the `run` method example given in Figure 6.2 is modified to invoke the `recover` method after adding a `FailureEvent` to the log for any uncaught exceptions.

ment, and a failed thread must recover to a state consistent with this output. These *output commit* events are handled in C-MTFT by our side-effect handlers that ensure that the state of the threads is consistent with output commit events by managing access to the file system and network connections. The O-MTFT approach can also utilize side-effect handlers for output commit events. We can treat all output as access to shared resources (easily so for files and network connections) so that the side-effect handlers are invoked by our system thread pools that process read and write requests for shared resources. As discussed above, individual threads can recover using the STM transaction logs while the pool threads can use the side-effect handlers. We discuss recovering the entire application below when we discuss what to do when optimism fails.

Although Mayfly does not provide support for them, checkpoints can be used to provide a quicker path to recover the state of a failed thread instead of re-execution using saved transactions from the STM. A multithreaded checkpoint [116] saves a consistent state of an multithreaded application by saving the values of all the state variables and which commands are being executed by which threads. Consistency is typically defined according to the happens-before relation representing causality [63] so that an application that recovers from a checkpoint by restoring all of the state variables and execution of threads from the points saved in the checkpoint is indistinguishable with respect to the happens-before relation from the application when the checkpoint was saved. However, due to the use of logs for the STM, we believe independent checkpoints can be taken as a means of truncating the logs of shared data access maintained by the STM. Since the STM logs all accesses to shared memory by assumption, the independent checkpoints will not suffer from the *domino effect* [94] that requires correct threads to rollback.

## 6.5 When Optimism Fails

In this section, we discuss what to do when optimism fails and the entire application requires recovery—possibly because a hard error requires a restart or even because shared data is corrupted. The O-MTFT approach depends upon shared memory consistency being maintained even through thread failures, which cannot be guaranteed under the Capricious failure model. The techniques discussed in Section 4.2 can be used to determine when optimism has failed and C-MTFT, which does not depend on the assumption that shared memory remains consistent, is required to recover. The C-MTFT approach is transparent to the application and can be run simultaneously with O-MTFT, albeit with the significant overhead demonstrated in Chapter 3. Integration between the two approaches is required because C-MTFT uses Java synchronization mechanisms to determine access to shared memory while O-MTFT requires use of an STM to access shared memory. The C-MTFT approach could be modified to use the transaction commits to determine the order of access to shared memory, or if complete transparency was required, could replicate the use of CAS as the synchronization primitive (although this would presumably incur a significant performance penalty).

We believe that a better approach to recover when optimism fails is instead to maintain the transaction logs on stable storage. These logs need to be stable only when an output commit occurs (as identified by the side-effect handlers). The performance of such an approach depends primarily on the form of stable storage, which depends upon the failure model because stable storage must survive all possible failures. If we constrain the failures to the Capricious—that is, we are concerned with only corruption of shared memory when optimism fails—we can save the transaction logs on NVRAM [87], in the filesystem, or even in separate memory regions to provide high probability that the logs do not suffer from corruption due to errant writes. Further, during recovery writes of transactions can be compared with the logs to detect erroneous writes in order to clean the logs. In this manner, O-MTFT can be improved to recover from both errant and erroneous writes that occur with low probability.

Checkpointing can also be used to recover when optimism fails. However, it is not sufficient in general to replace C-MTFT with checkpointing unless a checkpoint is taken before every output commit. A checkpoint alone cannot guarantee that the state of the application at the output commit can be recovered if any non-deterministic events occur between the checkpoint and the output commit event. Checkpoints can still be used to reduce the recovery time of C-MTFT or as a summary of the transaction logs in O-MTFT in the case when optimism fails by moving forward the initial state from which replay begins. Instead of beginning replay at the initial state of the application, replay can begin from the checkpoint, which guarantees a correct and consistent state of the application. Care must be taken when the checkpoint is used as a shortcut for re-execution of the transaction logs that the checkpoint does not included corrupted shared data. In this case, it may be useful to keep several checkpoints in order to roll back further if corruption is detected in the most recent checkpoint.

# Chapter 7

# Conclusions

This thesis presents methods to improve the fault tolerance of multithreaded applications. We develop a model of multithreaded applications along with the Capricious failure model. Our failure model describes the complex failures of multithreaded applications provide some defensive programming techniques—that is, that *erroneous writes* of bad data can be detect and that *errant writes* to bad locations are unlikely. Under this model we develop two approaches to improve the fault-tolerance of multithreaded applications: conservative multithreaded fault-tolerance (C-MTFT) and optimistic multithreaded fault-tolerance (O-MTFT).

The C-MTFT technique applies transparently to multithreaded applications to replicate access to shared data. The replication can occur over time (as with a stable log) or over space (as with multiple hardware machines). Our approach captures the access to shared data to capture the non-determinism inherent in the access order of multithreaded applications. By either replicating the synchronization order or the order in which threads are scheduled, C-MTFT ensures that all replicas modify shared data in identical orders, resulting in the same values for all commands. We also describe our side-effect handlers that capture the necessary information to handle output commit events that are observed by the environment outside of the application. These handlers ensure that (for certain classes of output commit events) the state of the application after recovery will be consistent with the environment.

Although the C-MTFT technique is transparent, it incurs high overhead to capture and replicate the non-determinism with which it is concerned. Our O-MTFT technique assumes some parts of memory can be used as stable storage and uses an

STM to manage and record access to shared data. We discuss several implementations of multiversion STM that provide threads with lock-free access guarantees even in the presence of failed threads. Although our STM have higher overhead than using simple locks, they provide much stronger access guarantees. We also discuss the design of Mayfly, which uses the multiple versions of our STM are during recovery of single threads. A single thread re-executes transactions as necessary to recreate a local state that is consistent with the current shared state of the correct threads. Further, threads are created with leases on their execution time—both in CPU time and in wall-clock time—allowing hung threads to be detected and recovered.

# Bibliography

[1] B. Alpern, J.-D. Choi, T. Ngo, M. Sridharan, and J. M. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Proceedings of the Fifteenth Annual IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, page 23, Apr. 2001.

[2] L. Alvisi. *Understanding the message logging paradigm for masking process crashes.* PhD thesis, Cornell University, 1996.

[3] L. Alvisi, T. C. Bressoud, A. El-Khashab, K. Marzullo, and D. Zagorodnov. Wrapping server-side TCP to mask connection failures. In *Proceedings of the Twentieth Annual Conference of the IEEE Communications Society*, pages 329–337, Apr. 2001.

[4] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems (TOCS)*, 15(2):134–165, May 1997.

[5] J. Arlat, J.-C. Fabre, M. Rodríguez, and F. Salles. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 51(2):138–163, Feb. 2002.

[6] M. Baker and M. Sullivan. The recovery box: Using fast recovery to provide high availability in the UNIX environment. In *Proceedings of the Summer 1992 USENIX Annual Technical Conference*, pages 31–43, June 1992.

[7] J. Bartlett, J. Gray, and R. Horst. Fault tolerance in tandem computer systems. In A. Avizienis, H. Kopetz, and J.-C. Laprie, editors, *The Evolution of Fault-Tolerant Systems*, pages 55–76. Springer-Verlag, Vienna, Austria, 1987.

[8] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault injection experiments using FIAT. *IEEE Transactions on Computers*, 39(4):575–582, Apr. 1990.

[9] C. Basile, Z. Kalbarczyk, K. Whisnant, and R. Iyer. Active replication of multithreaded applications. Technical Report CRHC-02-01, University of Illinois at Urbana-Champaign, 2002.

[10] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. K. Iyer. Loose synchronization of multithreaded replicas. In *Proceedings of the Twenty-First Annual IEEE Symposium on Reliable Distributed Systems*, pages 250–255, Oct. 2002.

[11] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[12] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[13] M. Boosten, R. W. Dobinson, and P. D. V. van der Stok. MESH: MEssaging and ScHeduling for fine-grain parallel processing on commodity platforms. In *Proceedings of the International Conference on Parallel and Distributed Techniques and Applications (PDPTA)*, June 1999.

[14] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 195–205, July 2004.

[15] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the Sixteenth Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 56–69, 2001.

[16] T. C. Bressoud. TFT: A Software System for Application-Transparent Fault Tolerance. In *Proceedings of the Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 128–137, 1998.

[17] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the Fifteenth Annual ACM Symposium on Operating Systems Principles*, Dec. 1995.

[18] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot: A technique for cheap recovery. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation (OSDI)*, December 2004.

[19] P. Chan, R. Lee, and D. Kramer. *The Java Class Libraries: 2nd Ed, Vol 1 Supplement for the Java$^{TM}$ 2 Platform, Std Ed, v1.2*. Addison-Wesley, June 1999.

[20] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The rio file cache: surviving operating system crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 74–83, Oct. 1996.

[21] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1998.

[22] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded application. In *Proceedings of the Fifteenth Annual IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2001.

[23] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 258–269, June 2002.

[24] J.-D. Choi and H. Srinivasa. Deterministic replay of java multithreaded applications. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 48–59, Aug. 1998.

[25] C. Constantinescu. Trends and challenges in vlsi circuit reliability. *IEEE Micro*, 23(4):14–19, July-August 2003.

[26] J. DeVale and P. Koopman. Robust software - no more excuses. In *Proceedings of the Annual IEEE International Conference on Dependable Systems and Networks*, pages 145–154, 2002.

[27] E. W. Dijkstra.

[28] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.

[29] D. Engler and K. Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth Annual ACM Symposium on Operating Systems Principles*, pages 237–252, Oct. 2003.

[30] R. Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge, Jan. 2005.

[31] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[32] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, Sept. 2003.

[33] R. Friedman and A. Kama. Transparent fault-tolerant java virtual machine. In *Proceedings of the Twenty-Second Annual IEEE Symposium on Reliable Distributed Systems*, pages 319–328, Oct. 2003.

[34] A. P. Goldberg, A. Gopal, K. Li, R. E. Strom, and D. F. Bacon. Transparent recovery of mach applications. In *Proceedings of the 1990 Usenix Mach Workshop*, pages 169–183, 1990.

[35] C. G. Gray and D. R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth Annual ACM Symposium on Operating Systems Principles*, pages 202–210, Dec. 1989.

[36] J. Gray. The transaction concept: Virtues and limitations. In *Proceedings of the Seventh International Conference on Very Large Data Bases (VLDB)*, pages 144–154, Sept. 1981.

[37] J. Gray. A census of tandem system availability between 1985 and 1990. *IEEE Trans. on Reliability*, 39(4):409–418, Oct 1990.

[38] J. Gray and D. P. Siewiorek. High-availability computer systems. *IEEE Computer*, 24(9):39–48, 1991.

[39] K. C. Gross, S. McMaster, A. Porter, A. Urmanov, and L. G. Votta. Towards dependability in everyday software using software telemetry. In *Proceedings of the Third IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASE'06)*, pages 9–18. IEEE Computer Society Press, 2006.

[40] W. Gu, Z. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *Proceedings of the Annual IEEE International Conference on Dependable Systems and Networks*, pages 459–468, 2003.

[41] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. In *Proceedings of the Nineteenth Annual IEEE International Symposium on Distributed Computing (DISC)*, volume 3724, pages 303–323, Sept. 2005.

[42] R. Guerraoui, M. Kapalka, and J. Vitek. Stmbench7: A benchmark for software transactional memory. In *Proceedings of the Second European Systems Conference (EuroSys'07)*, pages 315–324, Mar. 2007.

[43] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 101–111, June 2007.

[44] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the Twenty-Second Annual International Conference on Software Engineering (ICSE)*, pages 291–301, May 2002.

[45] T. Harris. Exceptions and side-effects in atomic blocks. In *PODC Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, July 2004.

[46] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, 2003.

[47] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 14–25, June 2006.

[48] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the Fifteenth Annual IEEE International Symposium on Distributed Computing (DISC)*, volume 2180, pages 300–314. Oct. 2001.

[49] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[50] M. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structure. In *Proceedings of the Sixteenth Annual IEEE International Symposium on Distributed Computing (DISC)*, volume 2508, pages 339–353. Oct. 2002.

[51] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the Twenty-Third International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, May 2003.

[52] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 92–101, July 2003.

[53] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[54] C. Hoare and R. H. Perrott, editors. *Towards a theory of parallel programming*, volume 9 of *Operating Systems Techniques*, pages 61–71. Academic Press, 1972.

[55] IBM. Jikes RVM, 2002. http://www.ibm.com/developerworks/oss/jikesrvm/.

[56] B. Igou and R. Silliman. User survey: High-availability and mission-critical services, north america. Published on WWW, Apr.

2005. http://www.gartner.com/resources/127000/127016/user_survey_highavailability_127016.pdf.

[57] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 151–160, Aug. 1994.

[58] T. Jarboui, J. Arlat, Y. Crouzet, and K. Kanoun. Experimental analysis of the errors induced into linux by three fault injection techniques. In *Proceedings of the Annual IEEE International Conference on Dependable Systems and Networks*, pages 331–336, 2002.

[59] Java specification request 51: New i/o apis for the java$^{TM}$ platform (final release). Published on WWW, May 2002. http://www.jcp.org/en/jsr/detail?id=51.

[60] M. Kalyanakrishnam, Z. Kalbarczyk, and R. Iyer. Failure data analysis of a lan of windows nt based computers. In *Proceedings of the Eighteenth Annual IEEE Symposium on Reliable Distributed Systems*, pages 178–187, Oct. 1999.

[61] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *Proceedings of the Fourteenth Annual IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 219–228, May 2000.

[62] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2), June 1981.

[63] L. Lamport. Time, clocks, and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[64] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, Sept. 1979.

[65] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 1982.

[66] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, 1987.

[67] I. Lee and R. K. Iyer. Faults, symptoms, and software fault tolerance in the tandem GUARDIAN operating system. In *Proceedings of the Twenty-Third Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 20–29, 1993.

[68] I. Lee and R. K. Iyer. Software dependability in the tandem GUARDIAN system. *IEEE Transactions on Software Engineering*, 21(5):455–467, May 1995.

[69] Y.-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, Jan. 1999.

[70] S. Liang. *The Java$^{TM}$ Native Interface: Programmer's Guide and Specification*. Addison-Wesley, June 1999.

[71] B. Liblit, A. X. Zheng, M. Naik, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2005.

[72] T. Lindholm and F. Yellin. *The Java$^{TM}$ Virtual Machine Specification, 2nd Ed.* Addison-Wesley, April 1999.

[73] C. Liu, X. Yan, H. Yu, J. Han, and P. S. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *Proceedings of SIAM Data Mining Conference (SDM 05)*, 2005.

[74] D. E. Lowell and P. M. Chen. Free transactions with rio vista. In *Proceedings of the Sixteenth Annual ACM Symposium on Operating Systems Principles*, pages 92–101, Oct. 1997.

[75] W. lun Kao, R. K. Iyer, and D. Tang. FINE: A fault injection and monitoring environment for tracing the unix system behavior under faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, Nov. 1993.

[76] H. Madeira, D. Costa, and M. Vieira. On the emulation of software faults by software fault injection. In *Proceedings of the Annual IEEE International Conference on Dependable Systems and Networks*, pages 417–426, 2000.

[77] V. J. Marathe, W. N. S. III, and M. L. Scott. Adaptive software transactional memory. In *Proceedings of the Nineteenth Annual IEEE International Symposium on Distributed Computing (DISC)*, volume 3724, pages 354–368, Sept. 2005.

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

[79] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 21–30, July 2002.

[80] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, May 1996.

[81] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Magazine of Security and Privacy*, pages 33–39, July/August 2003.

[82] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high performance microprocessor. In *Proceedings of the Thirty-Sixth Annual International Symposium on Microarchitecture (MICRO)*, pages 29–42, Dec. 2003.

[83] J. Napper and L. Alvisi. Lock-free serializable transactions. Technical Report CS-TR-05-04, The University of Texas at Austin, Department of Computer Sciences, Feb. 2005.

[84] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant java virtual machine. Technical Report TR02-56, University of Texas, Dept. of Computer Sciences, May 2002.

[85] J. Napper, L. Alvisi, and H. Vin. A fault-tolerant java virtual machine. In *Proceedings of the Annual IEEE International Conference on Dependable Systems and Networks*, pages 425–434, 2003.

[86] W. T. Ng and P. M. Chen. Integrating reliable memory in databases. In *Proceedings of the Thirty-First International Conference on Very Large Data Bases (VLDB)*, pages 76–85, Aug. 1997.

[87] W. T. Ng and P. M. Chen. The systematic improvement of fault tolerance in the rio file cache. In *Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing (FTCS)*, 1999.

[88] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 167–178, June 2003.

[89] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, Mar. 2003.

[90] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Presentation given at the 1996 USENIX Annual Technical Conference, Jan 1996.

[91] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[92] F. Pizlo, M. Prochazka, S. Jagannathan, and J. Vitek. Transactional lock-free objects for real-time java. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[93] A. Podgurski, D. Lean, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the Twenty-Third Annual International Conference on Software Engineering (ICSE)*, pages 465–475, May 2003.

[94] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, 1(2):220–232, 1975.

[95] C. P. Research. 2001 cost of downtime online survey. Published on WWW, 2001. `http://www.contingencyplanningresearch.com/2001Survey.pdf`.

[96] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the Twentieth Annual IEEE International Symposium on Distributed Computing (DISC)*, volume 4167, pages 284–298, Sept. 2006.

[97] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT '06)*, New York, NY, USA, June 2006. ACM Press.

[98] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic race detector for multi-threaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, Oct. 1997.

[99] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[100] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the Annual IEEE International Conference on Dependable Systems and Networks*, pages 249–258, 2006.

[101] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, Aug. 1995.

[102] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on soft error rate of combinational logic. In *Proceedings of the Annual IEEE International Conference on Dependable Systems and Networks*, pages 389–398, 2002.

[103] D. P. Siewiorek, R. Chillarege, and Z. T. Kalbarczyk. Reflections on industry trends and experimental research in dependability. *IEEE Transactions on Dependable and Secure Computing*, 1(2):109–127, Apr. 2004.

[104] J. H. Slye and E. N. Elnozahy. Support for software interrupts in log-based rollback-recovery. *IEEE Transactions on Computers*, 47(10):1113–1123, 1998.

[105] M. Spear, V. Marathe, W. S. III, and M. Scott. Conflict detection and validation strategies for software transactional memory. In *Proceedings of the Twentieth Annual IEEE International Symposium on Distributed Computing (DISC)*, volume 4167, Sept. 2006.

[106] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(3):204–226, 1985.

[107] M. Sullivan and R. Chillarege. Software defects and their impact on system availability - a study of field failures in operating systems. In *Proceedings of the Twenty-First Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 2–9, 1991.

[108] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *Proceedings of the Twenty-Second Annual International Symposium on Fault-Tolerant Computing (FTCS)*, pages 475–484, July 1992.

[109] M. Sullivan and M. Stonebraker. Using write protected data structures to improve software fault tolerance in highly available database management systems. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of the Seventeenth International Conference on Very Large Data Bases (VLDB)*, pages 171–180, Sept. 1991.

[110] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems*, May 2003.

[111] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *Proceedings of the Nineteenth Annual ACM Symposium on Operating Systems Principles*, pages 268–281, Oct. 2003.

[112] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth Annual ACM Symposium on Operating Systems Principles*, pages 203–216, Dec. 1993.

[113] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *European Conference on Object-Oriented Programming (ECOOP)*, June 2004.

[114] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth Annual ACM Symposium on Operating Systems Principles*, pages 230–243, Oct. 2001.

[115] I. William N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 240–248, July 2005.

[116] J.-M. Y. D.-F. Z. X.-D. Yang. User-level implementation of checkpointing for multithreaded applications on windows nt. In *12th Asian Test Symposium (ATS)*, pages 496–499, Nov. 2003.

[117] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Twentieth Annual ACM Symposium on Operating Systems Principles*, pages 221–234, Oct. 2005.

[118] J. F. Zeigler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1):19–39, Jan. 1996.

# Vita

Jeff Napper was born in Houston, Texas in the 1970s, which one might think puts a certain stamp on a person. He attended the University of Texas for his undergraduate and as is to be expected, got stuck in wonderful Austin, Texas. After graduation in 1997, he worked a year at Applied Research Labs for the University of Texas when he decided to go to graduate school to seek his fortunes. Clearly immune to fiduciary gain, he remained in graduate school throughout the Internet boom of the early 2000s. Luckily, he met the wonderful Sarah Brown, and she graciously accepted his other proposal. After graduation, he will take a postdoc position at Vrije Universiteit Amsterdam.

Contrary to the expectations of most people outside of Texas that he meets, Jeff has no discernible Texas accent.

Permanent Address: 2215 S. Belmont
Richmond, TX 77469
USA

This dissertation was typeset with LaTeX $2_\varepsilon$[1] by the author.

[1]LaTeX $2_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended

by Bert Kay, James A. Bednar, and Ayman El-Khashab.